NOVA IMS Information Management School

MAA

Mestrado em Métodos Analíticos Avançados
Master Program in Advanced Analytics

Single layer optimization with Particle Swarm Optimization

A new approach to optimize Deep Neural Networks

Guilherme de Oliveira Crespo Martins

Dissertation presented as partial requirement for obtaining the Master's degree in Advanced Analytics

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

SINGLE LAYER OPTIMIZATION WITH PARTICLE SWARM OPTIMIZATION

by

Guilherme Martins

Dissertation presented as partial requirement for obtaining the Master's degree in Advanced Analytics

Advisor: Mauro Castelli

Co Advisor*:* Illya Bakurov

ACKNOWLEDGEMENTS

I would like to thank...

My teachers at Nova IMS from which I learned so much over the last years, in particular my supervisor, Prof. Dr. Mauro Castelli and Prof. Illya Bakurov for helping me develop this thesis and, along with Prof. Dr. Leonardo Vanneschi, for sparking my interest in the topic of Machine Learning.

My parents for having the patience and dedication to support me over the last 24 years and to whom I owe everything.

ABSTRACT

Deep Neural Networks attempt to simulate the behaviour of the brain to solve complex problems. Today, they are currently used for various real world applications such as natural language processing, image recognition, self-driving cars, and much more. However, these models, can be very computationally expensive and take a considerable amount of time to train. In this thesis, we attempt to use swarm intelligence to optimize Deep Neural Networks with a smaller computational budget. To achieve this goal, we implement a method that takes any model and selects the layer that can contribute the most for the optimization of said model. Afterwards, we further optimize the layer selected with the Particle Swarm Optimization algorithm in an attempt to take advantage of its ability to surpass local optimums.

Keywords

machine learning; deep neural networks; particle swarm optimization; gradient descent; computer vision

Contents

List of Figures

List of Tables

List of Abbreviations

ANN Artificial Neural Networks

ADAM Adaptive Moment Estimation

CNN Convolutional Neural Networks

DL Deep learning

DNN Deep Neural Networks

ML Machine learning

PSO Particle Swarm Optimization

SGD Stochastic Gradient Descent

1 INTRODUCTION

1.1 Deep Learning

The field of Deep Learning (DL) is one whose origins can be traced back to the 20th century. Researchers had been striving to create models that could interpret abstract concepts such as the meaning of sentences and image recognition. Computers could easily solve complex mathematical problems, problems that were difficult for humans, yet struggled to solve these seemingly easy tasks such as recognizing a dog in a picture, or explaining what was happening in a movie.

The reason humans can solve the aforementioned abstract tasks easily is due to the fact that we build a knowledge bank throughout our lifetime. Researchers attempted to find a shortcut path to this understanding by giving models some basic rules from which, they hoped, computers could emerge knowledge. This approach, also known as the knowledge based approach, has largely failed due to the high complexity of the world around us. For example, the Cyc, a rule based model [9] failed to understand what would be considered a simple description to us. This description stated that a person named Fred, was shaving his beard in the morning, with an electric razor. When analyzing this sentence, Cyc detected an inconsistency: it knew that people do not have electrical parts. However, because Fred was holding an electric razor, it believed the entity "FredWhileShaving" contained electrical parts. It therefore asked whether Fred was still a person while he was shaving. For a human it is easy to discern that Fred and the electric razor are two separate entities. But to formulate this knowledge in a set of general enough rules is a difficult proposition, particularly without the experience we have acquired through our lifes.

The solution to these problems, it turned out, was to allow computers to learn from experience, much like we do, and understand the world in terms of a hierarchy of concepts, as shown in Figure 1, with each concept defined through its relation to simpler concepts. This approach bypasses the need to specify rules and knowledge into the computer by gathering knowledge from experience. The models built with this approach can construct this hierarchy of concepts through a series of layers with high degrees of depth. This approach was therefore named *deep learning*.

In 1989, Yan LeCunn produced the first practical demonstration of one of these models by combining neural networks with back propagation to read handwritten numbers. This

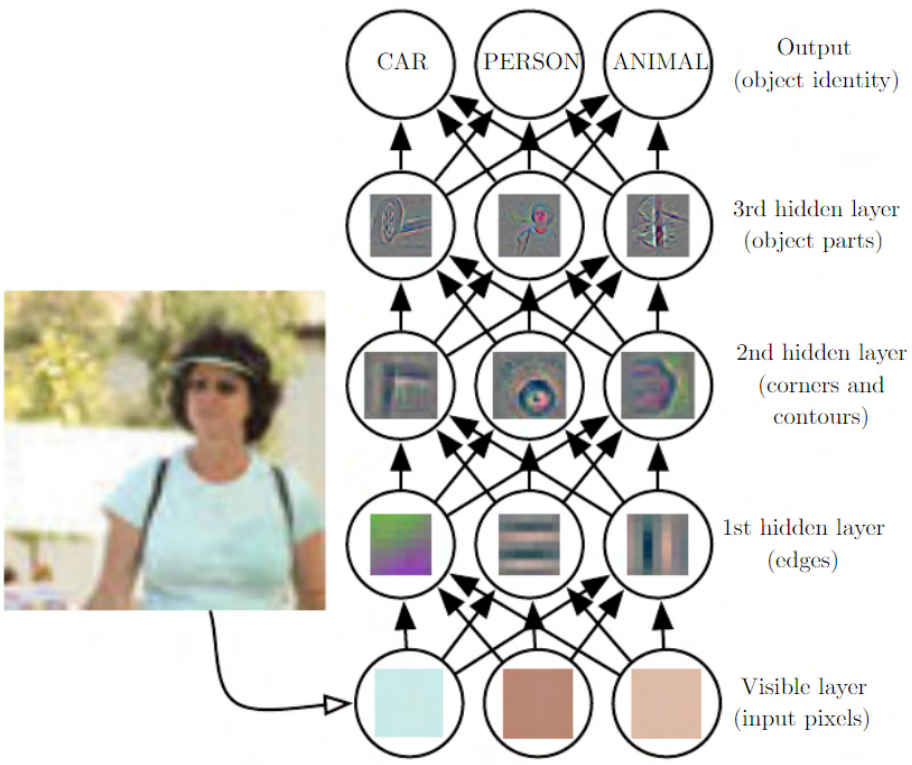

Figure 1: Hierarchy of concepts in a deep learning model (Source: [1])

system would go on to be used to read the numbers of handwritten checks. Decades later, the advancement in computation technology, namely transistor size reductions and random access memory size increases, has made the use of these models in practical settings very much a reality with a number of companies creating or improving existing products with artificial neural networks (ANNs).

1.2 Challenges of deep learning problems

A common characteristic of the learning algorithms (for example backpropagation) that DL systems use is that these algorithms tend to get stuck in local minimum points in the fitness landscape. These points provide a challenge to researchers in that they keep models from achieving the global optimum, i.e. the best solution possible. In the fitness landscape of a neural network this challenge is even more exacerbated by the number of local optimum that there tends to exist and the extremely high dimensional search space.

To better navigate this landscape several optimization algorithms have been proposed to improve upon the original Stochastic Gradient Descent (SGD) such as the introduction of

momentum [10], or algorithms such as AdaGrad [11] and adaptive moment estimation (Adam) [12]. However, despite of these improvements, training durations of DL models are often both time and computationally expensive. This arises from the complexity of the problems as well as the deepness of the models used which can still get stuck on local optima for several training epochs. Many attempts have tried to reduce this training time either by optimizing or improving existing processes like in [13], [14], [15], or by creating completely new methods or hybrids of existing ones.

Evolutionary algorithms and Particle Swarm Optimization (PSO) have the capacity to reduce some drawbacks of gradient based optimization such as getting stuck in a local minima and slow convergence rates [16]. And yet, as we will see, the use of these algorithms to perform deep neural network weight optimizations is lacking. The reason for this seems clear; the enormous amount of parameters DL models have, make the fitness landscape extremely complex [17]. However, swarm intelligence algorithms do provide better ability to leave local optimum whilst still being able to progress towards better fitness values which may give these algorithms an advantage when attempting to escape local optimum points. Furthermore, it is possible to adjust the exploration and convergence abilities of the swarm intelligence algorithms which gives them a slight edge over gradient based algorithms in this regard.

1.3 Objectives

Our objective in this thesis is to test the optimization of deep neural networks (DNNs) through the application of swarm intelligence in the form of the PSO algorithm, in order to attempt to reach a global or improved local optimum in a more computationally efficient way. As stated previously, evolutionary and swarm intelligence algorithms don't do well with high amounts of dimensions. We tackle this issue by selecting a specific layer to optimize in the target network, substantially reducing the number of parameters to optimize. Our goal can, therefore, be divided into two parts. Firstly, we intend to find a method of selecting a layer in the network that has a high impact on the performance of the network. Secondly, we will explore the best combination of algorithm and hyperparameters to optimize the target network, whilst comparing its performance to the optimization of the entire network with the predetermined gradient based optimization algorithm. More specifically we will focus on parameters such as the ANNs learning rate, positional and velocity clamping of particles, cognitive and social factors, inertia and population size. In order to complete this exercise, we will first train the chosen models until they reach a plateau in loss. Afterwards, we will apply the layer selection methods to get

our target layer, which will then be passed to our optimization algorithms.

2 THEORETICAL BACKGROUND

2.1 Deep Learning

In the field of machine learning (ML), DL achieves great power and flexibility by representing the world in a nested hierarchy of concepts. It has already proven itself useful in a variety of practical situations and disciplines such as computer vision, speech and audio processing, robotics, finance, etc. The most classical model of DL, and one which forms the basis to many commercial applications is called Deep FeedForward Network. Its name is given from the fact there exists no feedback connections between the layers that constitute its architecture. Like most neural networks, Feedforward networks are composed of one input layer, n hidden layers and one output layer. The *neurons* that form these layers are loosely inspired by human neurons and, in its most basic form commonly called perceptron, receive $n + 1$ inputs, n data values and one *bias* to which different weights are applied. The sum product of these inputs and weights is then passed to an activation function, sigmoid for example, and the neuron outputs one single value as shown in Figure 2. These basic structures can be thought of as no more than a non linear extension to a linear model, with the activation function adding non linearity to the linear sum product operation.

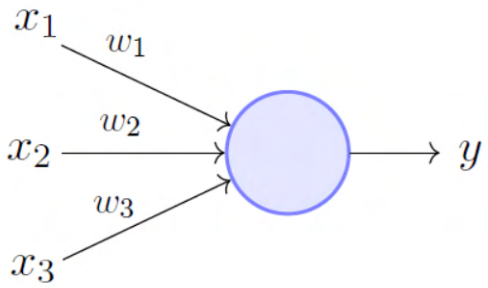

Figure 2: Perceptron (Source: [2])

By looking into the above described layers we would find thousands of the previously mentioned neurons with the sole difference being the number of inputs and outputs associated with each one. For example, considering a model with 3 layers, one input (l_i), one hidden(l_h) and one output(l_o), with an input size of 100 values, we could immediately affirm that each neuron in l_i contains exactly 100 input values plus 1 bias, totaling 101 input values. To determine

the size of the output of a neuron in l_i we would have to define the size, i.e. the number of neurons, in l_h. These multi-layer networks, are often called multi-layer perceptrons. By finding an appropriate set of weights and activation functions, it is proven that a multi-layer perceptron can approximate any smooth, measurable function between the input and output vectors [18].

2.1.1 Learning to learn

When solving a classification problem for example, we may think to set the accuracy as the target measure that our model needs to improve upon. However, ANNs optimize a different cost function $L(o)$ that indirectly optimizes the original performance measure. This cost function (cross-entropy for instance) is used instead of our actual measure because it allows the model to increase its certainty. For example, a model that, while "looking" at an image of a cat, returns an output that it is 51% sure it is a cat and 49% sure it is a dog would be as accurate as one that returns an 80% probability that the image is a cat. However, we can say that the second model is considerably better. A cost function typically considers how certain a model is of its answers and not just whether it is right or wrong. During training, the chosen performance measure is optimized with a training set of the data and the generalization ability of the model is expected to improve given a set of assumptions. Firstly that the examples in each dataset are independent from each other, and secondly that both the training and the test set are identically distributed. These assumptions imply that there is a common underlying distribution between the train set and test set and, therefore, that by learning this underlying distribution given the training set, it is possible to improve the performance of the model on unseen data present in the test set.

An important concept to understand and visualize in the field of DL and ML in general, is that of the fitness landscape. The fitness landscape can be thought of as a three dimensional space on which the x and y coordinates define a specific set of weights and bias of a model and the z axis represents the value of the cost function for (x,y) coordinates as shown in Figure 3.

In this particular image, the z axis represents the fitness F that can be described as the negative of the cost function or $F(o) = -L(o)$.

Although in practice the fitness landscape is an $n + 1$ dimensional search space, with n being the number of parameters. By thinking of the fitness landscape in this way we can imagine a mechanism that improves model performance. This fundamental component that drives most optimization methods for DL today is the concept of the derivative. The derivative allows us to determine the direction and the inclination of the slope at the position of a given

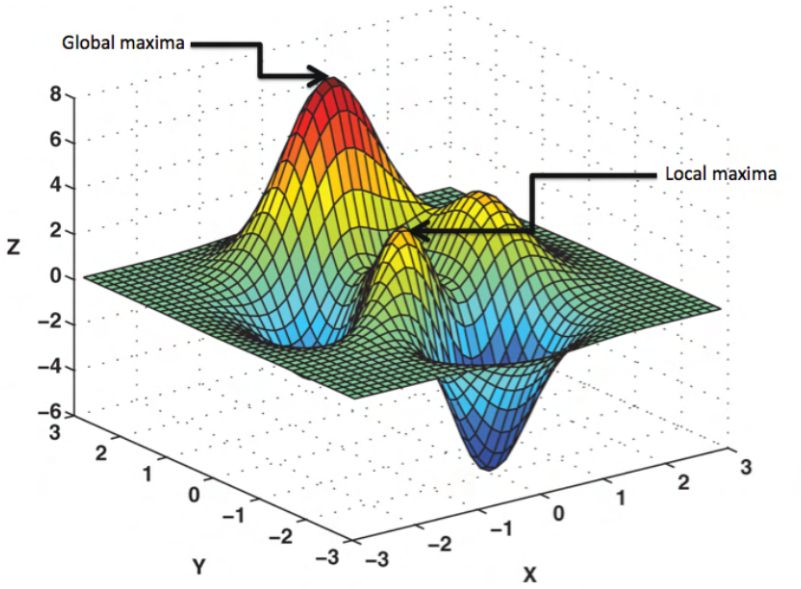

Figure 3: A simple fitness landscape (Source: [3])

point. It informs the way in which to change x (in this case, layer weights) in order to reduce y (our cost function). The use of derivatives to improve model's performance is called gradient descent and was first published in [19]. This method, described by the formula in (1), introduces a velocity to the weight w equal to the derivative of the loss function with respect to the given weight, that is then regulated by a learning rate η, a value usually between 0 and 1.

$$v_t = \eta \nabla L(w) \tag{1}$$

$$w_{t+1} = w_t - v_t \tag{2}$$

However using all samples of the training set to calculate the loss for each training step would be computationally intensive. In order to accelerate this process, instead of using all samples, a single data point or a subset of the training set can be used. This method is called Stochastic Gradient Descent and is what many consider the base optimization method for DL. Since this first approach to the optimization of ANNs, several improvements were made to deal with different problems that were faced. In 1964 the concept of momentum was introduced in the field in [20] which improved the performance of models particularly in certain scenarios often visualized as ravines in the fitness landscape. In these situations, the typical behaviour of the SGD has trouble reaching the center of said ravine in a straight forward fashion instead oscillating around the edges. By introducing a percentage β of the $t-1$ velocity, as shown in formula 3, this method partially conserves the direction of the previous gradient. This modified

behaviour would enable a point to move to the center of a ravine with less oscillation.

$$v_t = \beta v_{t-1} + \eta \nabla L(w) \tag{3}$$

Despite its improvements, the SGD with momentum still had its issues. The same properties of the method that would enable it to converge quicker on a optimum could also overshot the target causing the solution to start climbing uphill. Ideally, the optimization method should be somewhat aware that with its current momentum it will miss the optimal point. To address this problem the Nesterov accelerated gradient was introduced [21]. This algorithm attempts to give the optimizer this knowledge by calculating the gradient not with respect to the current position, but instead with respect to an approximate future position as shown in 4.

$$v_t = \beta v_{t-1} + \eta \nabla L(w - \beta v_{t-1}) \tag{4}$$

These previous methods and many others like them, however, failed to dynamically adapt the method itself to the input data the algorithm was receiving. In [11] researchers John Duchi, Elad Hazan and Yoram Singer introduced a new algorithm that implemented this characteristics. AdaGrad, as they called it, assimilates information seen in previous iterations to inform the size of the learning rate. To achieve this, each parameter update is altered to divide the learning rate by a component D_t, defined in (7), that adapts the learning rate based on previous gradients.

$$g_t = \eta \nabla L(w_t) \tag{5}$$

$$G_t = \sum_{\tau=1}^{t} (g_\tau * g_\tau) \tag{6}$$

$$D_t = \sqrt{\epsilon + G_t} \tag{7}$$

$$v_t = \frac{\eta}{D_t} \nabla L(w_t) \tag{8}$$

Here, each weight w_t is updated by the product of the adjusted learning rate and the gradient of the w_t at time t. The parameter ϵ is an extremely small value that prevents divide by zero errors. Despite its advantages, one major problem with this approach was that the learning rate would approach zero as the sum of gradients increased with iterations, at which point the model could not learn any new information. To counter this behaviour, researchers in

[22] proposed AdaDelta, a method that recursively calculates past averages of gradients. More specifically, considering:

$$g^2 = g_\tau * g_\tau \tag{9}$$

We get the recursive average:

$$A[g^2]_t = \gamma A[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{10}$$

Which replaces the parameter D_t in the denominator of the update function which becomes:

$$D_t = \sqrt{\epsilon + A[g^2]_t} \tag{11}$$

Finally, in [12], the Adam algorithm is introduced. In addition to calculating the average of past gradients, an exponentialy decaying average is also accounted by

$$E[g]_t = \gamma E[g]_{t-1} + (1-\gamma)g_t \tag{12}$$

Which changes the update to:

$$v_t^i = \frac{\eta}{\sqrt{A[g^2]_t + \epsilon}} E[g]_t \tag{13}$$

This difference between Adagrad and Adam enables the latter to converge faster in Convolutional Neural Networks (CNNs) as shown in [12].

2.1.2 Batch normalization

As described previously, it is advantageous to ANNs that the distribution of the training data has a similar distribution to the test data in order to ensure proper learning. This property also applies to each particular input of each layer in the network. After all, due to its recursive nature, the formula that describes a neural network is similar to the one that describes a single layer. For example, a simple two layered network can be described by the formula 14, where the input i is fed to $F1$ with weights Θ_1. However if we simplify the formula so that $F1(i, \Theta_1) = x$ we end up with 15 which means that the gradient descent step of this second equation is equivalent to that of the whole network. With this demonstration we can say that the input distribution characteristics are also valid at the layer level.

$$L = F2(F1(i, \Theta_1), \Theta_2) \tag{14}$$

$$L = F2(x, \Theta_2) \tag{15}$$

Yet, the input distribution of a specific layer is always changing due to the variations in the output of previous layers. In order to fix this problem a Batch Normalization layer is introduced in between convolutional layers in order to keep the distribution of the inputs constant throughout the training of the neural network. As described in [23], this layer uses the mean and standard deviation of each feature over a batch size and applies a scaling (γ) and shifting (β) factor, formalizing the equation in 17.

$$x = \frac{x - E[x]}{\sqrt{Var(x)}} \tag{16}$$

$$y = \gamma x + \beta \tag{17}$$

2.1.3 Convolution

In many real world scenarios, data has more than one dimension. For instance, in problems such as image recognition, a picture can be processed as a two dimensional input. These types of problems, in order to be solved, require, then, a model that has to be proportionally bigger, which presents several challenges with respect to memory usage and computation power. In order to mitigate these problems, convolutional layers were introduced into DL models. More classical ANNs typically connect every input to every output, this approach however would prove extremely intensive on the computer's resources, to reduce this complexity, a convolutional layer has sparse interactions. Instead of connecting every input with every output, convolutional layers filter input layers with a (usually) smaller window, more commonly know as a kernel (Figure 4). In practice, this means we can store less parameters and perform fewer operations. The efficiency improvements can be quite large as a matrix multiplication performed in regular ANNs takes $O(m * n)$ while if we reduce the number of connections to k we end up with $O(k * n)$. Another dimension in which convolutional layers improve efficiency has to do with parameter sharing. In a traditional neural network, each connection from n_l to n_{l+1} has its own weight w, inside a convolutional layer however, weights inside the kernel are used across the entire input layer significantly reducing the number of parameters. If we think of an input layer l_0 as a matrix of 30 rows by 30 columns connecting to a layer l_1 with size 20x20, in a regular neural network we would end up with $30^2 * 20^2$ weights, while a convolutional layer with 5x5 kernel would have 25 weights.

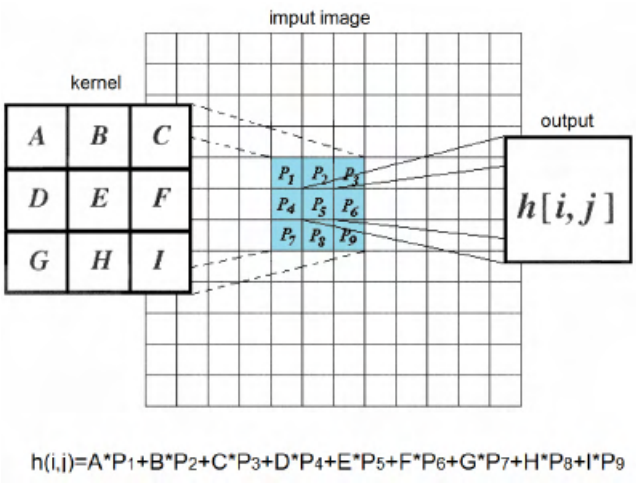

Figure 4: A 3x3 kernel (Source: [4])

2.1.4 Residual Neural Networks

Introduced in [8], the Deep Residual Network (ResNet) is a convolutional neural network that introduces a residual framework into the network. As the network gets deeper, the gradient needs to pass more and more multiplications to get to the layers closer to the input. As it does the gradient can get smaller and smaller until it approaches 0 and the network can no longer improve and, in some cases, will even degrade in performance. This problem is also refered to as the vanishing gradient problem. Similarly, with network deepness, input information from the samples as well as from earlier layers, gets lost with the several weight multiplications and sums it goes throughout the network until the final output layer. To solve these issues, the research team found that it is easier to optimize a residual of the target function than the actual target function. More specifically, if we consider a block of 2 consecutive convolutional layers that would normally optimize an identity function $F(x)$, and instead added a shortcut connection that would subtract the input to the output of the 2 convolutional layers, we would make it so the convolutional block optimizes for $F(x) - x$, as shown in Figure 5, which in essence gives additional information about the input data to deeper layers.

This method allowed the researchers to achieve better results compared with other models with similar depth.

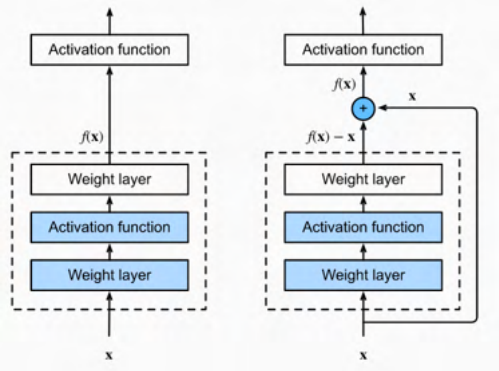

Figure 5: Comparison between a normal and a residual block (Source: [5])

2.2 Layer Importance

In ANNs, several authors propose to use calculate the importance of layers and neuron connections in order to then prune the network's architecture, removing the least important components of the model. In [24] the authors propose a novel method to identify which layers to remove from the network by applying a scaling factor the each individual layer in the network. This scaling factor is set to 1 and trained along side the weights of the network. With enough training, scalars attached to layers considered less useful will approach 0, effectively neutralizing them. One advantage of this approach is that it can use the scaling factors present in batch normalization, which is a type of layer already embedded in many of the currently used CNNs, and therefore cut down on the added computational costs.

2.3 Particle Swarm Optimization

Particle Swarm Otimization (PSO) is a search algorithm in which a population of particles moves to find an optimal point in its environment (search space). The origins of the PSO can be traced back to the *Boid* (Bird-oid) model designed by Reynolds in 1987 [25]. Inspired by the movement of flocks, a system was created where several particles, named *boids*, in a coordinate system follow three base rules from which flock behaviour emerges. The *boids* should firstly avoid collisions with nearby flockmates, secondly they should match velocities with nearby *boids*, and finally they should also attempt to stay close to the center of the flock. Later on, Heppner designed a model were the flock would move towards a pre-determined point, a "roost". This was achieved by adding a homing and a randomness variable. The homing variable determined the attraction the *boids* felt to the "roost", whilst the random variable introduced some variation

to the flocks behavior, similar to what would be observed in the wild. This approach to the problem of simulating flock behavior led researchers to ask questions about how birds would find food. Unlike the "roost" in Heppner's simulation, the flock of birds would not know the location of food. Yet, if a bird feeder is placed within hours, birds will come to its location. This behavior led researchers to suspect that birds had ways of capitalizing on each other's knowledge. Based on this thought process, James Kennedy introduces the "cornfield vector" [18]. Each *boid* would evaluate its current position based on equation (18) and would remember their past best position as well as its value. Moreover, each *boid* would also remember the best position that any *boid* of the flock had reached.

$$Eval = \sqrt{(presentx - 100)^2} + \sqrt{(presenty - 100)^2} \tag{18}$$

The results from this approach had surprisingly good results. The flock would very quickly converge on the point of lowest score, which in this case was the point at (100,100). More importantly, it proved that the algorithm could find the optimal point without giving the point's location to the flock in advance. After further research and experiments, it was found that the algorithm would work just as well and look just as realistic without the added randomness variable, so it was removed. Next velocity matching was also found to slow the speed of convergence, altought it made the movement of the *boids* look more realistic it to was removed. What was left was an algorithm that could effectively find the best solution to a variety of problems. Following this research, in 1995 Kennedy and Eberhart in [18] introduce the PSO algorithm. As described previously, in this algorithm particles with no mass or volume represent solutions to the problem that is being solved and move through the solution space influenced by a social and a cognitive behaviour. However, before describing their movement in this search space it is necessary to define how they are initialized and how many of these particles need to exist. The solution to the first question is quite simple. All particles start with a randomized position and velocity within the search space that can be constrained to some degree. As suggested in [26] and [27] particles should always be constrained to a maximum velocity defined as v_{max}. This constraint serves as insurance that the particles don't become so fast that they move past optimums and keeps particles consigned to more local search space. If, on the other hand v_max is too low the particle's movement becomes limited and may cause particles to not explore sufficiently beyond local solutions [26].

In [18], the researches introduce a velocity limit $v_{max,j}$ that limits the particles's velocity in dimension j according to the formula in 19.

$$v_{i,j}^{t+1} = \begin{cases} v_{i,j}^{t+1} & \text{if } -v_{max,j} \leq v_{i,j}^{t+1} \leq v_{max,j} \\ v_{max,j} & \text{if } v_{max,j} < v_{i,j}^{t+1} \\ -v_{max,j} & \text{if } v_{i,j}^{t+1} < -v_{max,j} \end{cases} \tag{19}$$

However in a situation where the velocity of a specific particle hits the limit in one or more dimensions, with this approach, its direction would be changed. For example, if a particle had a velocity vector with values $[2, 6, 3, 4]$ and the limit for all dimensions was 5 then the velocity vector would be changed to $[2, 5, 3, 4]$. This problem would be a characteristic of all environments with a number of dimensions bigger then 1. Another approach, as described in [28], is to limit the magnitude of the velocity vector to some v_{max}. When this limit is surpassed, the velocity vector is updated according to equation 20 which reduces the particle's velocity by a factor without changing its direction.

$$v_i^{t+1} = \begin{cases} v_i^{t+1} & \text{if } |v_i^{t+1}| \leq v_{max} \\ \frac{v_{max}}{|v_i^{t+1}|} v_i^{t+1} & \text{if } |v_i^{t+1}| > v_{max} \end{cases} \tag{20}$$

In some cases it is also beneficial to constrain the search space itself, this can be either due to some solution imposed restrictions or if the optimum solution is known to be within a confined location in the search space. In [27], researchers suggested that, particles that exceed this domain are re-randomized. This re-randomization is applied to both position and velocity. However, optimal positions in the search space will be hard to reach with this definition since particles that move even slightly of the domain previously defined get moved to a completely random location. A possible solution to this problem could be to push particles that hit the "walls" in the opposite direction by a percentage of the speed they previously had. The second question is tied to the problem's complexity which can be defined by the number of local optimums, the dimensions of the search space and some possible constraints. In [29], Piotrowski et al. suggest population sizes between 50 to 300 particles should be used in real world problems. With these questions asked we can begin to understand how particles in the PSO move. The velocity of each particle is influenced by two factors: social and cognitive. The cognitive factor represents the particle's ability to always remember its best past position. This position is defined by researchers as $pbest_i$ which contains the best past position of particle i. In similar fashion, the social factor represents the swarms ability to communicate information with each other and contains the best position of any particle in the swarm. The set of coordinates values associated to this global best solution is defined as $gbest$. These two criteria will deter-

mine much of the path the particles wil take during the optimization process. Each particle i is expressed in the search space with two characteristics which are mathematically defined as vectors. The $x_i(t)$ defines the particles position at time t and the vector $v_i(t)$ define the particle's velocity. As described earlier, position changes of the particles are affected by the previously defined $pbest_i$ and $gbest$. This update is done by calculating the distance between these coordinates and the current position of each particle at iteration t and integrating these values into the calculation of the updated velocity vector. More specifically, the velocity vector of particle i at time t is updated by taking to account the distance between the coordinates associated with the personal best value, i.e., $pbest_i$, and the current position of the particle, weighed by $R1$, a random value within the range of 0 and 1 generated by a uniform distribution at each iteration, and by $c1$, a positive valued constant named the cognitive acceleration constant. This second component represents the linear attraction towards the best position ever found by particle i $pbest_i$. The distance between the coordinates associated with the global best value, i.e., $gbest$, and the current position of the particle, weighed by $R2$, a random value within the range of 0 and 1 generated by a uniform distribution at each iteration, and by c2, a positive valued constant named the social acceleration constant. This last component is described as the linear attraction towards the best position ever found by any particle in the group, $gbest$, thus named "social knowledge". The velocity equation is then defined as described in (21)

$$v_i(t) = v_i(t-1) + c1 * R1 * (pbest_i - x_i(t-1)) + c2 * R2 * (gbest - x_i(t-1)) \quad (21)$$

The position of each particle i, at time t is then determined by the sum of the previous position vector $x_i(t–1)$ and the updated velocity vector $v_i(t)$ computed by (21) as shown in equation (22).

$$x = x + v_x \quad (22)$$

Assuming a two-dimensional search space, one can visualize a movement of a particle from $t-1$ to t as shown in Figure 6.

This version of the PSO however was not a very effective tool for optimization problems and in 1998 Shi and Eberhart [30] proposed a new formula that introduced the concept of inertia.

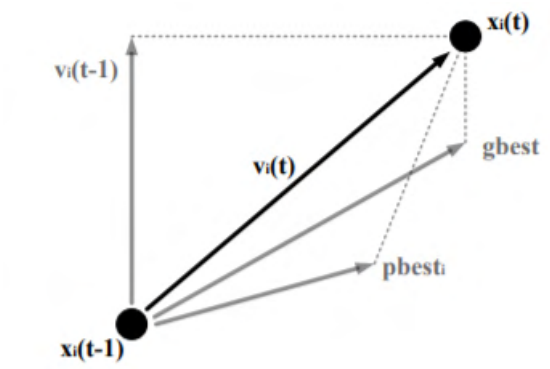

Figure 6: Visualization of the movement of a particle in a two dimensional search space

$$v_i(t) = w(t) * v_i(t-1) + c1 * R1 * (pbest_i - x_i(t-1)) + c2 * R2 * (gbest - x_i(t-1)) \quad (23)$$

As its name suggests, the inertia (w) variable would determine how much the previous velocity would affect the new one, effectively reducing the importance of previous knowledge. This variable also brought a trade-off between the local search ability and the global search ability intrinsic to the equation.

Furthermore, the researchers found that an adaptive inertia, more specifically one that would decrease with time produced promising results. One possible way to describe this change in the inertia value can be described by the equation in (24). Here, w_{max} and w_{min} represent the boundaries the inertia value is confined to and t_{max} represents the maximum running time of the algorithm.

$$w(t) = w_{max} - \frac{w_{max} - w_{min}}{t_{max}} * t \quad (24)$$

To efficiently construct this algorithm, stopping criteria have to be properly defined. As shown in equation (24) defined by the t_{max} variable, one way to define the end of the algorithm is by defining a maximum number of time steps or iterations. If no other stopping criteria is defined, the algorithm will only stop when this maximum number of iterations independently of whether or not the global optimum was reached. It is then important to accurately define the dimension of t_max. A value that is too small will make the algorithm never find the best solution and on the other hand a value that is too big can result in wasted computational effort if the algorithm finds a local optimum from which it cannot escape. Some attempts to reduce this

problem have been suggested such as [31] where a criterion that monitors the improvement of the fitness value is proposed.

Finally, PSO algorithm can be separated in two versions, the global and local version. The global version is the formula presented above, while the local version only differs in one variable. The $gbest$ that represents the global best position of the swarm, is replaced with the best neighbor position. This means that each particle will retain the best position of itself and the best position of the group of particles that are close to it. This allows for a more exploratory behaviour in the algorithm.

Further improvements have been developed throughout the years. In [32], a new version of the PSO algorithm is presented in which a new factor, called the optimum swarm center is added. This optimum swarm center is updated every T iterations by calculating the geometric center from the local optimum positions of all swarm members. This algorithm showed promising results by outperforming the standard PSO in the three benchmark functions tests in the paper, Schwefel, Generalized Rastrigin and Generalized Griewank.

In [7] the MD-PSO algorithm is introduced. This algorithm removes the dimensional restrictions of the normal PSO by adding a dimensional component to each particle. In summary, besides the positional and velocity information contained in each particle, a new dimensional component is added. This means that a particle's position and velocity is always related to some d dimension which can vary between two parameters D_{min} and D_{max}. Consequently, the local and global best are also accompanied by a best local and global dimension size. Furthermore the d component can be updated with a formula similar to the position update in regular PSO. Considering d to be $xd_a(t)$, with a being a specific particle at time t, a $vd_a(t)$ is defined that describes the velocity in the dimensional component. With these variables the researches define the velocity function (25) to update $xd_a(t)$, where $xlbest_a$ is particle a local best d and $dbest$ being the global best d.

$$vd_a(t+1) = \lfloor vd_a(t) + c_1 r_1(t)(xlbest_a(t) - xd_a(t)) + c_2 r_2(dbest(t) - xd_a(t)) \rfloor \qquad (25)$$

To better understand the behaviour of this algorithm, it is useful to think of each possible d value as being a separate search space. Each particle a will then have $D_{max} - D_{min}$ separate positions and velocities, one for each search space. A change in the d component of a particle from $xd1_a$ to $xd2_a$ will then *deactivate* the particle's $xd1_a$ branch and *activate* $xd2_a$, meaning that future position updates will be done on this new branch until d is changed again.

It is worth pointing out that d does not necessarily equal the number of dimensions in its respective search space.

One potential problem that arises from having two separated factors in the velocity update function is that both $pbest_i$ and $gbest$ can be stuck in the same local optimum which would effectively prevent the ith particle to be stuck in that local optimum. Additionally, if $pbest_i$ and $gbest$ are geometrically opposite of each other particles can be stuck oscillating between the two locations. In order to overcome the above shortcomings, a branch of variations of PSO arose. In these versions, the social and cognitive factors of the update function are combined with some formula reducing the equation to:

$$v_i(t) = w(t) * v_i(t-1) + c * r * (e_i - x_i(t-1)) \tag{26}$$

Where e_i is constructed with some combination of the previous factors like in [33] where it becomes a linear combination. Several other combinations exist troughout the literature like in [34] and [35]. Another possible way to create this combined factor e is proposed in [36]. In this paper researchers propose to use genetic algorithm (GA) to create exemplars of this vector.

$$e_i = \frac{c_1 * r_1 * pbest_i + c_2 * r_2 * gbest}{c_1 * r_1 + c2 * r_2} \tag{27}$$

In [37] researchers suggest to use an ensemble with different PSO algorithms. This model, named EPSO, starts with a learning phase where each PSO is given a set number of iterations to optimize particles. After the maximum number of iterations is reached, a success rate is calculated for each algorithm which will then provide the probability that any given model has of being chosen to update each particle. This probability is continuously updated in each time step until the end of the algorithm. The results presented showed the EPSO model performed better then the standard PSO particularly in higher dimensional problems.

2.4 Swarm intelligence and evolutionary algorithms in neural network optimization

Across the literature, evolutionary algorithms have been used to optimize ANNs with three different approaches. A first approach would be to iteratively optimize the ANN's architecture,

another approach followed is to optimize the ANN's weights and the third would be to optimize both weight and architecture. In [38, 7, 39], PSO was used to find an optimal architecture as well as to optimize the weights. The authors in [38] evaluate three different PSO algorithms (basic PSO, SGPSO, and NMPSO) to evolve one hidden layer feed forward neural network. To achieve this, a formula to calculate the maximum number of neurons (MNN) a generated NN could contain was created (28). Here m represents input dimension and n represents the output dimension.

$$MNN = m + n + ((m + n)/2) \tag{28}$$

Researches also defined a set of six activation functions the evolutionary algorithms could work with. Information about topology, activation functions and values of weights and bias was then encoded in a matrix as described in [6]. The matrix's first column T, as shown in Figure 7, contains information about the topology, with each line i containing an integer value from 0 to $2^{MNN} - 1$ that when converted to a binary string represents all the connections from neuron i. For example, assuming that the number 57 is present in row i, by transforming this integer to its binary code "0111001", we can know that neuron i is connected to neurons 2, 3, 4 and 7.

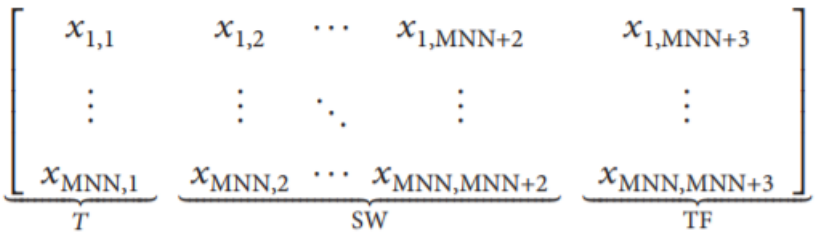

Figure 7: Matrix encoding information about a FeedForward Neural Network (Source: [6])

Columns belonging to the SW section represent values of weights and bias of each possible connection with ranges between [-4,4] and [-2,2] respectively. Finally, the TF section contains the activation function for each neuron with values ranging between [0,5].

Kiranyaz et al [7] use MDPSO to opmize NNs. As explained previously, this model introduces a method for the PSO to loose its dimensional restrictions, enabling it to optimize the ANN's architecture as well as its weights simultaneously. To do this, researchers used the dimensional d component of MDPSO as an index that points to different architectures of different ANNs. For example assuming a vector R_{min} equal to $9, 1, 1, 2$ and a vector R_{max} equal to $9, 8, 4, 2$ we can get a set of NN architectures as shown in Figure 8. Particles would then explore

these architectures by moving in the d dimension as described in the previous section. Results showed that this algorithm was capable of effectively search the configuration set and finding a good configuration of weights and architectures presenting similar results to a BP approach where each individual architecture is extensively searched. Furthermore, the characteristics of the PSO drive the algorithm to favor less complex architectures when the difference in performance is not too great, enabling it to find more compact solutions.

Dim.	Configuration	Dim.	Configuration
1	9×2	22	$9 \times 5 \times 2 \times 2$
2	$9 \times 1 \times 2$	23	$9 \times 6 \times 2 \times 2$
3	$9 \times 2 \times 2$	24	$9 \times 7 \times 2 \times 2$
4	$9 \times 3 \times 2$	25	$9 \times 8 \times 2 \times 2$
5	$9 \times 4 \times 2$	26	$9 \times 1 \times 3 \times 2$
6	$9 \times 5 \times 2$	27	$9 \times 2 \times 3 \times 2$
7	$9 \times 6 \times 2$	28	$9 \times 3 \times 3 \times 2$
8	$9 \times 7 \times 2$	29	$9 \times 4 \times 3 \times 2$
9	$9 \times 8 \times 2$	30	$9 \times 5 \times 3 \times 2$
10	$9 \times 1 \times 1 \times 2$	31	$9 \times 6 \times 3 \times 2$
11	$9 \times 2 \times 1 \times 2$	32	$9 \times 7 \times 3 \times 2$
12	$9 \times 3 \times 1 \times 2$	33	$9 \times 8 \times 3 \times 2$
13	$9 \times 4 \times 1 \times 2$	34	$9 \times 1 \times 4 \times 2$
14	$9 \times 5 \times 1 \times 2$	35	$9 \times 2 \times 4 \times 2$
15	$9 \times 6 \times 1 \times 2$	36	$9 \times 3 \times 4 \times 2$
16	$9 \times 7 \times 1 \times 2$	37	$9 \times 4 \times 4 \times 2$
17	$9 \times 8 \times 1 \times 2$	38	$9 \times 5 \times 4 \times 2$
18	$9 \times 1 \times 2 \times 2$	39	$9 \times 6 \times 4 \times 2$
19	$9 \times 2 \times 2 \times 2$	40	$9 \times 7 \times 4 \times 2$
20	$9 \times 3 \times 2 \times 2$	41	$9 \times 8 \times 4 \times 2$
21	$9 \times 4 \times 2 \times 2$		

Figure 8: Possible NN configurations (Source: [7])

In [40], Zhang et al. suggests using two nested PSO algorithms to find an optimal architecture and to optimize its weights. Each of these optimization phases, called outer and inner phase respectively, run in an alternate fashion. Considering only ANNs with one hidden layer, the algorithm starts by creating a population P of NP NNs with different number of nodes in the hidden layer. Then, for each parent NN belonging to P, $N-1$ child networks are generated with the same number of neurons but random weights. Researchers then proceed to the inner phase where they run the PSO algorithm on each of these populations to optimize the weights of their individuals. The best network from this stage then replaces its parent network in population P. Afterwards comes the outer phase where another PSO algorithm is used to optimize the number of hidden neurons. If for any given architecture, neurons need to be added, then the next inner phase will only optimize the newly added neurons. On the other hand, if neu-

rons need to be removed, all the remaining neurons are trained by the inner PSO. The results presented showed a good generalization ability of the algorithm and it was able to successfully estimate the quality of jet fuel based on thirteen variables.

Researchers in [39] present another method that evolves populations of entire ANNs with uniformly distributed number of hidden nodes and densities of connections. Wang et al. [41] defined a new enconding strategy that enables individual solutions in the PSO method to encode full CNNs. The IPPSO encoding strategy as it was named describes an ip like string that defines all possible configurations for three types of layers, convolutional, pooling and fully connected. This ip string is able to have a predefined unchanging size which enables it to be used in the PSO. To allow networks to have different sizes, the authors also added a forth layer, the ”Disabled” layer, that cuts some layers from each individual.

In [42] PSO is used ”solely” to optimize ANNs weights. In this paper, the Greedy Iterative Layered Training (GILT) algorithm is proposed. This algorithm separates the optimization of the layers of the neural network, optimizing one layer at a time while freezing the remaining layers. This algorithm ensures the reduction of the number of parameters to optimize which enables the use of PSO as the optimization method. Furthermore, the use of the PSO enables a better exploration of the search space whilst keeping a good exploitation ability. Results were promising with the GILT-PSO algorithm over-performing both normal PSO and BP on two datasets (seeds and breast cancer) and beating BP in another (Wine dataset). In a more practical setting, researchers in [43] used PSO to optimize weights of a 3-layer feed forward neural network. In this paper, air quality data from the Hong Kong Environmental Protection Department (HKEPD) was used to compare SGD optimization approach to a PSO one. Results showed that tasked with the problem of predicting CO2 levels, a PSO based approach proved to achieve better solutions than its gradient descent based counter part. Zhang et al [44] proposed a PSO-BP hybrid algorithm that evolved a one hidden layer feed forward neural network. Using PSO with an adaptive learning rate, the algorithm would explore solutions until a plateau was hit. Afterwards, the BP algorithm would be used with the best global solution and if it found a better solution, this new global best solution would replace the worst performing solution in the swarm and the algorithm would return to its PSO ”mode”. This method achieved good results, outperforming regular adaptive PSO and BP in the IRIS classification problem.

In [45] researchers successfully improved performance of a one hidden layer, feed forward neural network in the iris dataset by using PSO as an optimization strategy, achieving better results when combining PSO with Cuckoo Search (CS).

PSO algorithms have also been used to optimize hyperparameters of CNNs and residual neural networks. Kim et al [46] purposes to use PSO instead of the more commonly used grid search since it provides a more global search.

Genetic algorithms have also been used to perform similar tasks as [47] shows. In this paper, GAs are used in order to evolve the topology of the ANNs with promising results. GAs have also been used to optimize weights of ANNs as shown in [48, 49] and, more recently, in [50]. In this latest article, the Accordion method for encoding weights in a chromosome is presented. This technique encodes the weights in such a way that each gene of the chromosome will contain entire layers, effectively ensuring that the chromosome size is considerably reduced.

2.5 Other approaches to training optimization

It is worth noting that some alternative approaches to reduce the training time and compute effort have been explored in the past. In [13] researchers introduce the Selective-Backdrop algorithm that can be implemented on top of variations of the SGD such as AdaGrad, RMSPror and Adam. This algorithm generates batches using a non-uniform selection criteria designed to require fewer backward pass calculations to reach a given loss. Each example of the batch is selected with a probability P that is a function of its current loss determined by a forward pass of the neural network. Through various experiments, the researchers determined that the probability function should be a monotonically increasing function were a higher loss value results in a higher probability of the example to be selected for the batch. This definition follows the idea that it is more efficient to show the model more examples that it has a higher difficulty interpreting. The results showed impressive values with an average improvement of 2.96 and 18.21% on CIFAR10 and CIFAR100 respectively on four different models including ResNet18 and MobileNetv2, as well as a 1.4 and 1.5 times improvement on the time to reach the final error.

3 PROPOSED APPROACH

During the training of DNNs, it is common to observe a pattern of fast improvement followed by a lengthy period of stagnation. This behaviour, termed ”plateau phenomenon” [51, 52], emerges when models face local optimums from which they have difficulty escaping [53, 54].

In order to attempt to accelerate this escape, we propose a new approach where we select a single layer of the model and improve it using PSO.

3.1 Models

In this thesis we will work with three different models, two of which are custom made and the third is the extensively used ResNet18 model ([8, 55, 56, 57, 58]). The first of the three models was named SimpleNet. Inspired by the LeNet-5 [59] with modifications to support 3 color channels as well as increased neuron numbers, this model consists of a first block of two convolutional - batch normalization pairs with a maximum pooling layer in between, followed by a linear block consisting of 3 linear layers as shown in Figure 9. The choosen activation function was the ReLu. The following model we will work with will be a reduced version of ResNet18

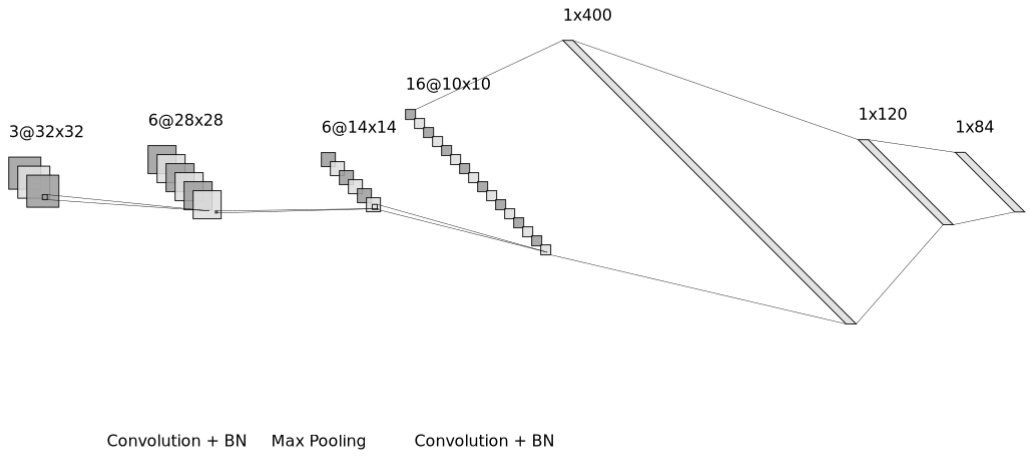

Figure 9: SimpleNet Architecture

which we will call by ReducedResNet. This model is similar to its parent ResNet but with one less convolutional block. Finnaly we will also test the ResNet model with 18 layers (ResNet18), as shown in Figure 10, to allow us to better understand the behaviour PSO algorithms have on models with proven results in the literature.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				

Figure 10: ResNet Architecture (Source: [8])

3.2 Pre-training models

As stated before the first objective of the work is to train the models from scratch until a plateau in performance (i.e. validation loss) is hit. To do this, we will perform a number of experiments to empirically find an iteration size sufficient so that the improvement both in accuracy and loss stagnates. Two gradient descent based algorithms will be tested to perform this optimization, SGD and ADAM. Furthermore, we will also perform tests with three different learning rate levels in order to achieve the best validation loss possible without dedicating too much time tuning the parameters for this stage.

3.3 Layer Selection

After the model is trained we move onto selecting the layer that will be further optimized. We will consider two approaches to achieve this, a pruning based technique and the selection of the linear layer closest to the output.

3.3.1 Pruning selection

Several research papers have focused on reducing neural network size by removing specific nodes, connections, or entire layers. These approaches commonly use a method for weighting the importance of each layer or connection. In [24] the authors make use of batch normalization weights to complete this task, proceeding then to prune the neurons that are less important to

the network. We propose to use the same method to select our target layer, whilst keeping the network's architecture. As explained in the paper, this approach is possibly the most effective way to learn scaling parameters since it uses existing neural network architecture and therefore negates any impact to the network's performance. The implementation of this method in this thesis will follow what is described in [24] which uses L1 normalization as a sparsity-induced penalty (a type of penalty that tends to create several zero valued weights) on scalars and sub-gradient descent (an optimization algorithm for optimizing nondifferentiable convex functions) as the optimization method for this L1 penalty term, as shown in (29).

$$g_t = g_{t-1} + \lambda * |w_{t-1}| \tag{29}$$

Here w and g refer to each batch normalization layer's weights and gradients respectively, whilst the scale sparse rate (λ) serves as a weight between 0 and 1 that controls the importance of the scaling factors (the batch normalization weights, in this case). The final scaling factors of each layer will then be averaged in order to find the most important layers.

3.3.2 Classifier layer

Since classifier layers in our models are not followed by batch normalization layers they get excluded from the pool of possible selected layers by the previous approach. Furthermore, classifier layer's weights directly affect the output and therefore any change to them should result in a considerable change to the output. For these reasons, the layer selected from the previously mentioned pruning approach will be compared with the final classifier layer of each model in each experiment, allowing us to better understand the quality of the pruning selection method.

3.4 Layer Optimization

After selecting the layer to be optimized we will study the behaviour of the PSO with inertia algorithm and compare it with gradient descent optimization on the same layer and on the whole model. This comparison will allow us to understand whether or not these have a better chance of thriving in this highly dimensional search space. Each algorithm will be given 15 iterations to improve upon the previous best validation loss. This value may seem low but the number of

individuals per population pushes us to limit iterations in order to seek positive results with a lower or similar computational budget. In order to be able to use PSO to optimize the target layer weights and bias, we need to select an enconding to represent these weights and bias as individuals in the swarm. There are three ways of encoding this information as described in [44]. Matrix encoding creates particles as matrices, keeping weights in the same form that is used in ANNs. Vector encoding flattens the weights matrices, creating a single vector that contains all weights and bias. Finally, binary enconding creates a binary string for all weights, which, when the number of weights is large, becomes very complex to encode and decode. When dealing with ANNs, the first two encoding strategies are often used. In this thesis the vector encoding strategy was implemented. Each optimization approach will start with an initial solution that will correspond to the last state of the weights and bias of the layer selected in the first pre-training seed. Even though we realize the optimal approach would be to have each experiments run for $m * n$ seeds with m corresponding to the number of seeds in the pre-training phase and n to the number of seeds in the single-layer phase, given the time and compute resource constraints we felt our approach enables us to explore more parameters.

3.4.1 Particle Swarm Optimization

To study the best parameters for the PSO in this problem, we will first, define our baseline parameters for the PSO. Afterwards, each parameter will be tested with different values and its impact on results will be studied. Finally, the final parameter values will be defined and results presented. In our experiments we will be studying inertia, positional and velocity clamping, cognitive and social factors and population size. Each of these parameters can play an important role in the behaviour of the algorithm.

According to the literature, inertia provides a key role in balancing exploration and exploitation. Its creators, Shi and Eberhart in [30], state that a large inertia value is conducive to a global search whilst a small one facilitates local search. Besides the original version where the inertia value remains constant, several improvements have been suggested that dynamically adapt inertia throughout the algorithm's running time. When it comes to choosing a value as our baseline, we decided to select a value of 1.2. This high inertia value seems suited to our use case since it will increase our exploratory ability, which should allow us to leave the plateau the model starts in.

The cognitive and social factors of the PSO algorithm will play a role in balancing the attraction to the particles own best position and the global best position of the swarm. As a

baseline value for these parameters, we choose the value 2 for both since on average it makes the weights on the social and cognitive factors be 1, according to [18]. This initial equilibrium between the two parameters will be a good starting point to understand the model's behaviour and how it changes with variations in the balance of these factors.

Initially, we will not bound the particles' position or velocity, leaving both of these parameters to run free. During our experiments, however, we will test the effects that both positional and velocity bounding have on the behaviour and performance of the algorithm. For the positional bounding study, we will use a factor f that will scale an l variable, defined as the maximum absolute value of all dimensions of the initial particle's position, to increase the particles's search space. The boundaries will then become $[-f * l, f * l]$. Particles that leave the defined bounds will be randomized into the bounded space.

Similarly, to control velocity, the variable vb is defined as the maximum absolute gradient of the initial solution. Particles will then be limited to the $[-vb, vb]$ interval. Much like with the bounding experiment, a factor f will then be added to scale this variable.

Unlike positional bounding however, particles that accelerate past the maximum established velocity do not get completely randomized. Instead, much like is implemented in [28], particles maintain their original direction whilst the magnitude of their velocity gets shortened by the result of the $f * vb$ limit over the maximum absolute value of the velocity vector. Finally, in our baseline parameter set the population will start with 100 individuals which seems to be a widely used baseline value and sigma value will start at 0.2.

3.4.2 Single layer gradient based optimization

As we stated above, we will also experiment with optimizing the selected layer with a gradient based algorithm. The optimization method chosen and its learning rate will be the same used in the pre-training phase of the given model. For example, if the SGD optimizer with a learning rate of 0.01 is the best performer for ResNet18, then that is the configuration that will be used to optimize the chosen layer. It would be interesting to explore different parameter values or optimizers but given the resources available we felt it was best to leave this exploration for future works.

3.4.3 Performance Evaluation

We will evaluate models and parameter sets through the use of the Cross-Entropy loss function (30). We will also present results for accuracy although they wont be relevant towards the decisions we will make.

$$L = -\sum_{t=i}^{n} t_i \log(p_i) \tag{30}$$

3.4.4 Implementation and tools used

For this dissertation, all code was written in *Python* and the *pytorch* module was used for the implementation of the DNNs. This module was selected over *tensorflow* due to the exposure that the module gives to components of the networks such as the calculated gradients throughout the training process. The ability to attach custom components to the layers with the concept of *hooks* was also instrumental in implementing the pruning layer selection process.

4 EXPERIMENTAL SETUP

4.1 Datasets

In order to better evaluate the performance of the models, we selected four datasets with varying degrees of complexity.

- MNIST dataset [60] which contains gray scale images with a shape of 28x28x1 of handwritten digits with 60000 training images and a test set of 10000.
- FashionMnist dataset [61] which much like MNIST contains gray scale images with a shape of 28x28x1 of different clothes with 60000 training images and a test set of 10000.
- CIFAR10 dataset [62] with 10 classes containing 6000 32x32x3 images of objects and animals. The training set has a size of 50000 and the test set 10000
- CIFAR100 [62], which contains 100 classes with 600 32x32x3 images each. This dataset is essentially an upscale of CIFAR10, containing 10 times more classes.

These datasets were choosen in order to provide our models with different levels of input complexity. The simpler Mnist dataset contains more clearly distinguishable numbers. The FashionMnist dataset introduces additional complexity in the images presented by having images with a less defined shape, CIFAR10 introduces color channels and CIFAR100 provides the models with more variability in the input. In the pre-processing of these images we have resized both MNIST and FashionMnist to a size of 32 by 32 to keep layers with the same dimensions across datasets. Furthermore, images have been vertically flipped with a probability of 0.2 and each pixel was normalized with a mean and standard deviation of 0.5.

4.2 Pre-training models

After running some preliminary experiments we decided to allow every model 20 iterations (read epochs) for the pre-training phase, to learn using three learning rates (0.01, 0.001 and 0.0001) and two gradient based optimization methods, SGD and Adam.

4.3 Layer Optimization

As described previously, we defined the following baseline parameter set:

- Population: 100
- Inertia: 1.2
- Cognitive Factor: 2
- Social Factor: 2
- Sigma: 0.2

From this baseline we will then attempt to improve results by varying each parameter and analysing results. In total we will conduct 6 experiments:

- Gradient based optimization experiment: Compares baseline results with optimizing the target layer with the same optimization method used in pre-training for the given model
- Inertia experiment: Experiments with lower inertia values of 0.5 and 0.9.

- Particle positional bounding: Explores how the algorithm behaves when its particles have a more restricted search space
- Particle velocity bounding: Has the same intent has the positional bounding experiment but with velocity
- Population experiment: Compares baseline population value of 100 with a lower (50) and higher (150) values.
- Cognitive and Social factors experiment: Experiments with a 3 to 1 and 1 to 3 ratio between cognitive and social factors.

To characterise the behaviour of each parameter set throughout the experiments we will be capturing several metrics. From the best individual we will capture the training and validation loss of each seed, calculating its average and standard deviation. From the population we will also monitor the average and standard deviation of training loss as well as the step size defined as the absolute median of the velocity vectors of the individuals. Further more we will also monitor information about whether the best solution changed and whether the particles start going out of positional and velocity bounds.

In summary, per iteration we monitor the following metrics:

- Best individual training loss
- Best individual validation loss
- Averaged population training loss
- Standard deviation of the population training loss
- Average population step size
- Standard deviation of population step size
- Total number of dimensions in the individuals position that go out of bounds
- Total number of dimensions in the individuals velocity that go out of bounds
- Whether the best solution changed (1 or 0)

From each of these data points we then calculate the average and standard deviation per seed.

5 RESULTS

5.1 Pre-training models

As previously stated, we started this phase by allowing every model 20 iterations to learn using three learning rates (0.01, 0.001 and 0.0001). Every model was also tested with two different optimization methods: Adam and SGD. After running the experiment with 5 different seeds the result were as shown in Table 1.

Dataset	Best Model	Lr	Optimizer	Best Loss	Best Acc
MNIST	ResNet18	0,0001	Adam	0,03781	99%
FashionMNIST	ResNet18	0,001	Adam	0,25633	91%
CIFAR10	ReducedResNet	0,001	Adam	0,76917	74%
CIFAR100	ReducedResNet	0,001	Adam	2,05603	46%

Table 1: Best average validation loss and accuracy per dataset

Optimizer	Lr	SimpleNet	ReducedResNet	ResNet18
Adam	0,0001	1,4262	1,1496	1,4490
Adam	0,001	1,6021	1,1552	**1,1582**
Adam	0,01	**1,3693**	**1,0281**	1,3338
SGD	0,0001	2,6553	2,4279	1,9467
SGD	0,001	2,0035	1,8285	1,4897
SGD	0,01	1,3774	1,2376	1,3086

Table 2: Best overall average validation loss

As expected, both accuracy and loss results show that, as datasets become increasingly complex, accuracy decreases. However, unexpectedly, the ResNet18 model was outperformed by the simpler ReducedResNet in the CIFAR100 dataset. This result is even more pronounced when using the learning rate of 0.001. When we observe the evolution of loss per iteration in both CIFAR10 and CIFAR100 datasets, we can see that the ResNet's performance decreases when it reaches iteration 8.

Given the results presented, we decided to move forward to the next phase with the Adam optimized version of the models and with the 0.01 and 0.001 learning rates. These selections were made based on the average loss of each model, optimizer and learning rate combination across all datasets, as seen in Table 2. For the SimpleNet and ReducedResNet we will

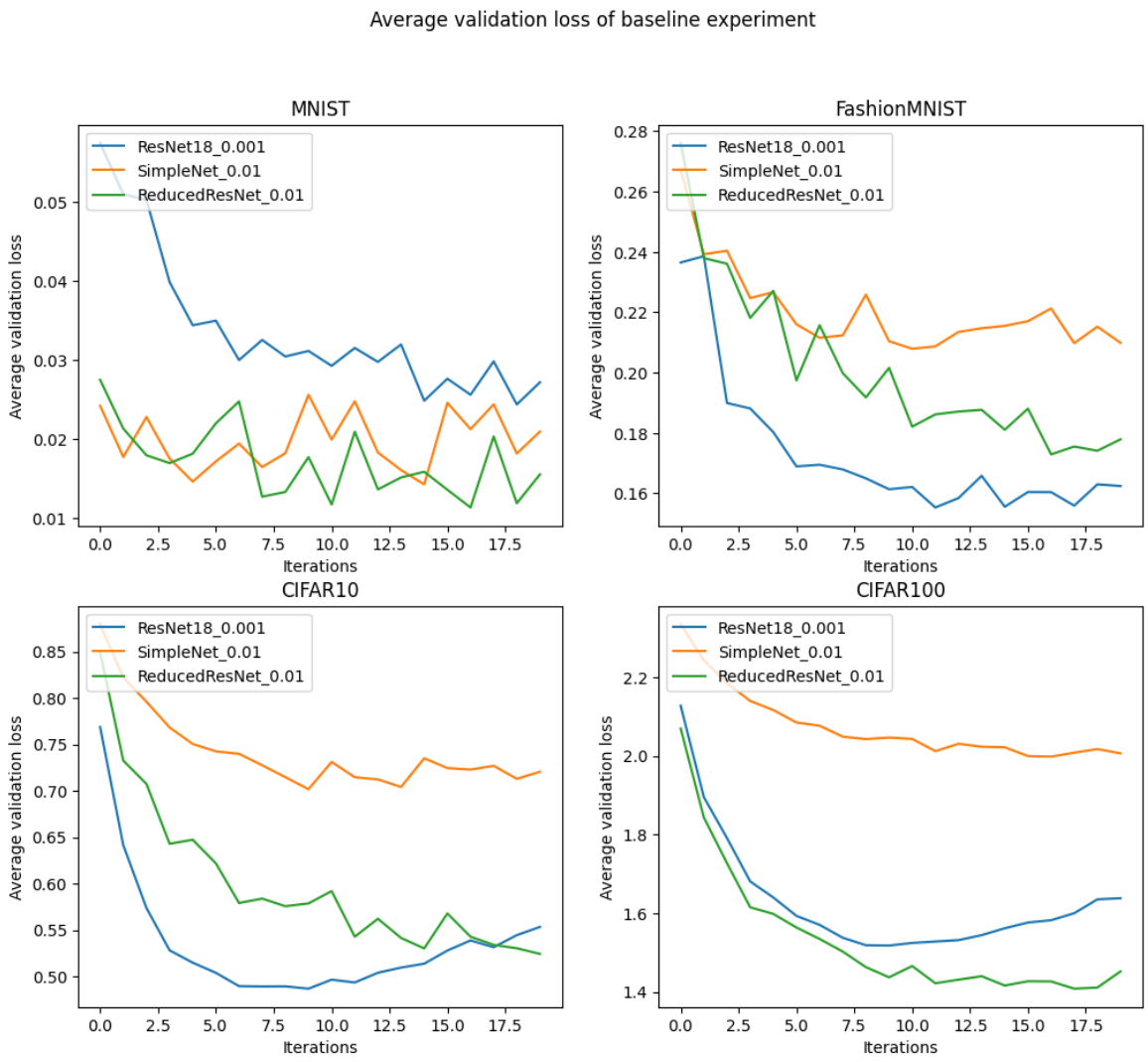

Figure 11: Average validation loss in pre-training with best parameters

be using the model optimized with 0.01 whilst for the remaining model ResNet18 the learning rate of 0.001 was selected. In summary the models that move on to the next phase are:

- SimpleNet with Adam optimizer and 0.01 learning rate
- ReducedResNet with Adam optimizer and 0.01 learning rate
- ResNet18 with Adam optimizer and 0.001 learning rate

Since all models achieve an almost perfect accuracy in the MNIST dataset, we will not perform experiments with this dataset for the next steps since there is little to no improvements that can be made.

5.2 Layer selection

After having our models trained and datasets selected, we move on to selecting the layers we will be targeting for optimization. Besides the final classifier layer, chosen in every model, our prune selection method choose the following layers per model and dataset:

Dataset	Model	Layer	Dimensions
FashionMNIST, CIFAR10, CIFAR100	SimpleNet	.conv2	96
CIFAR100	ResNet18	.conv1	192
FashionMNIST, CIFAR10	SimpleNet	fc3	840
FashionMNIST, CIFAR10	ReducedResNet	fc	1280
FashionMNIST, CIFAR10	ResNet18	fc	5120
CIFAR100	SimpleNet	fc3	8400
CIFAR100	ReducedResNet	fc	12800
FashionMNIST, CIFAR10, CIFAR100	ReducedResNet	.layer2.0.conv2	16384
CIFAR100	ResNet18	fc	51200
FashionMNIST, CIFAR10	ResNet18	.layer4.0.downsample.conv	131072

Table 3: Pruning method layer selection

In summary the layers chosen were the same on each dataset per model with the ResNet18 model in the CIFAR100 being the exception to the rule.

5.3 Layer Optimization

5.3.1 Baseline PSO

We started by experimenting with optimizing the selected layer with a PSO algorithm with a baseline parameter set of 100 individuals, a sigma value of 0.2, inertia of 1.2 and cognitive and social factors both at 2. With these parameters however, the model failed to improve in any way and in fact, a degradation in performance was observed across environments as it is shown in Figure 12. In general, validation loss either spikes to a higher level in the first 5 iterations or it stagnates.

We hypothesize that the main driving force to this behaviour is that, unlike with gradient based optimization methods where the best validation loss belongs always to the only existent solution, with PSO, the best validation loss can jump between several solutions, making its value more variable. Furthermore, since the global best solution changes with training loss, the particle that contains the best validation loss, bp, may stop exerting its influence on other individuals if its training loss is not the best in the population. At the same time if this situation occurs, particle bp will also be pulled away from its positions by particles with worse validation loss. This behaviour, when in an environment with a much higher number of solutions with worse validation loss than the value achieved by the pre-trained model would lead to the degradation of performance we observe. In order to perform a detailed explanation we will now analyze each model's performance.

FashionMnist

Due to the simplicity of this dataset, the starting state of the models was already at a fairly low validation loss. The SimpleNet model was at 0.357, ReducedResNet at 0.315 and ResNet18 at 0.282, with accuracies of 89, 90 and 91% respectively.

Given these conditions, in the SimpleNet's case, with the convolutional (conv2) layer targeted, the performance of the model degraded for 6 iterations before plateauing at approximately 0.407. In the linear (fc3) layer case, the model performance remained around 0.349 throughout the training cycle. It is interesting to note that, despite the absence of degradation of the linear layer, average training loss of the populations showed growing degradation in both cases. Additionally, looking at the step size of each layer, represented by the absolute median of the velocities at each iteration, an exponential growth is also observed (Figure 13).

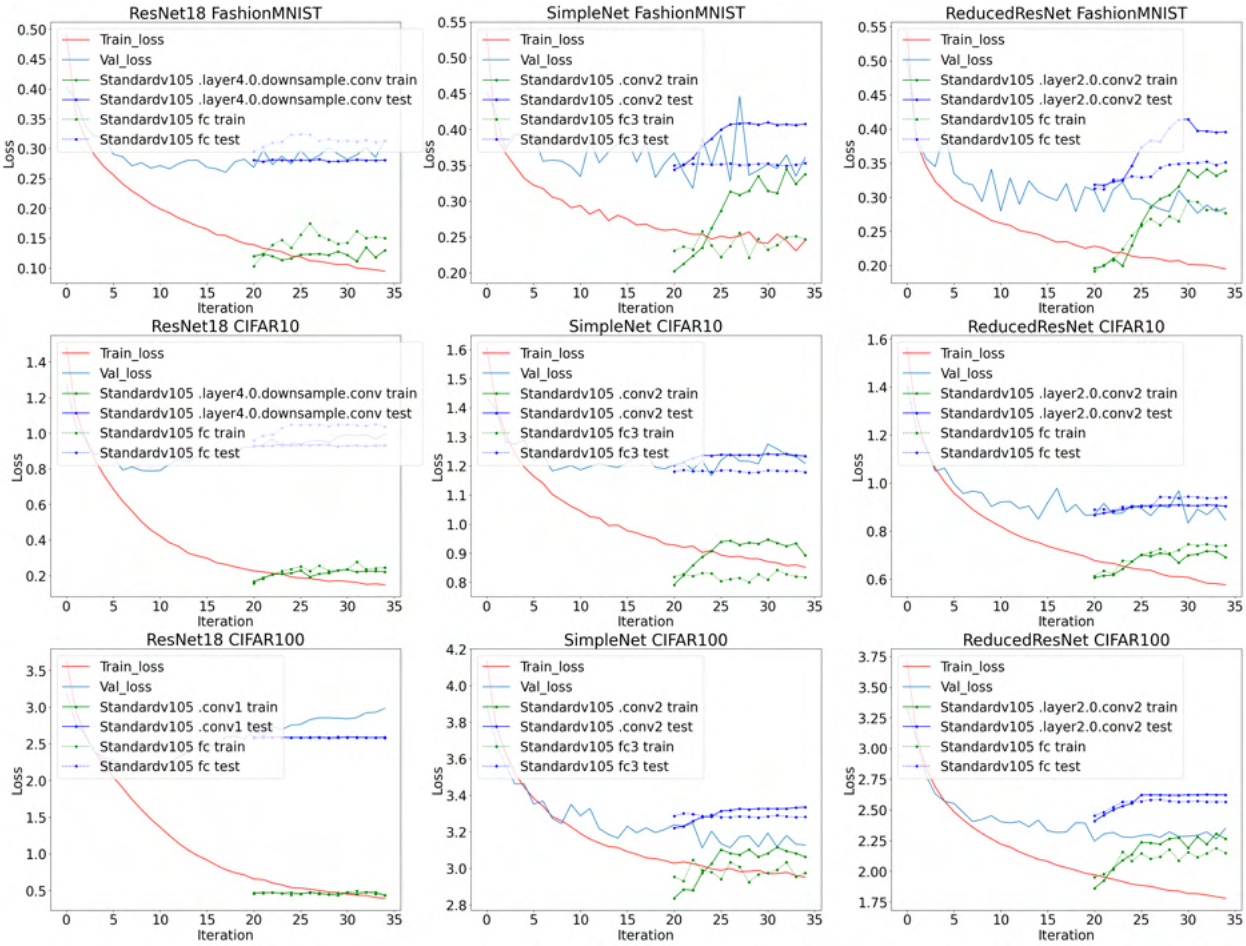

Figure 12: Baseline train and validation loss

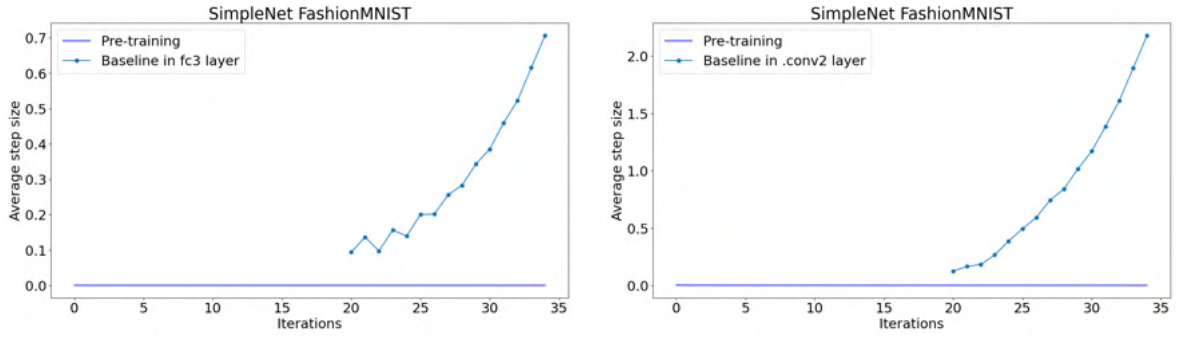

Figure 13: Step size in SimpleNet model with the FashionMNIST dataset

The only significant difference observed in both layers seems to be the evolution of the best solution. In the convolutional layer during the first five iterations the best individual remained the same across seeds before the best solution started to be replaced as the best validation loss of the population started to stabilize as shown in Figure 14.

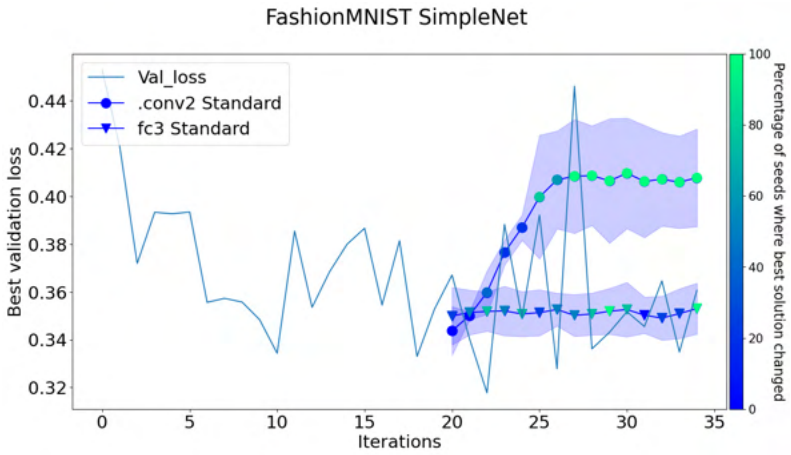

Figure 14: Average validation loss of SimpleNet in FashionMnist

With the ReducedResNet model, validation loss started on 0.312 and 0.318 for the linear and convolutional layer respectively. The first layer then proceeded to degrade the validation loss until 0.351. The convolutional layer slowly degraded the model's performance, ending with a validation loss of 0.395. Looking at the changes in the best solution we can once again see that the best validation loss stabilizes as the best solution starts to be swapped more often (Figure 15).

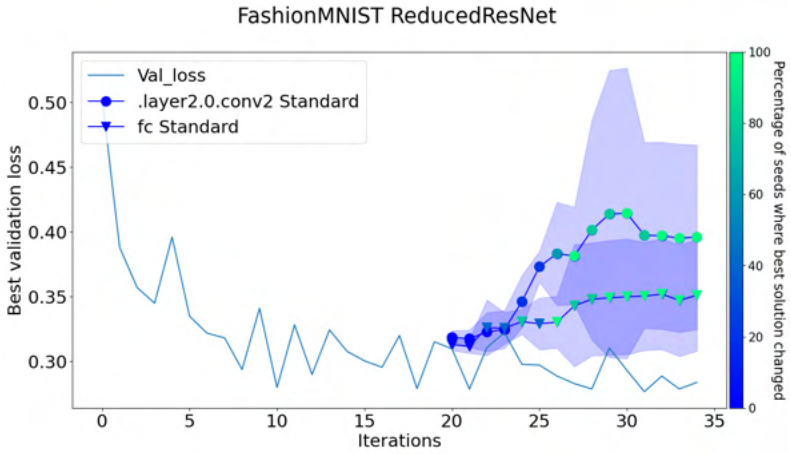

Figure 15: Average validation loss of ReducedResNet in FashionMnist

Finally, in the ResNet18 model we find that unlike in the other models, the linear layer performs worst then the convolutional counterpart, with the first degrading performance throughout iterations whilst the second managing to maintain validation loss. Furthermore, we find that the standard deviation between seeds is much higher in the linear layer. Looking at the Figure 16 we can see a somewhat similar behaviour to what was observed in the SimpleNet

model but with reversed layers. The linear layer degrades whilst the best validation loss in the convolutional layer barely changes across seeds and stays stable.

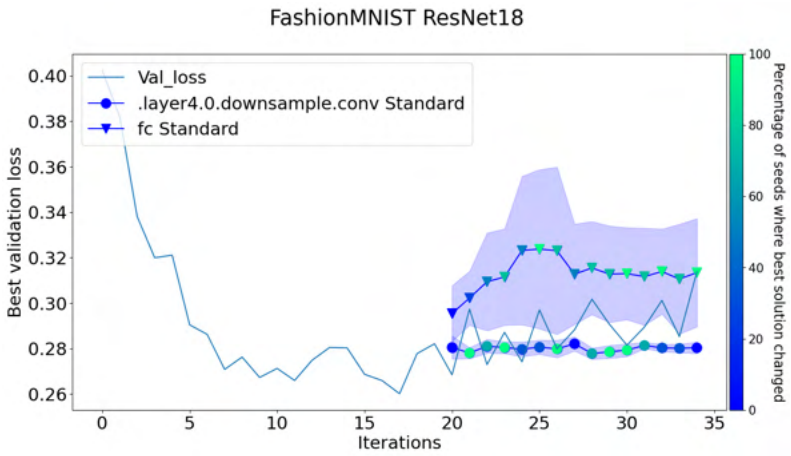

Figure 16: Average validation loss of ResNet18 in FashionMnist

CIFAR10

The CIFAR10 dataset increases in complexity by having photos in realistic settings as well introducing color in the images. The starting performance of the models was on average at 1,06, 0.877, 0.791 and accuracy of 63, 74 and 74% for SimpleNet, ReducedResNet and ResNet18 respectively. In the convolutional layer of the SimpleNet model, the performance validation worsens to 1.20 in the first iteration and continues to degrade until it plateaus at around 1.23. The linear layer performs better and stays around 1.18. Similarly to what was observed in the FashionMnist dataset, the convolutional layer in this model degrades, with the changing of the best solution following a similar behaviour as well. However the linear layer on this dataset showed a much lower standard deviation of the best validation loss (Figure 17). Finally, the step size behaved in similar fashion to what has been previously observed.

Looking at the ReducedResNet model, both the linear and convolutional layer start with similar validation loss, 0.88 and 0.87 respectively, and degrade to 0.94 and 0.90 (Figure 18). In this model we again find that as the training progresses, the best solution becomes more and more unstable and starts to be replaced at every seed in every iteration.

Lastly, much like in the previous dataset, the linear layer in ResNet18 degrades whilst the convolutional remains stable. More specifically, the linear layer jumps to 0.96 and proceeds to degrade to 1.03 whilst the convolutional layer remains around 0.93 throughout the 15 iterations. (Figure 19).

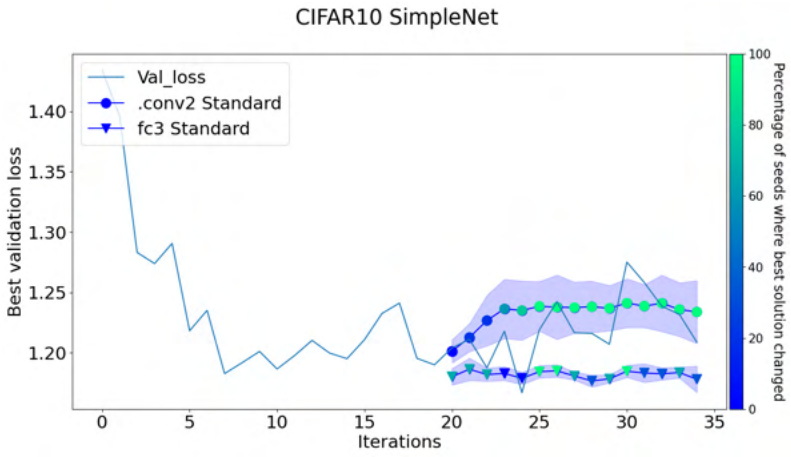

Figure 17: Average validation loss of SimpleNet in CIFAR10

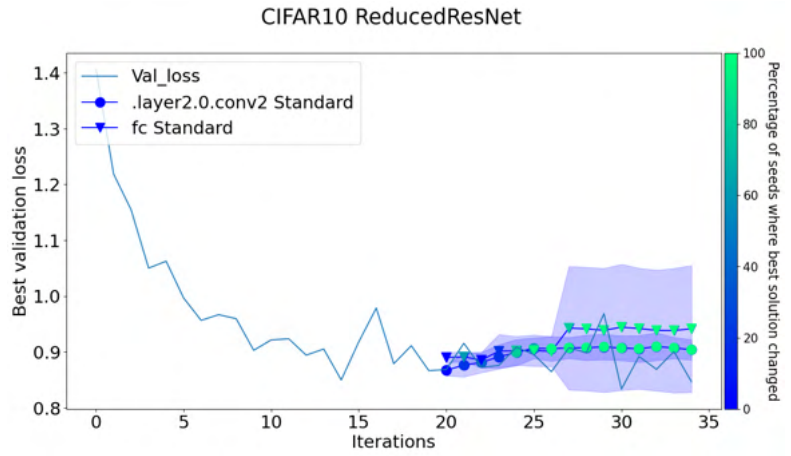

Figure 18: Average validation loss of ReducedResNet in CIFAR10

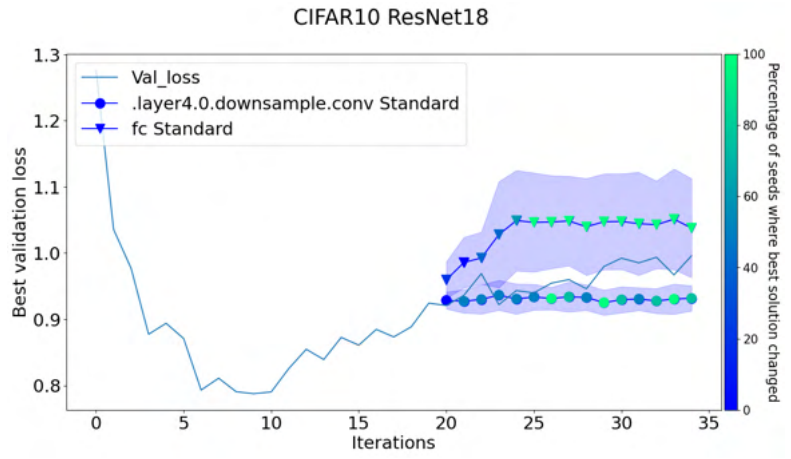

Figure 19: Average validation loss of ResNet18 in CIFAR10

CIFAR100

Finally, in the CIFAR100 dataset, the models are confronted with a number of classes ten times higher. As previously stated, model performance in this dataset after the pre-training phase was at 2.802, 2.536, 2.42 and 30, 34, 39% for SimpleNet, ReducedResNet and ResNet18 respectively.

The linear layer of the SimpleNet model starts with a higher validation loss of about 3.285 and remained around this value throughout training. On the other hand the convolutional layer started lower at 3.21 but degraded to 3.33 (Figure 22). On all metrics the behaviour of the model in this environment was similar to what was observed in the previous datasets, with an exponential increase in step size and a growing training loss of the population.

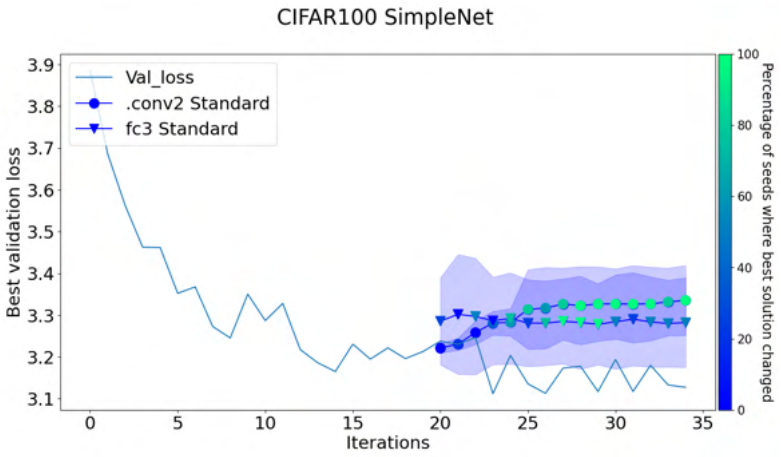

Figure 20: Average validation loss of SimpleNet in CIFAR100

On the ReducedResNet model the linear layer started at 2.45 and degraded to 2.56 and the convolutional layer started at 2.40 and degraded to 2.62. In the ResNet18 model, both linear and convolutional layers performed very similarly maintaining an validation loss close to 2.59 and 2.58 for the linear and convolutional layer respectively. Furthermore both layers had very low standard deviation per seed and similar standard deviation per individual on each population. On both these models we clearly see a decrease in standard deviation of the validation loss

Overall, results were unsatisfactory. In the SimpleNet and ResNet18 models in particular, differences between the two layers were very stark with one layer remaining in a stable state throughout iterations while the other degrades. The layer that degrades, whether it is convolutional or the linear layer seems to depend on the model. The reason for this behaviour, we believe, may be related to two factors. The first one has to do with the layer selection. In

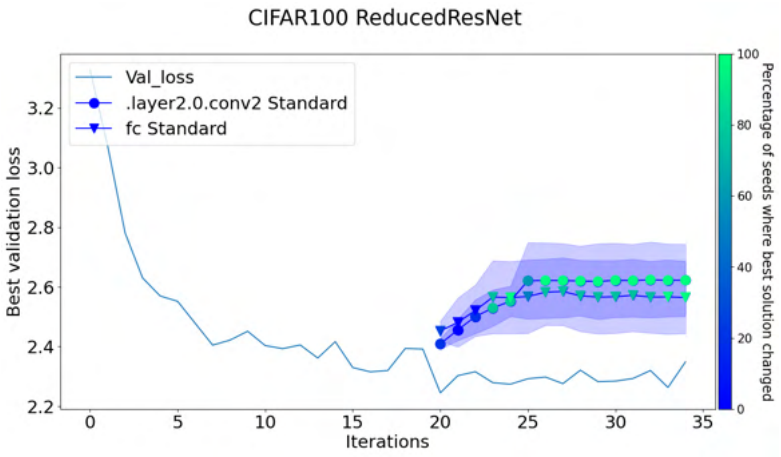

Figure 21: Average validation loss of ReducedResNet in CIFAR100

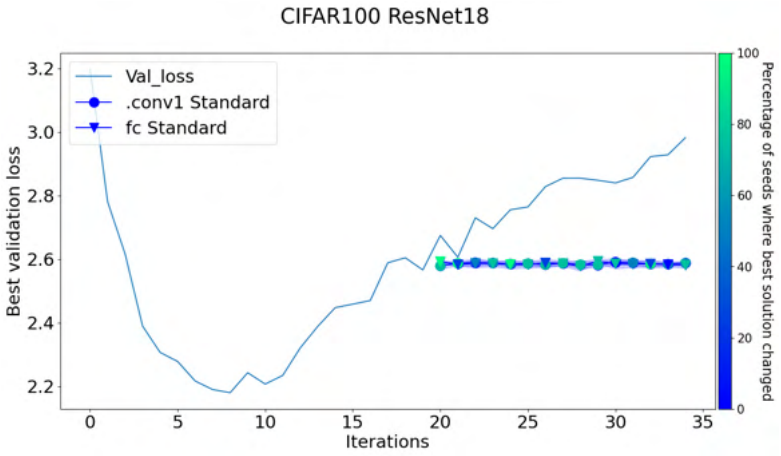

Figure 22: Average validation loss of ResNet18 in CIFAR100

each model, the PSO's behaviour will be more transparent, when applied to a layer with higher importance, which in this experiment seems to be the case for the second convolutional layer in the SimpleNet and the linear layer of the ResNet18. As for the ReducedResNet model the chosen convolutional and linear layer choosen seem to have the same impact on model performance with exception to the FashionMNIST case where the linear layer is clearly much more impactful. The second factor is also related to the layer selected but with respect to the step size. Each layer will have a step size with varying magnitudes, which means that the difference of the step size in the pre-training and the training phase may vary a lot between layers in the same model. As is demonstrated in Figure 23.

When looking at the changes in the best solution we can see a clear increase in activity in later iterations, which seems to be partially related to the exponential growth of step size as shown in Figure 24. In this image we can see a comparison between a normalized average

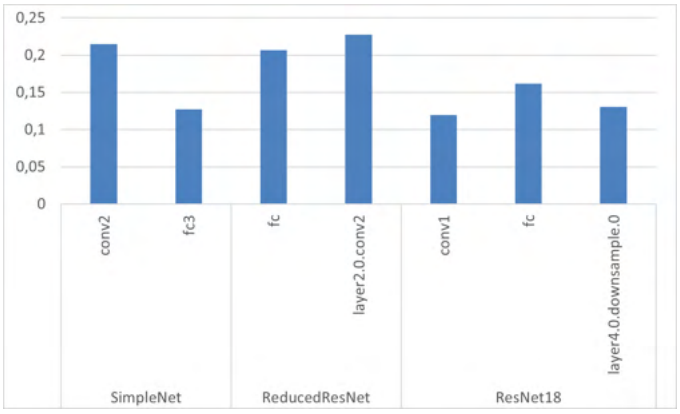

Figure 23: Relation between step size and fraction of seeds that changed best solution per iteration

step size against the averaged fraction of seeds where the best solution changes per iteration aggregated across all datasets, models and layers.

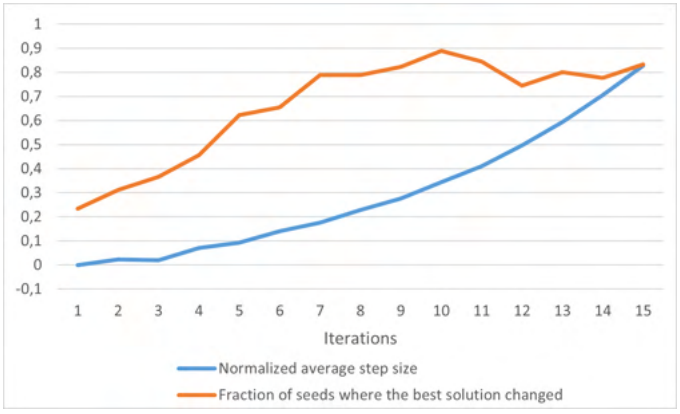

Figure 24: Relation between step size and fraction of seeds that changed best solution per iteration

Along with this increase in step size we also observe an exponential increase in the mean training loss of the population, particularly high in the linear layer of the SimpleNet as seen in Figure 25.

Finnaly, after analyzing how much impact changing the best solution has on validation loss, we conclude that, changing the best solution, on average, guarantees a smaller degradation of performance. In the FashionMnist dataset validation loss increases on average 0.032682 when there is no change to the best solution, compared with a insignificant increase of 0.0007 when there is. On the other datasets, Cifar10 and Cifar100, we find that changing the best solution guarantees, on average, an even smaller degradation or even an improvement in validation

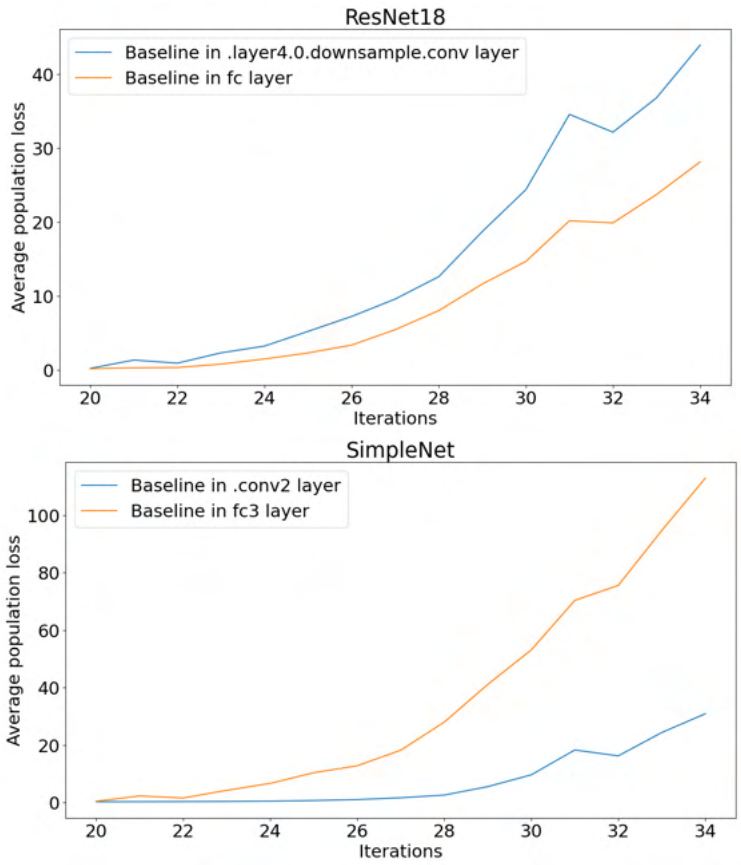

Figure 25: Population mean training loss

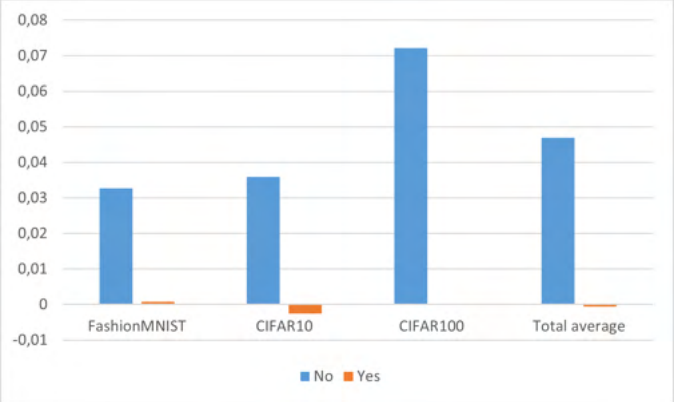

Figure 26: Change in loss with and without best solution change

loss, as seen in Figure 26 Overall, the total average across datasets came to a degradation of 0.046921 when there was no change to the best solution against a slight improvement of 0.0006 when there was. It is worth noting, however, that after the first iterations, usually the PSO keeps changing the best solution, meaning that we have more data available for situations when the best solution changes. Despite not being conclusive, these results seem to indicate that our hypothesis that the main driving factor of degradation was the change of the best solution to solutions with much worse validation loss was incorrect.

5.3.2 Single layer gradient optimization

In order to compare different optimization strategies, as well as to better understand the impact that the choice of the layer has on results, we studied the performance of optimizing the chosen layer with the gradient based optimizer selected in the pre-training phase for each model, the results of which we present in the following section.

FashionMnist

In this dataset the Adam optimization of a single layer worked surprisingly well, particularly with the convolutional layers. When applied to the SimpleNet model the gradient based optimization managed to improve model performance by 4% to 0.305 validation loss in the convolutional layer. In the ReducedResNet model the convolutional layer had similar performance gains with an improvement of 9.9% to a loss of 0.249 and the linear layer showed better performance by improving its validation fitness to 0.264, a percentage improvement of 4.3%. Lastly,

ResNet18 showed a 1.8% improvement with the convolutional layer as shown in Table 4. It is worth noting that the improvements observed were mostly achieved in the first iteration, with small gains in performance observed throughout the rest of the iterations,the exception being the ResNet18 model which showed a continued improvement throughout the iterations (Figure 27).

Model	Layer	Experiment	Best Validation	Diff to pre-training	Diff to pre-training (%)
ReducedResNet	layer2.0.conv2	Adam optim	0,249	-0,027	-9,9
ResNet18	layer4.0.downsample.0	Adam optim	0,255	-0,005	-1,9
ReducedResNet	fc	Adam optim	0,265	-0,012	-4,3
ResNet18	fc	Adam optim	0,267	0,007	2,6
ResNet18	layer4.0.downsample.0	Standard	0,278	0,018	6,8
ResNet18	fc	Standard	0,295	0,035	13,6
SimpleNet	conv2	Adam optim	0,305	-0,013	-4,0
ReducedResNet	fc	Standard	0,312	0,035	12,7
ReducedResNet	layer2.0.conv2	Standard	0,317	0,041	14,8
SimpleNet	fc3	Adam optim	0,334	0,016	5,0
SimpleNet	conv2	Standard	0,344	0,026	8,2
SimpleNet	fc3	Standard	0,349	0,031	9,8

Table 4: Single Layer Optimization results

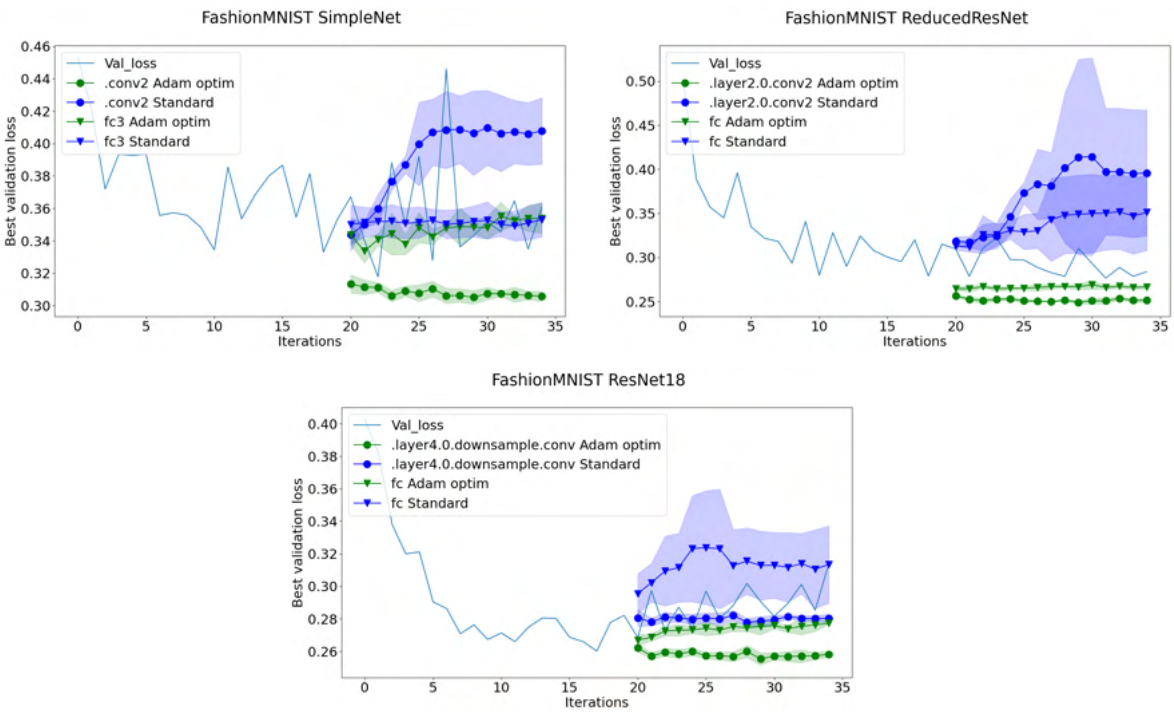

Figure 27: Best validation loss of single layer gradient optimization

CIFAR10

On the CIFAR10 dataset, improvements were only observed in the SimpleNet and ReducedResNet model. In the first, only the convolutional layer was able to improve performance by 1.5% whilst with the linear layer performance degraded by 1.77%, underperforming the baseline result. On the ReducedResNet results were better with a 6.28 and 2,84% improvement in the convolutional and linear layer respectively. Finally on the ResNet18 model both layers were not able to improve performance.

CIFAR100

Of the three datasets, the CIFAR100 dataset showed the worst results for this experiment, with performance improving only in the ReducedResNet model. Here the convolutional layer improved model performance by 5.22% to 2.123 loss and the linear layer showed a 2.71% improvement.

Overall, using gradient based optimization on a single layer seems to yield reason-

ably good results. Improvements in validation loss were achieved in several situations which also allows us to confirm that improvement of the model with a single layer being optimized is possible. We also found that in situations where the model was in degradation, such as the ResNet18 in the CIFAR100 dataset, using a gradient based algorithm on a single layer reversed this trajectory when applied to the correct layer. In our experiment, the linear layer showed a similar behaviour to what the full model optimization had shown but when optimizing the convolutional layer the model stabilized performance, as shown in Figure 28.

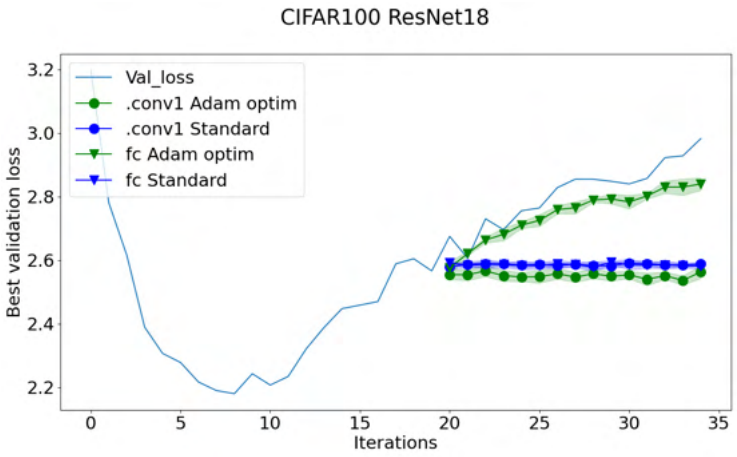

Figure 28: Adam optimizer on ResNet18 in Cifar100

The results of this experiment also showed that the pruning method to select an optimization layer seems to have led to better results then the classifier layers as shown in Figure 29.

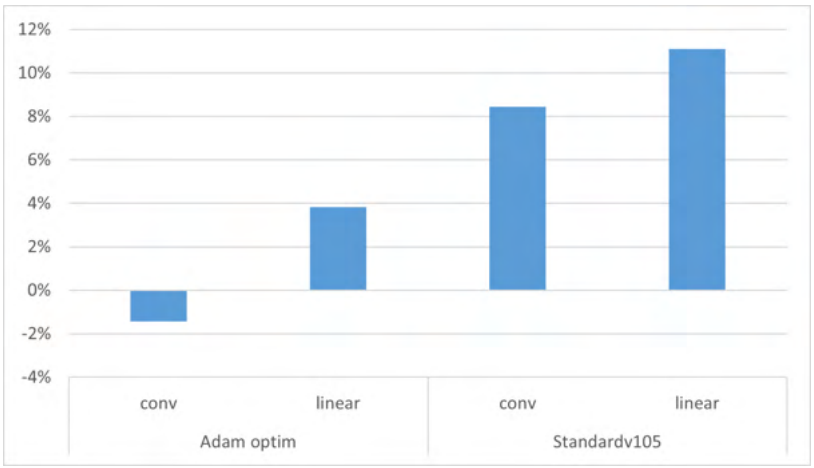

Figure 29: Average % improvement over best validation in pre-training

5.3.3 Inertia

As described previously, inertia, much like momentum in gradient based algorithms determines how much of the previous direction v_{t-1} gets carried to the next iteration v_t. A higher value will then prevent *tight turns* and makes the algorithm less random. In this experiment we tested the PSO with two lower values of inertia 0.5 and 0.9 and, as expected in both cases, step size increased in a much more controlled fashion as shown in Figure 30.

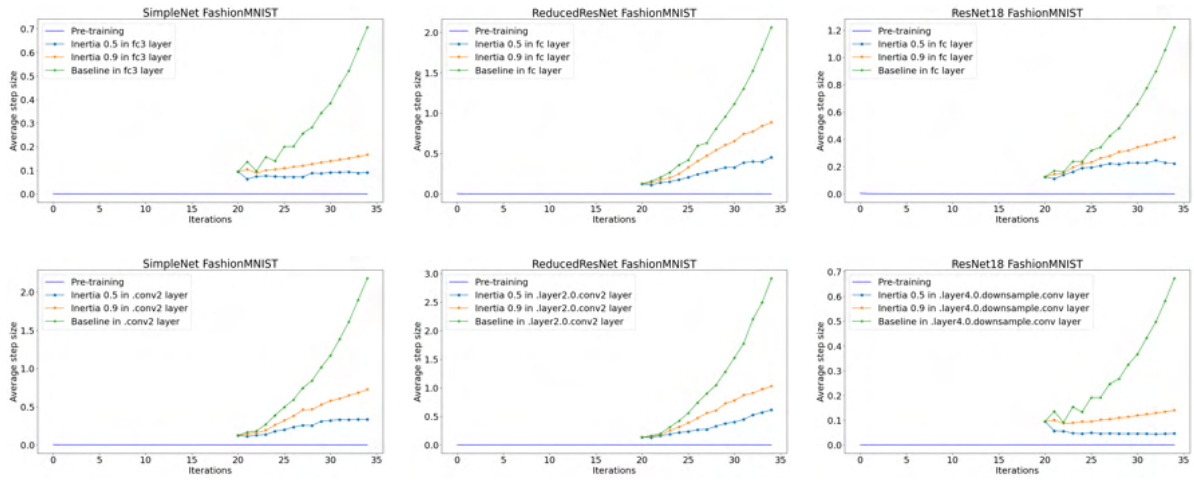

Figure 30: Average step size in the FashionMnist dataset

Another positive result of reducing the inertia value was that the degradation of the average training loss of the population was significantly reduced as we can see in Figure 31. These results on step size and average validation loss were manifested in every dataset and for all models. However, the best validation loss did not have any relevant improvements overall when compared with the baseline inertia values. A possible reason for this failure to improve might be related to the size of the steps being taken. Despite their explosion in later iterations being more under control when reducing the inertia values, they are still significantly bigger then the values the gradient based approach presents.

5.3.4 Positional Bounding

Experiments in positional bounding using f values of 1 and 2 (identified as DBounds and DBounds2 respectively) did not show any improvement in the validation loss of the models. As shown in Figure 32, we witnessed some success in containing the degradation of the performance in models, however, at the same time, this boundary definition approach proved to be very volatile and

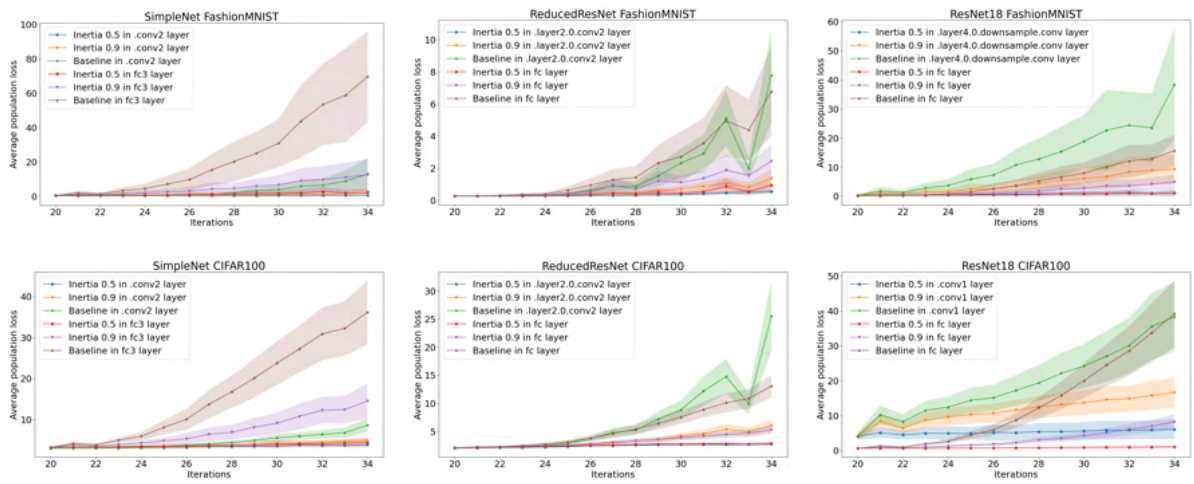

Figure 31: Average training loss per inertia experiment

will in certain scenarios massively degrade a model's performance and keep it from improving as was the case with the convolutional layer on the SimpleNet model in the CIFAR10 dataset with $f = 1$.

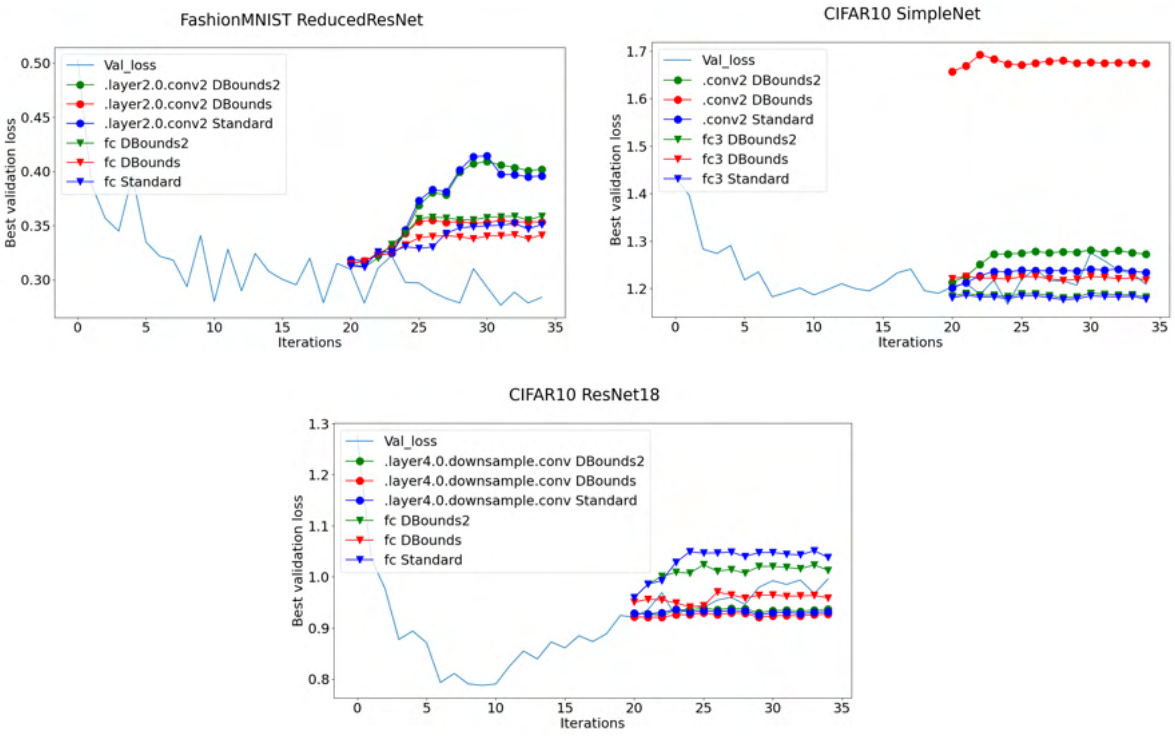

Figure 32: Validation loss of positional bounding experiment

We believe that the success observed in the prevention of performance degradation was mainly due to boundaries keeping particles from reaching unreasonable positions in the

search space. However when set incorrectly positional boundaries can doom any changes of a model improving performance. Given these results, the complexity of this task, and the resources that would have to be used in order to find a good boundary for each of these environments, we elected to leave further exploration of this PSO component for future works.

5.3.5 Velocity Bounding

Similarly to the previous experience with positional bounding, we also experimented with the bounding of velocity. After experimenting with a factor of 1 and 2, results showed that the velocity bounding is instrumental in reducing step size and consequently prevent the degradation that was seen in previous iterations. However the difference between these two factor values was very small which led us to experiment with more diverging factors of 1 and 10 (named DVBounds and DVBounds2 respectively).

In this experiment the overall behaviour if the PSO improved dramatically. The best validation loss was improved over the baseline PSO in all settings except one (in the CIFAR10 dataset, the linear layer of the SimpleNet model achieved a slightly lower validation loss). This improvement seems to be related to the regulation of the step size which was much closer to the values observed with gradient based optimization and did not show any exponential growth throughout iterations as demonstrated in Figure 33.

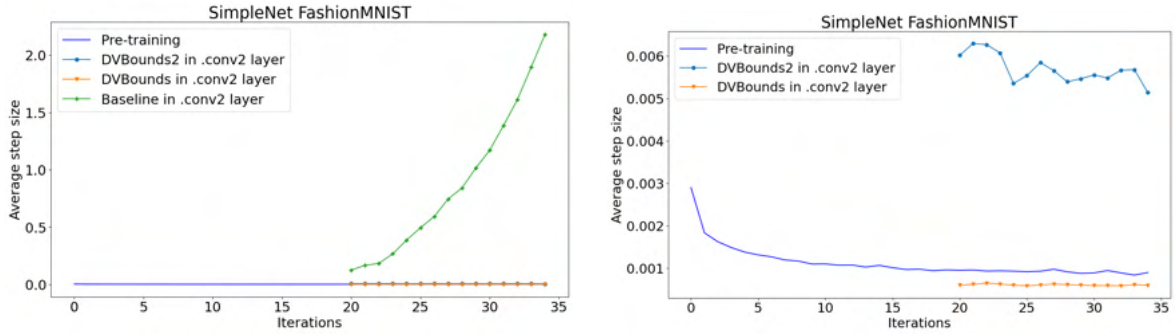

Figure 33: Step size in SimpleNet model with the FashionMNIST dataset

This control in the step size stopped the algorithm from degrading its performance which led to improvements in the average population training loss and best validation loss. Unlike the baseline PSO where the average population training loss would degrade, in this experiment the average training loss was improving or stagnant throughout training as shown in Figure 34.

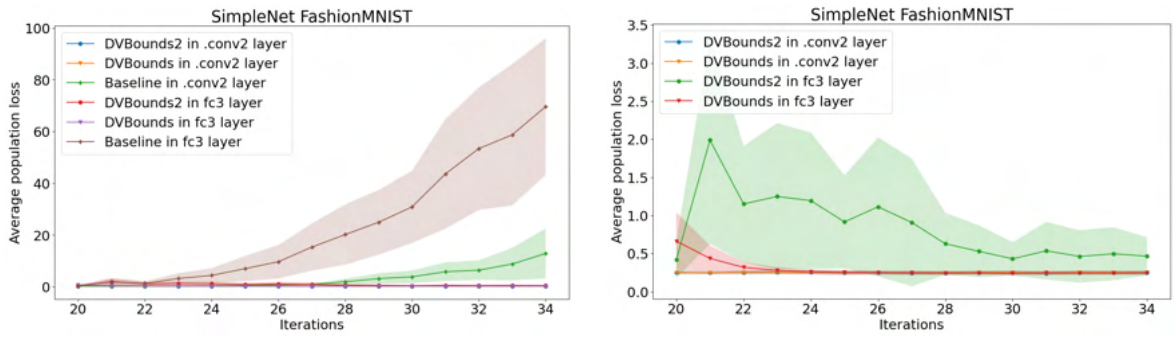

Figure 34: Average population training loss

Additionally, in the CIFAR10 dataset, the PSO algorithm was able to improve performance in the ReducedResNet model with both velocity clamping factors, achieving better results when the f factor was set to 10. More speciffically, with this factor value, the validation loss improved by 1.7% and 1.4% over the pre-training values.

Another effect of the reduction in the magnitude of the step size is that there is a significant reduction in standard deviation of the training loss of the population as well as a reduction in the number of times the best solution is changed as shown in Table 5.

Experiment	% of times best solution changes	Std. of training loss
DVBounds	19%	0,12
DVBounds2	15%	0,22
Standard	66%	2,90

Table 5: Effects of velocity bounding

We hypothesize that these results were possible due to the fact that smaller step sizes ensure that particles don't move to *unreasonable* positions for each search space. Given the reasonable success of this experiment, we decided that in the following experiments, we will use the DVBounds2 velocity bounding settings to ensure step size remains controlled whilst trying to maximize some of the exploration ability that was partially restrained by the smaller step size.

5.3.6 Population size

In the PSO algorithm population size can be viewed as a parameter that increases variation in algorithm 's behaviour. With this is mind, we expected that by increasing population size we

would see an increase in the exploration ability of the algorihtm.

In the FashionMnist dataset, all parameter sets were quite similar in throughout training in terms of validation loss of the best individual. On average, the lower population had the lowest validation loss of the parameter sets, followed by the normal and higher population size. This pattern is observed in all datasets, altought the differences between different configurations are always minimal.

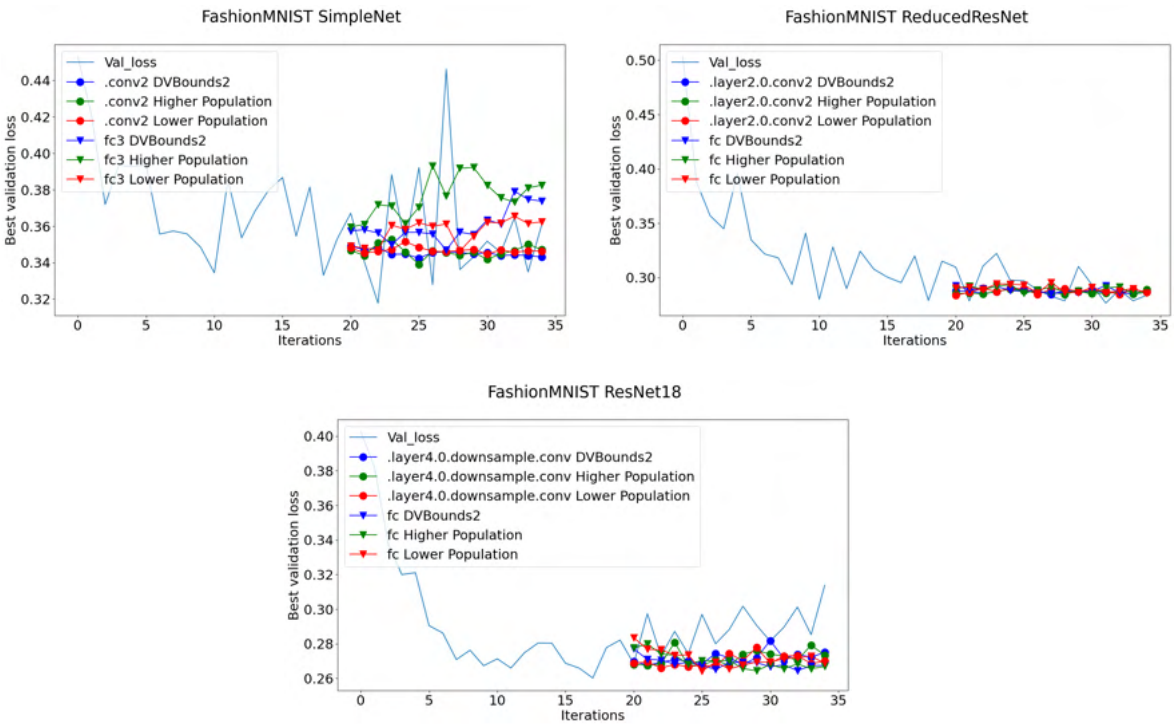

Figure 35: Best validation loss of population experiments in FashionMnist

In this dataset, step size also showed no noticeable differences between population sizes. When looking at the percentage of times the best solution changed we see an inverse relation with population size, with the percentage decreasing as the number of individuals in the population grow. Once again, this findings are seen in all datasets, as shown in Figure 36.

In the CIFAR10 and CIFAR100 datasets, changes in population size had an even lesser impact in the performance of the algorithm. We hypothesize that the reason for this might be that the numbers of individuals even in the highest setting might still be too small for any change to be noticeable, particularly given the number of dimensions involved. Inline with this thinking we find that by taking the best population size per each dimension, layers with more then 10 000 dimensions have an average population size of 122.9, whilst layers with dimensionalities bellow 1000, the average population size is 63.8 as show in Figure 37.

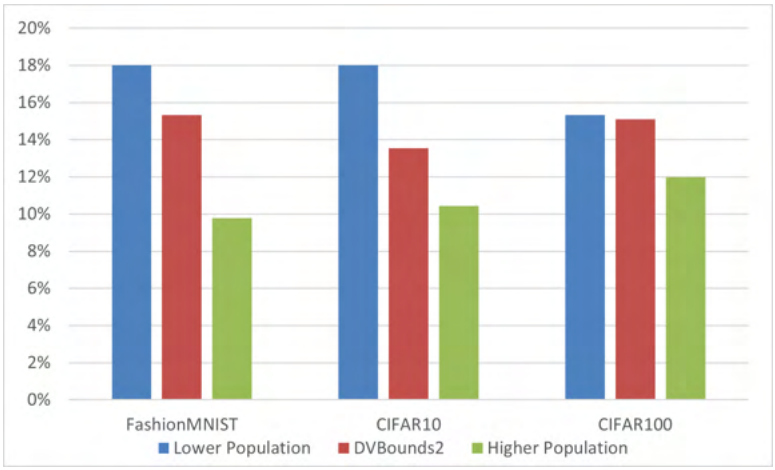

Figure 36: Average % of best solution changes

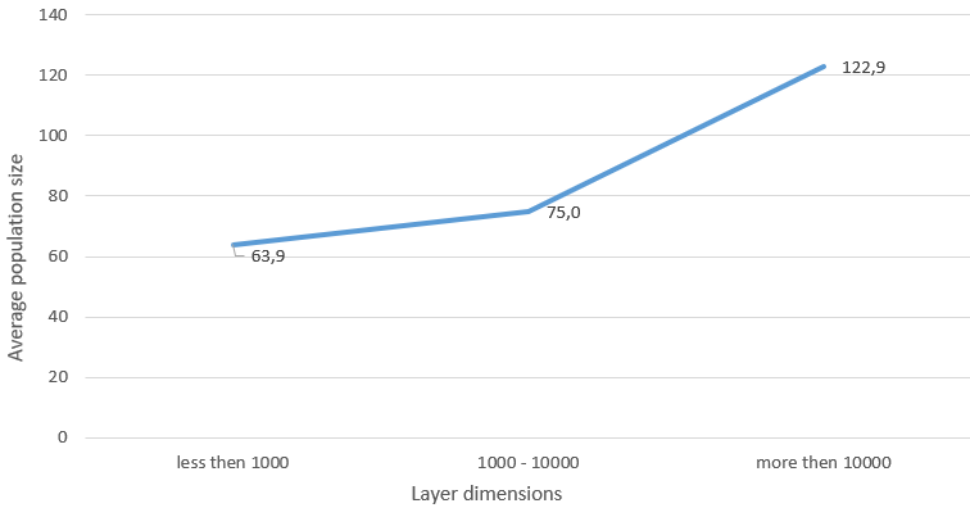

Figure 37: Average population size per layer dimensions

Given the results obtained in this experiment, and the apparent indifference to population size, at least given the values we did, we decided to reduce population size moving forward in order to reduce the computational cost of this optimization approach

5.3.7 Cognitive and Social factors

The results of increasing the social factor to 3 and the cognitive factor to 1 and vice-versa again failed to show any significant improvements to the overall performance of the model. For this experiment we believed a higher cognitive factor would benefit the explorability of the algorithm which could have led to breakthroughs in the validation loss. However, results showed that, even though a higher cognitive value performed better on average then the higher social,

it still did not out performed the balanced setting as shown in Figure 38.

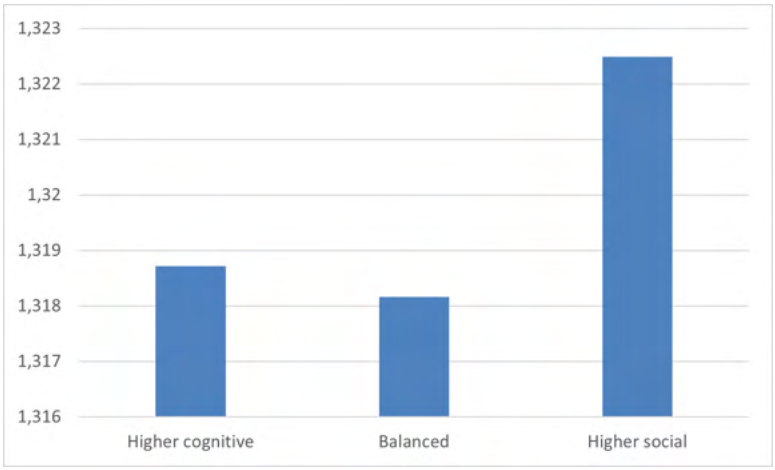

Figure 38: Averaged best validation loss

5.3.8 Future work

To further explore this methodology of optimizing DNNs, we believe further experimentation can be done, particularly with methods that attempt to combine PSO mechanics with gradient based optimization. A possible approach could be to add a gradient based optimization component that could then either replace the cognitive component of the particles or simply be added to the velocity update formula. Additionally, evolving the model with the PSO makes it simpler to take advantage of parallel computing which allows a much faster exploration of the search space. This base *PSO-GD* approach could then be further developed by introducing lifespan to particles and to allow particles to spawn neighbors giving a more dynamic behaviour to the algorithm. Based on our results in the optimization of the single layer with gradient based optimization, exploring discrete learning rate changes during the training of the models could also be worth exploring. This learning rate variations would also play well with the previously described approach of spawning new particles. For example, if the best particle spawns a clone of itself which is then optimized with a different learning rate, this new particle could possibly break new ground on the optimization process that would not be possible to its *parent* particle due to its low learning rate common to a longer running training process with ADAM and other optimizers. All the approaches presented above are made significantly more feasible due to the process of reducing the dimensions of the problem by choosing a single layer to optimize, which not only allows more optimization models to be tried but also reduces the computational cost

of the optimization process. Finally, several hybrid PSO approaches could be tested with this single layer optimization.

6 CONCLUSION

In conclusion, the use of the PSO algorithm to perform weight optimization of DNNs appears to be very ineffective, particularly in the higher dimensional spaces.

In our baseline experiments, it was immediately noticeable that the PSO algorithm was not able to improve upon the results the regular gradient based optimization had achieved up until iteration 20. At best, this approach made no significant change to the validation loss and at worst a steep performance degradation was observed.

In this first experiment we also provided evidence to reject our hypothesis that the replacement of the best solution inherent to the behaviour of the PSO, was not harming the optimization of the validation loss. Additionally, after reviewing the step size charts resulting from this trial we found that the step size was exploding throughout the training of the models. With these findings, we came to the conclusion that these behaviours are linked to the step size. This statement is backed by the observation that, when comparing the same neural network in the same dataset, the layer on which the PSO had a higher step size would always show higher performance degradation. Furthermore we found no relation between dimensionality and performance since this degradation occurred regardless of whether the degrading layer had a higher or lower number of dimensions. However, we did see a decrease in the ability of the PSO to change the validation loss in environments with higher complexity.

Our experiments with positional bounding showed this approach to be extremely volatile. Positional bounds that were set too tight made every movement of the particle go out of bounds and be randomly reset. Furthermore, defining the positional bounds in relation to the initial solution led to boundaries that varied substantially which led to very inconsistent results across the datasets and models.

Finally, experiments with velocity bounding showed the best results of all our experiments so far. This approach showed great success in mitigating the step size explosion as well as reducing degradation across datasets and models. Further work can be done on improving how the values are set so that the movement of particles is not too restricted and the PSO can

still express itself.

Looking at the effects of population size on the performance of the PSO we reached the conclusion that a lower population had better results on most settings. However, when analyzing the best population size per dimension we also found that population size and layer dimensions are correlated with one another. Given these results we believe that the values of population size used in our experiment could possibly not be differenciable enough for us to see the true effects of bigger populations on results. On the other hand, a higher number of individuals increases the computational cost of the approach which leads us to recommend that population size be adjusted in relation to the maximum number of iterations that a use case allows for.

After experimenting with the inertia values we found that, as expected, by reducing the inertia value, the exponential growth in step size previously observed was significantly reduced. However, validation loss did not show improvements over the results achieved in pre-training. The lower inertia value of 0.5 performed well across models and layers in the FashionMnist dataset, whilst the standard 1.2 and 0.9 values performed well on the CIFAR10. In the CIFAR100 dataset inertia showed very little impact in validation loss, with all variations performing very similarly to one another. Moving forward, a possible approach to take would be to combine an adaptive inertia with a less restrictive velocity clamping method.

Finally, balancing the cognitive and social factors in different combinations also seemed to make no direct impact on the performance of the PSO, with bigger variations in results apparently being related to changes in the scale of the step size.

Moving on to the single layer gradient descent optimization experiment, the results showed us a marked improvement over our baseline PSO approach. With this gradient based method, we were able to improve the best validation loss in several environments, particularly when working with the FashionMnist dataset. Of note was the performance of this method when applied to the layer selected by our pruning techinique which showed improvements on SimpleNet and ResNet18 regardless of the dataset considered. Furthermore the importance of the layer selection can be seen particularly in the CIFAR100 dataset with the ResNet18 model in which the linear layer degraded along the fully trained model but the convolutional layer reversed the trajectory and showed improvements on validation loss.

To conclude, although the PSO algorithm was not successful in improving validation loss, we believe the layer selection method presented can be further explored, given the results

obtained in conjunction with the single layer gradient based optimization, as an approach to speed up the improvement of validation loss in DNN training.

7 BIBLIOGRAPHY

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Nicola Fronzetti. Predictive neural network applications for insurance processes, 09 2019.

[3] Wilson Wongwe. Multimodal-fitness-landscape. http://www.wilsonmongwe.co.za/understanding-the-em-algorithm/multimodal-fitness-landscape/, 2015.

[4] Abiel Aguilar-González, Miguel Arias-Estrada, Madain Perez-Patricio, and Jorge camas anzueto. An fpga 2d-convolution unit based on the caph language. *Journal of Real-Time Image Processing*, 04 2019.

[5] Comparison between a normal and a residual block. https://d2l.ai/chapter_convolutional-modern/resnet.html.

[6] Beatriz A. Garro, Humberto Sossa, and Roberto A. Vazquez. Design of artificial neural networks using a modified particle swarm optimization algorithm. In *2009 International Joint Conference on Neural Networks*, pages 938–945, 2009.

[7] Serkan Kiranyaz, Turker Ince, Alper Yildirim, and Moncef Gabbouj. Evolutionary artificial neural networks by multi-dimensional particle swarm optimization. *Neural networks : the official journal of the International Neural Network Society*, 22:1448–62, 07 2009.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[9] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[10] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 07 2011.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[13] Angela H. Jiang, Daniel L.-K. Wong, Giulio Zhou, David G. Andersen, Jeffrey Dean, Gregory R. Ganger, Gauri Joshi, Michael Kaminsky, Michael Kozuch, Zachary C. Lipton, and Padmanabhan Pillai. Accelerating deep learning by focusing on the biggest losers. *CoRR*, abs/1910.00762, 2019.

[14] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signsgd: compressed optimisation for non-convex problems. *CoRR*, abs/1802.04434, 2018.

[15] Tyler B Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[16] B. Rylander M. Cells. Neural network learning using particle swarm optimizers. *Advances in Information Science and Soft Computingg; WSEAS Press: Cancun, Mexico;*, page 224–226, 2002.

[17] E. T. Oldewage. The perils of particle swarm optimization in high dimensional problem spaces. 2017.

[18] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995.

[19] A. CAUCHY. Methode generale pour la resolution des systemes d'equations simultanees. *C.R. Acad. Sci. Paris*, 25:536–538, 1847.

[20] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[21] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. 1983.

[22] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[24] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. 2017.

[25] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[26] E. Azadani, Seyed Hossein Hosseinian, and Poria Hasanpor. Optimal placement of multiple statcom for voltage stability margin enhancement using particle swarm optimization. *Electrical Engineering*, 90:503–510, 09 2008.

[27] Y. Del Valle, J.C. Hernandez, G.K. Venayagamoorthy, and R.G. Harley. Multiple statcom allocation and sizing using particle swarm optimization. In *2006 IEEE PES Power Systems Conference and Exposition*, pages 1884–1891, 2006.

[28] Elre T. Oldewage, Andries P. Engelbrecht, and Christopher W. Cleghorn. The merits of velocity clamping particle swarm optimisation in high dimensional spaces. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017.

[29] Adam P. Piotrowski, Jaroslaw J. Napiorkowski, and Agnieszka E. Piotrowska. Population size in particle swarm optimization. *Swarm and Evolutionary Computation*, 58:100718, 2020.

[30] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 69–73, 1998.

[31] Karin Zielinski and Rainer Laur. Stopping criteria for a constrained single-objective particle swarm optimization algorithm. *Informatica (Slovenia)*, 31:51–59, 03 2007.

[32] Mingquan Chen. Second generation particle swarm optimization. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 90–96, 2008.

[33] Rui Mendes, James Kennedy, and João Neves. The fully imformed particle swarm: Simpler, mabe better. *Evolutionary Computation, IEEE Transactions on*, 8:204 – 210, 07 2004.

[34] J.J. Liang, A.K. Qin, P.N. Suganthan, and S. Baskar. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. *IEEE Transactions on Evolutionary Computation*, 10(3):281–295, 2006.

[35] Zhi-Hui Zhan, Jun Zhang, Yun Li, and Yu-hui Shi. Orthogonal learning particle swarm optimization. *IEEE Trans. Evolutionary Computation*, 15:832–847, 12 2011.

[36] Yue-Jiao Gong, Jing-Jing Li, Yicong Zhou, Yun Li, Henry Shu-Hung Chung, Yu-Hui Shi, and Jun Zhang. Genetic learning particle swarm optimization. *IEEE Transactions on Cybernetics*, 46(10):2277–2290, 2016.

[37] Nandar Lynn and Ponnuthurai Suganthan. Ensemble particle swarm optimizer. *Applied Soft Computing*, 55, 02 2017.

[38] Beatriz A. Garro and Roberto A. Vazquez. Designing artificial neural networks using particle swarm optimization algorithms. *Intell. Neuroscience*, 2015, 01 2015.

[39] Chunkai Zhang, Huihe Shao, and Yu Li. Particle swarm optimisation for evolving artificial neural network. 4:2487–2490 vol.4„ 2000.

[40] C. Zhang and H. Shao. An ann's evolved by a new evolutionary system and its application. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, volume 4, pages 3562–3563 vol.4, 2000.

[41] Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2018.

[42] L. L. Custode, C. L. Tecce, I. Bakurov, Mauro Castelli, A. Della Cioppa, and Leonardo Vanneschi. A greedy iterative layered framework for training feed forward neural networks. pages 513–529, 4 2020.

[43] Wei-Zhen Lu, Huiyuan Fan, and S Lo. Application of evolutionary neural network method in predicting pollutant levels in downtown area of hong kong. *Neurocomputing*, 51:387–400, 01 2003.

[44] Jing-Ru Zhang, Jun Zhang, Tat-Ming Lok, and Michael R. Lyu. A hybrid particle swarm optimization–back-propagation algorithm for feedforward neural network training. *Applied Mathematics and Computation*, 185(2):1026–1037, 2007. Special Issue on Intelligent Computing Theory and Methodology.

[45] Jeng-Fung Chen, Quang Hung Do, and Ho-Nien Hsieh. Training artificial neural networks by a hybrid pso-cs algorithm. *Algorithms*, 8:292–308, 06 2015.

[46] Tae-Young Kim and Sung-Bae Cho. Particle swarm optimization-based cnn-lstm networks for forecasting energy consumption. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 1510–1516, 2019.

[47] K. O. Stanley and R. Miikkulainen. Efficient evolution of neural network topologies. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 2, pages 1757–1762 vol.2, 2002.

[48] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, page 762–767, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[49] Yong Liang, Kwong-Sak Leung, and Zong-Ben Xu. Neural network training using genetic algorithm with a novel binary encoding. In Derong Liu, Shumin Fei, Zengguang Hou, Huaguang Zhang, and Changyin Sun, editors, *Advances in Neural Networks – ISNN 2007*, pages 371–380, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[50] Parsa Esfahanian and Mohammad Akhavan. Gacnn: Training deep convolutional neural networks with genetic algorithm, 2019.

[51] Hyeyoung Park, Masato Inoue, and Masato Okada. Learning dynamics of neural networks with singularity – standard gradient vs. natural gradient. volume 3157, pages 282–291, 09 2004.

[52] Mark Ainsworth and Yeonjong Shin. Plateau phenomenon in gradient descent training of relu networks: Explanation, quantification and avoidance. *CoRR*, abs/2007.07213, 2020.

[53] K. Fukumizu and S. Amari. Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural Networks*, 13(3):317–327, 2000.

[54] Weili Guo, Yuan Yang, Yingjiang Zhou, Yushun Tan, Haikun Wei, Aiguo Song, and Guochen Pang. Influence area of overlap singularity in multilayer perceptrons. *IEEE Access*, 6:60214–60223, 2018.

[55] Bohan Zhuang, Chunhua Shen, and Ian D. Reid. Training compact neural networks with binary weights and low precision activations. *CoRR*, abs/1808.02631, 2018.

[56] A. Ebrahimi, S. Luo, and R. Chiong. Introducing transfer learning to 3d resnet-18 for alzheimer's disease detection on mri images. In *2020 35th International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1–6, 2020.

[57] Tawsifur Rahman, Muhammad E. H. Chowdhury, Amith Khandakar, Khandaker R. Islam, Khandaker F. Islam, Zaid B. Mahbub, Muhammad A. Kadir, and Saad Kashem. Transfer

learning with deep convolutional neural network (cnn) for pneumonia detection using chest x-ray. *Applied Sciences*, 10(9), 2020.

[58] Piotr Szymak, Paweł Piskur, and Krzysztof Naus. The effectiveness of using a pretrained deep learning neural networks for object classification in underwater video. *Remote Sensing*, 12(18), 2020.

[59] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[60] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[61] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[62] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.