Investigating tunability of hyperparameters of a convolutional neural network through genetic algorithms

Alexandar Tsvetanov (5092671) Martin Rønning (2331136)

Abstract

This paper introduces a hyperparameter optimization method for convolutional neural networks. For the choice of hyperparameters to optimize, we gather most of the widely used hyperparameters in the field. The algorithm used for the optimization was a uniform crossover-based genetic algorithm using family tournament selection. The paper also delved into the topic of optimal ranges for hyperparameter optimization and hyperparameter tunability. Hyperparameter tunability is the concept of how much performance gain we can get for optimizing a particular hyperparameter, as opposed to giving it an informed default value. We performed a simple experiment to optimize all of the hyperparameters. The results gave insights for the field of hyperparameter tunability, particularly for finding optimal ranges.

1 Introduction

Computer vision is the technology that allows computers to understand imagery. The technology is a subcategory of artificial intelligence and has found numerous applications, among them in self-driving vehicles, medical imaging and 3D model building [8]. The key is to identify features or properties of an image and to relate them in terms of labels or annotations, then allow a trained model to identify unlabeled features based on a certain probability. It is precisely to improve this probability that is always the aim of computer vision, in order to say with some certainty what is actually present in an image.

One of the most used techniques in computer vision is the convolutional neural network (CNN). It is convolutional because it uses the principle of convolution by multiplying sections of the input image, where the "microscope" scanning these sections are referred to as kernels. The CNN architecture usually requires several convolution layers to extract useful features followed by fully connected layers to find the patterns between the features. After the architecture is decided the model is then trained on the data. At every step of the process of deciding the architecture and training the model, there are hyperparameters to be decided for the best possible performance.

The process of choosing the most optimal parameter has to be carefully considered because every parameter has a diminishing return and the effects of adjusting parameter values can be complex, where changing one hyperparameter value might influence the performance of another. Usually, the parameters are chosen at random or by using the same as ones of a similar model, or by following some kind of standard in the relevant field. The problem with this less informed approach is that some parameters can differ widely from application to application so a more informed approach is needed. There are many possible heuristic-based approaches, but this paper will focus on genetic algorithms.

Genetic algorithms are a class of optimization algorithms in the family of evolutionary algorithms, which are algorithms inspired by natural processes, particularly Darwinian natural selection. In the case of genetic algorithms, this includes a selection operator, a crossover (reproduction) operator and a mutation operator. The representation of a chromosome to be reproduced varies depending on the problem domain the genetic algorithm is being applied in. Finding a suitable representation can massively improve its solutions.[9]

1.1 Hyperparameters of the convolution

The first several layers of a CNN are the convolution layers. The first parameter here will be how many of these layers we need. Too little and we won't be able to find complex features and too much and we will waste computational resources without significant improvement.

After that, for every layer, we have to determine two factors: how big we want the filter to be and how many filters we will apply to the image per layer. This way we determine how much information we will extract at each step of the convolution.

Another two parameters that can be optimized, but are more task dependent, are the stride and the padding. The stride is the distance that the mask moves between iterations and is usually set to 1. The other one is padding which determines how many pixels we will add around the image and is mainly used to get more information around the edges and to resize the image.

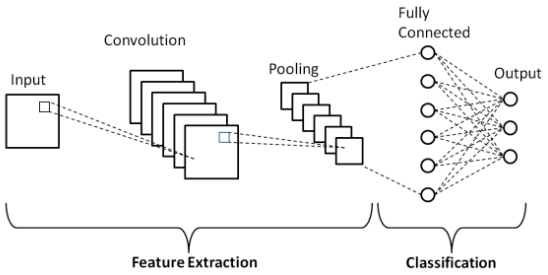

Figure 1: Visualization of a convolutional neural network [5]

The second type of layer we typically use between the convolution layers is the pooling layer. This type of layer does not gain any information, it only reduces the image and so the space in which we are searching. The pooling layer also has two hyperparameters. One is the size of the mask, the same as the convolution, but the second one is the type of operation that the mask does. The most used operation is averaging or getting the maximum in the mask.

1.2 Hyperparameters of the dense layers

In the convolution networks, after the features are extracted by the convolution layers, the patterns between them have to be determined for the classification. Here the convolution layers come into place and the first hyperparameter will be how many of them we want. With more fully connected layers it will be possible to model complex relationships between the features but too much and we will lose the more simple ones.

The other parameter after choosing how many layers we need is their size and the number of neurons in each of them. One other element that is not in the dense layers but usually complements them is the dropout. Dropout is a way to turn off some neurons by chance to prevent over-relying on them. Here there are two possible hyperparameters: Do we use dropout after a layer and what is the chance that a neuron will be turned off at every training iteration.

One element that can be used in both types of layers is batch normalization. The idea is similar to dropout by putting the weights in a closer range to reduce overreliance. The hyperparameter here will be whether we are using it after a particular layer. Another element used in both types of layers is the activation function applied after a layer.

1.3 Hyperparameters of the training phase

After the architecture of the model is set, it still has to be trained on the data. In this process, there are still several hyperparameters to choose from.

The first one is the number of epochs: how many times we will iterate over the data. To determine how much of the data we will use we set a batch size as another parameter.

We can also determine how the learning process will proceed through the learning rate, the momentum and the regularization parameters. The learning rate determines how fast we want the model to assimilate new facts. On the other hand, the regularization is to prevent overruling on some particular set of weights by punishing large values for weights. The momentum is a parameter which determines how much a previous update will influence the current one.

1.4 Genetic algorithms for hyperparameter optimization

Genetic algorithms have several benefits over other optimisation techniques, such as the ability to navigate away from local minima, which is a negative characteristic of random search optimisation. Meanwhile, grid search optimization is forgetful, meaning it does not take the prior iteration of the search into account to the next iteration, again in contrast to genetic algorithms. Thus, we are focusing on optimizing the hyperparameters through genetic algorithms in our study.

However, finding the right hyperparameters to tune is a complex problem, as the effects of tuning them are often codependent on each other, as well as the range of possible values not being obvious from before experimenting. This introduced the concept of tunability from the work of Weerts et al. (2020)[4]. We can

apply this concept in genetic algorithms in order to have a more sensible representation of the chromosome to be reproduced, and thus preserve the characteristics of an optimal solution to the next generations.

1.5 Evaluation metrics of a CNN model

A vital part of the genetic algorithms is the fitness function. To be able to improve it the algorithm needs information on whether the model performs better or worse with these hyperparameters. To measure this we introduce several evaluation metrics of the performance of the model.

The first and the most basic one is accuracy. It measures what percentage of all predictions are correct. In general, it is a valuable measurement, but it did not take into account problems such as class imbalance. This is highlighted when one class has much fewer examples in the data from another one, so that the best solution becomes to just label all of them as one class.

Better metrics for deciding the correctness, and in order to avoid class imbalance, are precision and recall. Precision is the amount of correctly labeled objects from one class from all labeled of that class and recall is the amount of correctly labeled of one class from all of that class. By using these two metrics we can find the F1-score which is a combination of both of them. This way we avoid the problem of class imbalance.

2 Related work

Ayan (2024) [1] implemented a genetic algorithm-based transfer learning approach to optimize the hyperparameters of a CNN in order to classify insects. Hyperparameters optimized included the freezing ratio, number of fully connected layers, number of neurons in a fully connected layer and the dropout rate. The fitness evaluation metrics used were (accuracy + precision + recall + F1-score) divided by 4. Furthermore, they used a roulette wheel selection method. Using the datasets Deng, D0 and IP102, all insect images, they achieved 97.58%, 99.89% and 71.84% accuracy scores respectively for each dataset.

Meanwhile, İnik (2023)[2] optimized a CNN for classifying urban sounds through a Particle Swarm Optimization algorithm (PSO). The hyperparameters they selected for the architecture optimization were the number of layers, number of filters per layer, filter size, stride and number of neurons in a fully connected layer. For the training hyperparameter optimization, they selected the number of epochs for obtaining the best CNN, the number of epochs for training the best CNN, dropout rate, mini-batch size and initial learning rate. After building different CNN models and selecting the three best for each dataset of ESC-10, ESC-50 and Urbansound8k, all datasets with urban sound samples, achieved 88.50%, 74.85% and 91.17% accuracy scores, respectively for each successful CNN model. Then, after augmenting the datasets, increasing their sizes and retraining the models, they scored 98.64%, 96.66% and 98.45%, respectively.

Lastly, Johnson et al. (2020)[3] optimized the number of layers, the number of filters and the filter size with a novel genetic algorithm using a sequential crossover operator. This emphasized diversity in the population in earlier generations and intensified the selection pressure in the latter generations. Using the datasets MNIST, CIFAR10 and CALTECH256, all general datasets, their approach achieved 99.56%, 84.55% and 33.3% accuracy, respectively.

3 Methodology

For the genetic algorithm, multiple selection methods exist. Experimenting with different selection methods might lead to more quickly achieving the optimal hyperparameter values. Furthermore, the stochastic element of the genetic algorithm can be experimented with. Trying different cutoff values for what individuals are selected will influence the selection pressure, which in turn might create more or less dominant solutions.[12][11]

As mentioned in Ayan (2024)[1], a recurring problem is getting stuck in local minima. The genetic algorithm implemented in this paper, to our knowledge, did not use any specific local search. This is another possible addition to the methodology to improve the performance of the genetic algorithm. Examples of local search are multi-start local search, iterated local search and genetic local search.

Similar to the work of Johnson et al. (2020)[3], one can also use alternative crossover operators to accomplish different patterns, such as a slow or fast convergence, dominated or non-dominated solutions, or similar. The behavior of the genetic algorithm and the patterns that arise also lead to varying results. Therefore, trying alternative genetic algorithms can influence the performance of the CNN.

In our study, we aim to first implement a genetic algorithm to optimize selected, tunable hyperparameters, then to implement a CNN with which we can test the performance of the optimized hyperparameters.

For the second phase of the project (post-draft), we thus aim to experiment with the possible hyperparameter candidates for optimization, as well as experimenting with at least one alternative genetic algorithm. The source code for this project is available on GitHub.

3.1 CNN implementation

We used the PyTorch library for the implementation of the CNN. The hyperparameters we are optimizing are the number of convolution layers, the number of kernels per layer, the size of the kernels, the stride of the kernels, the padding of the convolutions, the number of pooling layers, the size of the pooling, the stride of the pooling, type of pooling, number of dense layers, number of neurons per dense layer, activation function, whether we are using dropout, dropout rate, are whether we are using batch normalization, learning rate, training epochs, batch size, type of optimizer and L1 and L2 normalization rates. The implementation of the CNN starts with the construction of the architecture of the model. For this, we use three types of layers: convolution, pooling and fully connected. In order to reduce the searching space of the best hyperparameters for the architecture we used the standard LeNet structure of the layers and only optimized their numbers.

The typical CNN structure consists of several instances of going from a convolution layer to a pooling layer, for all convolution and pooling layers, and then proceeding to several fully connected layers to process the correlations between the features. However, as we allow values in certain ranges, which will be specified later, for the number of convolution and pooling layers, we add to the typical structure: if there are more convolution layers compared to pooling layers, or vice versa, we go from convolution layer to pooling layer as usual equally many times as the number of the lesser layer type, and then finally connect the remaining number of the greater layer type with the fully connected layers.

One assumption needed for the model to function is that we have at least one convolution and one fully connected layer. All other hyperparameters regarding a particular type of layer are set to all of the layers of that type. The training procedure is set to work with all types of optimizations and amounts of data that the hyperparameters require. To ensure that the model we are going to train is valid we introduce two methods. The first one is to limit the scope of possible values of parameters that can make the model invalid like the kernel sizes, strides and number of convolution and pooling layers. The second method is calculating the receptive field output of the feature extraction part (convolution and pooling) and ensuring it is greater than this output is greater so that there will be no information passed to the dense layers. In order to speed up the optimisation of the hyperparameters by the genetic algorithm we used ranges for the parameters that are similar to the ones used in models from the studies we examined by Sharma et al. (2019) and Wojciuk et al. (2022). After some initial testing before doing the experiment, we adapt ranges that have a visible negative impact on the learning process, beyond that of reasonable negative modifications expected from the genetic algorithm. This was the case with the momentum, which in very high values made the gradient explode and ruined any meaningful insight about the model performance. For the evaluation part we collect data from each epoch and calculate the final loss, the accuracy as well as the F1-score. Their summation is what we use as part of the fitness function for the genetic algorithm.

Algorithm 1 Calculate Output Size (in order to identify invalid models)

```
num_conv > 0 or num_pooling > 0 if num_conv > 0 then
    out_dim ← out_dim − kernel_size
    out_dim ← out_dim + 2 ∗ padding
    out_dim ← ⌊out_dim/conv_stride⌋
    out_dim ← out_dim + 1
    if out_dim < 1 then
        return 0
    end
    num_conv ← num_conv − 1
end
if num_pooling > 0 then
    out_dim ← out_dim − pool_size
    out_dim ← ⌊out_dim/pool_stride⌋
    out_dim ← out_dim + 1
    if out_dim < 1 then
        return 0
    end
    num_pooling ← num_pooling − 1
end
out_dim ← int(out_dim × out_dim × num_kernels)
return out_dim
```

Hyperparameter	Range
num_conv	1–5
num_kernels	1–20
kernel_size	3–5
conv_stride	1–2
num_pooling	0–2
pool_size	2
pool_stride	1–2
num_dense	1–5
num_neurons	100–200
padding	0–5
activation_fun	0–3
pool_type	0–1
dropout	0–1
dropout_rate	0.2–0.5(float)
batch_norm	0–1
learning_rate	$1 \times 10^{-5} - 1$(float)
epochs	1–50
batch_size	20–100
momentum	0.5–0.95 (float)
l1_norm_rate	$1 \times 10^{-5} - 1$(float)
optimizer	0–2
l2_pen	$1 \times 10^{-5} - 1$(float)

Table 1: Ranges of Hyperparameters

3.2 Datasets

The model was trained on the FashionMNIST dataset. It consists of 70000 grayscale images of clothes in 10 classes. Each of the images is 28x28 pixels. The reason we chose this dataset for the task was partly due to the size of the images being enough to train a model with only a couple of layers, which makes it computationally manageable for the optimization process. While still being sufficiently complex for the training task, as opposed to the standard MNIST dataset which features are easily modeled, it is not too complex so that we do not require excessively computationally expensive operations to extract the relevant features.

3.3 Genetic algorithm implementation

For the implementation of the genetic algorithm, we were not able to experiment with different variants of it due to time constraints of our project. Therefore, we implemented a fairly standard genetic algorithm and used this for our experiment. The algorithm takes the number of members of a population, the number of generations and the mutation rate. Each member of a population is a representation of a hyperparameter configuration, represented in code as a class of variables, named HPChromosome. Each variable thus corresponds to one hyperparameter, where the possible values are all defined as a range of reasonable values determined by us. When instantiating one HPChromosome, random values are selected within these ranges for all hyperparameters, and this thus makes up for one possible hyperparameter configuration.

The amount of times that the selection, mutation and tournament procedures will take place depend on the amount of generations. The aforementioned procedures will be further explained later. The fitness function will be executed during each tournament procedure twice, while each tournament procedure is executed for half the population, which effectively means the fitness function will be executed the same number of times as the number of generations.

Fitness function was implemented as such:

$$f(x) = \frac{(1 - \text{loss}) + \text{accuracy} + \text{F1\_score}}{3} \tag{1}$$

As the loss difference, accuracy and F1-score all have range 0–1, the fitness score too will have range 0–1.

In order to obtain these evaluation metrics we run the CNN with the corresponding hyperparameter configuration on 10% of the training data. This is an expensive operation, making the runtime of the program very slow. We could have selected a smaller portion of the training data to train the CNN, but we also wanted to make sure that the trends of the hyperparameter configuration were effectively conveyed by its performance. To save some time, we do not call the fitness function more than necessary. Once the fitness function for one configuration has been executed, this value is saved for the remainder of the run. It is only necessary to run the fitness function at the beginning of the run when creating the population, and when creating new children during the crossover procedure.

The selection procedure is executed for each generation, where we select random pairs of the population to do crossover. Our selected crossover operator is the uniform crossover. Since our representation can be considered as a fixed array of ranges, absolute positioning cannot be changed, i.e. we cannot give values of a range of index 1 to the value at index 2. Uniform crossover is one crossover operator that respects absolute positioning and is also fairly simple to implement. The algorithm iterates over the length of the hyperparameter configuration (in our case 22) and for each index, randomly (50% chance) selects the i-th

hyperparameter of the first parent to give the first child and the i-th hyperparameter of the second parent to give the second child, and vice versa. This results in two random combinations, children, of the two parents.

After crossover has been executed per pair and two children are produced, the tournament procedure is executed where then the two most fit configurations are chosen (where children are prioritized should they be equal to parents). In the same process, there's a small chance of mutation for the selected (pre-tournament) pair, depending on the given mutation rate. This is done by generating a random float number between 0 and 1, and comparing it against the mutation rate. If the number is greater than the mutation rate, mutation is done on one of the hyperparameters. There are many different approaches to mutation, and we would have liked to test different mutation rates as well as mutating all hyperparameters as opposed to just one, but this would instead have to be done in future research.

3.4 Tunability

One concept we researched for this paper was the idea of tunability. Because of the enormous search space of the hyperparameters we were looking for ways to find which are the most useful parameter to optimize so we can save computational resources. In general, the solution to this problem is to perform extensive testing of different hyperparameters for a particular variety of datasets and use a statistical approach to determine the most useful hyperparameters and ranges. First, we looked at the research of Sharma et al. (2019)[7] and Wojciuk et al. (2022)[10] for their research on optimal hyperparameter values on classification tasks. The Sharma et al. (2019)[7] paper residual neural network on 10 famous datasets with benchmark performance and then used functional ANOVA to analyse the contribution of each hyperparameter. The initial ranges were chosen by research to similar experiments end the personal experience of the authors.

Wojciuk et al. (2022)[10] Conducted a similar experiment but on a smaller scale. They used the Functional ANOVA with CNN on three datasets. The difference here is that they both optimize the ranges and the hyperparameters importance for the models. This study provided more general results for the importance of parameters and their ranges on more simple types of models.

Probst et al. (2018)[6] present the idea that many parameters are usually performing well enough with their default values and most of the improvement comes from only a small amount of them. The method uses a surrogate model with default hyperparameters and based on them determines improvement of the model after optimizing parameter only by one. To define the default parameters of the surrogate method large-scale research was done to look at the best values of different parameters in many sets. By combining this information a crossed default configuration of hyperparameters was constructed. From these baseline values, we can define the best-performing value of a parameter and the performance gain of optimising it per set by changing only it and measuring the overall improvement of the model. The ranges of values were constructed as probability distributions over the best-performing values for the different sets. The best value of a parameter per set is the one that gives the biggest improvement in the performance of the model with all default values and the one where it is the only one optimized. The approach generalizes the effects when multiple pairs of hyperparameters are optimized simultaneously for all the sets. This way we can obtain the best combination of hyperparameters per set that add the most performance gain. In future work on this project, we can try to incorporate this method to choose the most optimal set of hyperparameters to test on.

The results obtained by the mentioned papers can be used as an informed starting point for choosing ranges of hyperparameters and which one to spend resources to optimise. We used this information for determining our ranges for the hyperparameters. As a future work more wide range research can be implemented to find optimal ranges and improvement potential of hyperparameters using the above-mentioned statistical methods.

4 Results

We chose a baseline model initially partly based on values of aforementioned papers, partly based on being somewhat informed in the domain. For instance, we chose the learning rate to not be too low, but still much lower than what we put as the upper bound in the range for learning rate, as we think a too high learning rate will result in too volatile behavior to properly train the model.

Hyperparameter	Value
num_conv	5
num_kernels	16
kernel_size	3
conv_stride	1
num_pooling	2
pool_size	2
pool_stride	2
num_dense	4
num_neurons	120
padding	1
activation_fun	0
pool_type	0
dropout	1
dropout_rate	0.2
batch_norm	1
learning_rate	1×10^{-3}
epochs	10
batch_size	64
momentum	0.9
l1_norm_rate	1×10^{-3}
optimizer	0
l2_pen	1×10^{-5}

Table 2: Baseline hyperparameter configuration, scoring an average of 0.95 (2 s.f.) fitness over 20 runs on 10% training data

Running the genetic algorithm with population size 20 with mutation rate 0.2 and for 20 generations yielded surprisingly good results. We did not get a very diverse set of good solutions, as some of the top performing ones shared similar traits. Notice for instance the learning rate, being identical to one another among the top 5 performing solutions, while being consistently different and also of a much higher value among the worst performing ones.

Furthermore we can see that this learning rate is an order of magnitude higher than the baseline model's learning rate, but still significantly lower than the maximum of 1. Also we can see that the momentum is significantly higher in the baseline compared to our best solutions, and that in fact our worst solutions correspond more with almost maximum momentum.

Another finding was related to the l1_norm_rate and the l2_pen, as they are both much lower in the best performing solutions as opposed to the worst performing ones. This finding also matches that of the baseline, which too have a low l1_norm_rate and l2_pen value.

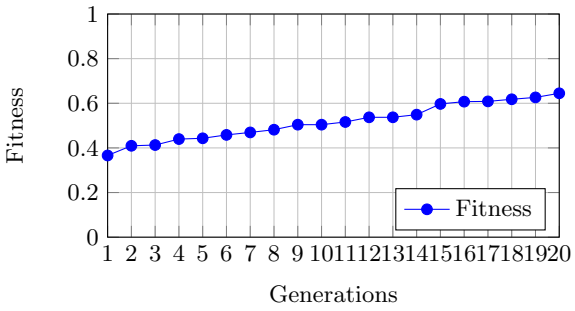

Figure 2: Average fitness over generations

Hyperparameter	1st	2nd	3rd	4th	5th
num_conv	2	2	2	2	2
num_kernels	5	15	5	5	15
kernel_size	3	3	3	3	3
conv_stride	1	1	1	1	1
num_pooling	1	1	2	2	0
pool_size	2	2	2	2	2
pool_stride	1	1	1	1	2
num_dense	1	1	1	1	1
num_neurons	140	181	140	148	181
padding	0	0	0	3	0
activation_fun	2	2	0	0	2
pool_type	0	0	0	1	0
dropout	0	0	0	0	0
dropout_rate	0.3686	0.4287	0.4287	0.4287	0.2805
batch_norm	1	1	1	1	1
learning_rate	0.0132	0.0132	0.0132	0.0132	0.0132
epochs	41	41	41	41	7
batch_size	47	91	47	47	63
momentum	0.5228	0.6318	0.693	0.6318	0.6377
l1_norm_rate	0.0756	0.0756	0.0756	0.0756	0.0756
optimizer	0	0	0	0	1
l2_pen	0.0830	0.0830	0.0830	0.0830	0.0830
fitness (2 s.f.)	0.88	0.87	0.87	0.87	0.86

Table 3: Best hyperparameter configurations

Hyperparameter	1st	2nd	3rd	4th	5th
num_conv	1	1	2	5	4
num_kernels	5	10	10	16	17
kernel_size	3	3	3	3	3
conv_stride	1	1	1	2	2
num_pooling	0	2	2	2	2
pool_size	2	2	2	2	2
pool_stride	1	1	1	1	2
num_dense	2	4	4	4	2
num_neurons	187	161	166	148	161
padding	0	4	2	3	3
activation_fun	0	1	0	3	0
pool_type	1	0	0	0	1
dropout	0	0	1	0	0
dropout_rate	0.4525	0.3362	0.4102	0.3362	0.4916
batch_norm	0	1	0	0	1
learning_rate	0.6820	0.7273	0.7273	0.4540	0.4895
epochs	36	15	28	11	15
batch_size	29	95	25	99	47
momentum	0.9302	0.6623	0.7994	0.9078	0.5030
l1_norm_rate	0.2350	0.9519	0.6831	0.6491	0.1613
optimizer	1	0	0	2	0
l2_pen	0.7537	0.5189	0.5189	0.2404	0.9079
fitness (2 s.f.)	0.36	0.49	0.55	0.57	0.57

Table 4: Worst hyperparameter configurations

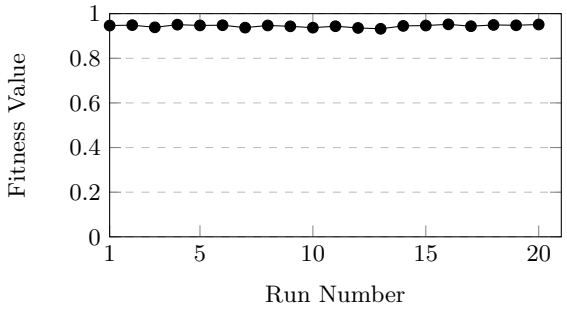

Figure 3: Best solution scoring an average of 0.94 (2 s.f.) fitness over 20 runs on 10% training data

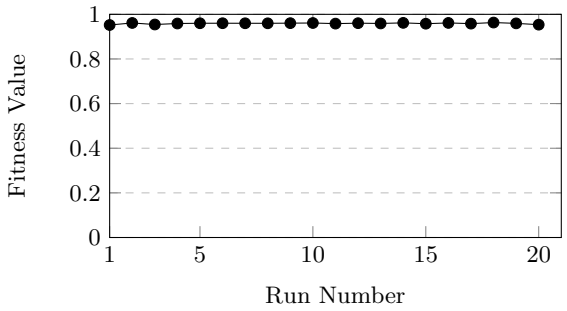

Figure 4: Best solution scoring an average of 0.96 (2 s.f.) fitness over 20 runs on 10% training data

5 Discussion

The results show that the average fitness generally progressively increased for each generation, which was expected, as this is the intended behavior for a genetic algorithm with a low mutation rate. It was however not guaranteed to always increase, as can be seen, although it never drastically decreased either. This was somewhat surprising, as in theory the better hyperparameter configurations should always be selected from the family, but this might be due to rare situations where very bad models are created and some existing models are mutated to be worse than they were, canceling out any slight increase of average fitness for that generation.

As explained, the CNN implementation calculated the output size of the model given the respective hyperparameter configuration and then determined whether it was a valid model or not. We tried to counteract this as much as possible with some initial testing to slightly tune the ranges, but could not completely eliminate that possibility. Therefore, some invalid models are created during the run, given a fitness value of -1, which is the worst possible fitness. This is a slightly problematic representation because there can be a scenario where there is a model with close to optimal hyperparameter configuration but is off by a very small value in a single hyperparameter which is then deemed as a very bad model. This will in turn discourage the selection of this model in further generations, as more fit models are preferred. At the same time, there is no current way to distinguish a very bad invalid model from a possibly very good invalid model. Therefore, we decided to stick with this representation, as it at the very least ensures we do not encourage very bad invalid models.

The conv_stride, num_dense, dropout, learning_rate, l1_norm_rate and l2_pen all followed the general same trend with values being identical among the best. This tells us that the value converged towards, and that the range probably was too high in variability. This further tells us that we can trim the range in order to get more solutions with various solutions around the value that was converged towards in order to get a more diverse set of solutions.

It is obvious that the population size as well as the number of generations should have been increased to see what fitness value the genetic algorithm could finally converge to, but this was regretfully not possible due to time constraints. Running the experiment took approximately half a day, although this was not precisely measured. Anyhow, the run time is very slow, thus doing experiments would require either a much faster implementation, which is unlikely to at all be possible, or a lot more time to perform the experiments. Nevertheless, it would be interesting to learn what are the limitations of the genetic algorithm in terms of finding the optimal hyperparameter configuration for future research.

5.1 Limitations

In general, there are many constraining factors for finding the optimal set of hyperparameters. The most pressing one is the exponential scale of necessary computational power when we add more parameters to be optimized simultaneously. This was the motivation to look at how useful optimizing a particular parameter will be. Another common problem in optimization is getting stuck in local minima. This is the phenomenon that occurs when a satisfactory result is achieved and no better one can be found unless the algorithm encounters locally worse results. With heuristic-based approaches, worsening the results to guide the algorithm in the right direction is discouraged behavior, so it would get stuck. One final limitation is that if the quality of the data is not adequate, the hyperparameter tuning can not compensate for that. Class imbalances can disturb the optimization process of the hyperparameters.

As for the implementation of the convolution network, there were several assumptions that we made in order to scale down the complexity of the optimization problem. One of them was that we assume that all layers of the same type have the same hyperparameters. In developing production models this is rarely the case. For example, the number of neurons per fully connected layer usually decreases on subsequent layers. Another assumption was that one pooling layer would follow one convolution like in popular architectures like Alexnet and LeNet but this is not always the case. Useful techniques like batch normalization and dropout can be used in different parts of the network but we made the assumption that we are either using them on every possible location or nowhere to make this parameter a binary choice instead of a vector with possible locations. One technique that has proven extremely useful in image classification is residual skip connections, but unfortunately, we found no way of adding it in a way that will not significantly increase the complexity, so we decided to leave it as a possible future improvement of the method.

Some possible limitations of the tunability algorithm that we want to add to our research can be that the approach won't scale with the number of hyperparameters we have and some of them have to be discarded.

6 Conclusions

This research provided a standard method of tuning hyperparameters for convolutional neural networks. Using a standard approach through an optimization heuristic for finding the optimal values of a simple model, we were able to incorporate a larger range of parameters to be tuned compared to most of the research in the area. Despite limited time to perform experiments, we got insight into the tunability of the ranges in order to find optimal values for each hyperparameter, for almost all widely used parameters. This can be used in further studies on similar datasets for similar models, and can find more insight through longer experiments. As discussed, we were able to find insights about which parameters were more prone to deviate in the given ranges and which ones can be defaulted. This can help future researchers in the field of tunability of hyperparameters.

References

[1] E. Ayan. Genetic algorithm-based hyperparameter optimization for convolutional neural networks in the classification of crop pests. *Arabian Journal for Science and Engineering*, 2023b.

[2] Ö. Inik. Cnn hyper-parameter optimization for environmental sound classification. *Applied Acoustics*, 2023b.

[3] Valderrama-A. Valle C. Crawford B. Soto R. Ñanculef R. Johnson, F. Automating configuration of convolutional neural network hyperparameters using genetic algorithm. *EEE Access*, 2020.

[4] C. Mueller A. Vanschoren J. J.P. Weerts, H. Importance of tuning hyperparameters of machine learning algorithms. *arXiv*, 2020.

[5] Rhee E. J. Phung, V. H. A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets. *Applied Sciences (Basel)*, 2019.

[6] Bischl B. Boulesteix A. Ayan E. Probst, P. Tunability: Importance of hyperparameters of machine learning algorithms. *arXiv*, 2018.

[7] Van Rijn J. N. Hutter F. Müller A. Sharma, A. Hyperparameter importance for image classification by residual neural networks. *In Lecture Notes in Computer Science*, 2019.

[8] R. Szeliski. *Computer vision: algorithms and applications*. Choice Reviews Online, 2011.

[9] M. Kleinnijenhuis A. J. Farmer D. Vie, A. Qualities, challenges and future of genetic algorithms: a literature review. *arXiv*, 2021.

[10] Swiderska-Chadaj Z. Siwek K. Gertych A. Wojciuk, M. The role of hyperparameter optimization in fine-tuning of cnn models. *Social Science Research Network*, 2022.

[11] Yan-M. Basodi S. Ji C. Pan Y. Xiao, X. Efficient hyperparameter optimization in deep learning using a variable length genetic algorithm. *arXiv*, 2020.

[12] Yoon-H. Kim H. Yoon H. Han S. Yoo, J. Optimization of hyper-parameter for cnn model using genetic algorithm. *ICECIE*, 2019.