Journal of Machine Learning Research XX (XX) X-XX Submitted X/XX; Published XX/XX

Speeding Up the Hyperparameter Optimization of Deep Convolutional Neural Networks

Tobias Hinz HINZ@INFORMATIK.UNI-HAMBURG.DE
and **Nicolás Navarro-Guerrero** NAVARRO@INFORMATIK.UNI-HAMBURG.DE
and **Sven Magg** MAGG@INFORMATIK.UNI-HAMBURG.DE
and **Stefan Wermter** WERMTER@INFORMATIK.UNI-HAMBURG.DE
Department of Informatics, Knowledge Technology Group
University of Hamburg
Vogt-Koelln-Str. 30, 22527 Hamburg, Germany

Editor: XX

Abstract

Most modern learning algorithms require the practitioner to manually set the values of many hyperparameters before the learning process can begin. However, with modern algorithms the evaluation of a given hyperparameter setting can take a considerable amount of time and the search space is often very high-dimensional. We propose the usage of a genetic algorithm to optimize the hyperparameters specifically for convolutional neural networks (CNNs). Additionally, we suggest using a lower-dimensional representation of the original data to quickly identify promising areas in the hyperparameter space. We compare the results of the genetic algorithm and the hyperparameter optimization on lower-dimensional inputs to random search and the "Tree of Parzen Estimators" (TPE) algorithm.

Our experiments show that the genetic algorithm finds hyperparameters that perform similar or better than those found by random search and the TPE algorithm. Additionally, they indicate that it is possible to speed up the optimization process by using lower-dimensional data representations at the beginning while increasing the dimensionality of the input later in the optimization process. This is independent of the underlying optimization procedure and considerably speeds up the hyperparameter optimization process, making the approach promising for future hyperparameter optimization algorithms.

Keywords: hyperparameter optimization, hyperparameter importance, convolutional neural networks, genetic algorithm, evolutionary algorithm

1. Introduction

The performance of many contemporary machine learning algorithms depends crucially on the specific initialization of hyperparameters such as their general architecture, the learning rate, regularization parameters, and many others (Bergstra et al., 2013b; Coates et al., 2011). Indeed, finding an optimal combination of hyperparameters can often make the difference between bad or average results and state-of-the-art performance (Bergstra et al., 2011; Swersky et al., 2013).

The goal of a typical machine learning algorithm A is to find a function f that minimizes some expected loss $\mathrm{L}(x; \mathrm{f})$ over samples x drawn independently from a distribution G_x. This is usually done by mapping a data set $X^{(train)}$ to the function f (Bergstra and Bengio, 2012).

Very often this is achieved by iteratively updating certain parameters θ to minimize the expected loss. However, the learning algorithm itself usually has parameters λ, commonly called hyperparameters, that need to be set manually before the optimization process can begin. This results in a learning algorithm A_λ and the function $\mathrm{f} = A_\lambda(X^{(train)})$.

Hyperparameter optimization tries to find the optimal set of hyperparameters $\lambda^{(*)}$ which minimizes the generalization error E for the given learning algorithm. This can be written as

$$\lambda^{(*)} = \underset{\lambda \in \Lambda}{\operatorname{argmin}} \; E_{x \sim G_x}[\mathrm{L}(x; A_\lambda(X^{(train)})].$$

This task becomes very challenging when the dimensionality of the hyperparameter space increases. Especially deep neural networks have tens of different hyperparameters that can be adjusted to any given input data set (Bergstra et al., 2011), resulting in a high dimensional search space. However, hyperparameter optimization problems usually have a low effective dimensionality: even though there is a significant number of hyperparameters, often only a subset of them has a measurable impact on the performance (Bergstra and Bengio, 2012). But, for a given learning algorithm different subsets of hyperparameters matter for different data sets (Bergstra and Bengio, 2012).

We focus on the hyperparameter optimization of one specific learning algorithm: convolutional neural networks (CNNs). In addition to the hyperparameters of standard artificial neural networks, CNNs feature hyperparameters such as the number of filters per convolutional layer and the filter size. One of the biggest challenges is that the evaluation of a given hyperparameter setting for CNNs can take a long time. This is especially the case for deeper models with a potentially high number of filters on each layer. As a result, the inputs to CNNs are often simplified or reduced in size, e.g. by reducing the resolution of images. However, recent studies show that images with higher resolution are advantageous for many classification tasks (Chevalier et al., 2015; Zheng et al., 2016). This is especially the case for images where the region of interest is small or when deformations resulting from the rescaling process degrade the performance. It is, therefore, preferable to use images with high resolution, even though this is usually impracticable especially for hyperparameter optimization purposes.

If hyperparameter values for images in a low and a high resolution are similar to each other, this could be used to find appropriate hyperparameters on images with low resolution and fine-tune them for the same images with high resolution. This is somewhat similar to hyperparameter optimization across data sets (Reif et al., 2012; Hutter et al., 2015). The idea here is that hyperparameters that are appropriate for a given data set might be a good starting point for optimization for other, similar, data sets. However, we do not use hyperparameters from different data sets, but instead, identify promising areas in the hyperparameter space on the lower-dimensional representation of the same data.

This work aims to examine two core problems: 1) the development of an evolutionary algorithm to optimize the hyperparameters of CNNs on any given input data and 2) whether it is possible to gain information about good hyperparameters for high-dimensional input from hyperparameters of the same data but in a lower-dimensional representation.

To solve the first problem we introduce a genetic algorithm that takes an arbitrary number of hyperparameters and then optimizes over all of them. For the second problem, we take this genetic algorithm and use it iteratively to optimize hyperparameters of CNNs

for image inputs with increasing resolution. That way, we have conceptually the same input data, but in different resolutions, i.e. different input dimensions. We then examine the dependencies between the hyperparameters found on the different input sizes to see if there are relationships present. This can be used to speed up the optimization process in future applications by starting with lower input sizes and increasing them iteratively during the optimization process.

2. Related Work

Traditionally, the choice of hyperparameters for a given problem is made by the experimenter. However, this requires a significant amount of experience, intuition, and trial and error. Additionally, results are usually not scientifically reproducible and sometimes suboptimal (Snoek et al., 2012). Recent results indicate that more sophisticated and automated approaches can find better hyperparameters – and thus achieve better results – than humans (Pinto et al., 2009; Coates et al., 2011; Bergstra et al., 2011; Snoek et al., 2012; Bardenet et al., 2013; Hutter et al., 2014).

Two of the most widely used methods are also two of the simplest: grid search and random search (Bergstra and Bengio, 2012). In grid search, a predetermined range of values is chosen for a given set of hyperparameters. Then a grid is constructed through every possible combination of all hyperparameter values. While grid search is easy to implement and trivial to parallelize, a big problem is that the grid grows exponentially with the number of hyperparameters. As a consequence, the resulting grid must not be too granular if it is to be evaluated in a reasonable amount of time. Together with a low effective dimensionality, the sufficiently granular grid is likely to be suboptimal since it will potentially cover many spaces of low importance while under-examining hyperparameters in areas of high importance.

Random search, on the other hand, draws a random value from a predefined distribution for each hyperparameter of interest. It is equally easy to implement and parallelize, but can have some advantages in higher-dimensional search spaces. Bergstra and Bengio (2012) empirically show that random search performs almost as or equally well in higher dimensional search spaces while being much quicker than grid search. Their explanation for this is that a grid of points gives an even coverage of the n-dimensional space, but at the cost of an inefficient coverage of each subspace. In contrast, random search produces points that are slightly less evenly distributed across the n-dimensional space but are more evenly distributed in each subspace. However, random search suffers from being non-adaptive, i.e. it does not take into account the results of previous evaluations and as a consequence may try hyperparameter settings that are unlikely to result in a performance increase.

One of the main problems of optimizing hyperparameters for learning algorithms is that it can take a long time to evaluate a given set of hyperparameters. As a result, sequential model-based optimization (SMBO) algorithms have been employed in many settings when the performance evaluation of a model is expensive (Hutter et al., 2009, 2011; Bergstra et al., 2011; Brendel and Schoenauer, 2011; Bardenet et al., 2013; Thornton et al., 2013). SMBO takes the approach of spending additional computing time on calculating the most promising next hyperparameter instantiation, with the goal that fewer evaluations of the learning algorithm itself are needed. To achieve this, SMBO algorithms employ a probabilistic

model to model the black box function f, which in this case is the learning algorithm. The model is built with any existing prior knowledge about the problem and point evaluations of f (Hutter et al., 2015). The model is then updated iteratively with the new knowledge gained by evaluating previously proposed hyperparameter settings. Based on the knowledge base obtained during the process, SMBO algorithms ideally improve the quality of the hyperparameters with only relatively few actual evaluations (Bergstra et al., 2011).

Bayesian optimization (Bergstra et al., 2011; Snoek et al., 2012; Swersky et al., 2013; Bergstra et al., 2013b) is one of the most used methods for SMBO and centers on building a probability model describing the performance given a hyperparameter configuration. The model is then continuously updated with new information gained by sample points providing information about the performance under a given hyperparameter configuration (Bergstra et al., 2013b). A common approach for Bayesian optimization is to assume that the unknown function that needs to be optimized is sampled from a Gaussian process (Rasmussen, 2004), which gives priors over functions that are closed under sampling (Bergstra et al., 2011; Snoek et al., 2012). However, other approaches, e.g. using decision trees (Hutter et al., 2009; Bergstra and Bengio, 2012) or neural networks (Snoek et al., 2015) have also been successful.

Another approach is to employ evolutionary algorithms for the search of optimal hyperparameters. Population-based optimization methods are well suited for optimization tasks over high-dimensional variable spaces, as they can evaluate many candidate solutions in parallel. They also offer the possibility of combining some sort of random search while utilizing the results of previous evaluations. Fiszelew et al. (2007), Young et al. (2015) and Navarro-Guerrero (2016, pp. 90-108) apply genetic algorithms to optimize a subset of hyperparameters for neural networks showing the general applicability of this method to the problem.

Zoph and Le (2016) train a recurrent neural network via reinforcement learning to find neural network architectures that are likely to yield a good performance on specific tasks. However, only architectural hyperparameters are optimized, while many other hyperparameters such as the learning rate and regularization parameters are manually chosen in the end. Additionally, the algorithm needs a high amount of resources, using several hundred GPUs and 15,000 hyperparameter evaluations during optimization. Smithson et al. (2016) use a neural network to predict the performance of a set of hyperparameters on a given data set. Candidate hyperparameters are sampled from a Gaussian distribution around previously evaluated solutions and given to a neural network for evaluation. If the neural network predicts a high performance the candidate solution is trained, tested and added to the training set for the neural network which predicts the hyperparameter performance. The neural network predictor is then retrained on the extended training set and a new candidate solution is sampled from the hyperparameter space.

Albelwi and Mahmood (2016) use the Nelder-Mead algorithm to optimize hyperparameters for CNNs. However, instead of using only the accuracy of a CNN as a measurement for its quality they propose the additional usage of mutual information results. The mutual information is obtained by using a deconvnet to visualize filters of trained CNNs. This visualization shows regions of the input that were learned by the respective filter. The quality of reconstruction—and by extension the quality of the CNN—is then measured as the mutual information between the original input and the reconstructed image. The cost

function for the Nelder-Mead is then a combination of both the CNN's accuracy and its ability for accurate visual reconstruction.

3. Methodology

Genetic algorithms (GA) are a population-based search algorithm and are well suited to solve high-dimensional and non-convex optimization problems. Through the definition of a fitness function, the quality of any given solution can be measured directly. Additionally, the optimization process is not based on gradients, meaning the fitness function does not need to be derivable.

The GA starts with randomly initialized solutions and evaluates their fitness. Then, a subset of solutions is chosen, with better solutions having a higher probability of being chosen. These solutions are recombined in an attempt to improve their fitness. Next, some of the new solutions are randomly mutated to further explore the parameter space. Thereupon the fitness of the newly "created" solutions is evaluated and the process is repeated until either a good enough solution is found, the quality of the solutions does not improve anymore, or a certain amount of time or computational work is met. Figure 1 shows a visualization of the process.

Since the GA has some stochastic elements we run it three times for the same input data. Due to the random weight initialization of a CNN, the training process itself also contains stochastic elements and ideally it would be repeated several times for each hyperparameter setting to obtain statistically significant data. However, this is impractical during the optimization process since it would drastically increase the time that is needed.

To somewhat circumvent the problem that we do not train a CNN repeatedly we save the best five CNNs at the end of each optimization process. After an appropriate amount of generations, the best individuals will likely converge to similar values for their hyperparameters. The average validation error for any one input setting and the corresponding optimization algorithm is therefore calculated as the average of the $3 \times 5 = 15$ best individuals found by the respective optimization procedure. This provides statistically more significant data since we do not rely on the performance of a given CNN that was only evaluated once, but rather on the average performance of 15 CNNs with similar hyperparameters.

The general process of the GA is depicted in Figure 1. After evaluating each CNN's fitness, some CNNs are selected as "parents". These parents are recombined to create "children" and these children together with the three best CNNs from the previous generation constitute the new generation. Finally, individual hyperparameters can mutate to increase the variance of hyperparameter values. Each optimization run is executed for a total of 30 generations.

We also measure the average time it takes to evaluate one generation. After each generation the three best CNNs are automatically transferred to the next generation without being evaluated again. Except for the very first generation, one generation therefore corresponds to 47 evaluations of CNNs. As a consequence, we have 50 evaluations in the first generation and 47 evaluations in the following 29 generations, for a total of $50 + 29 * 47 = 1413$ evaluations. We will now describe the general implementation of the GA.

Initialization For each optimization procedure with our GA, we initialize a population of 50 CNNs. The CNNs are encoded with a direct encoding strategy for which the hyper-

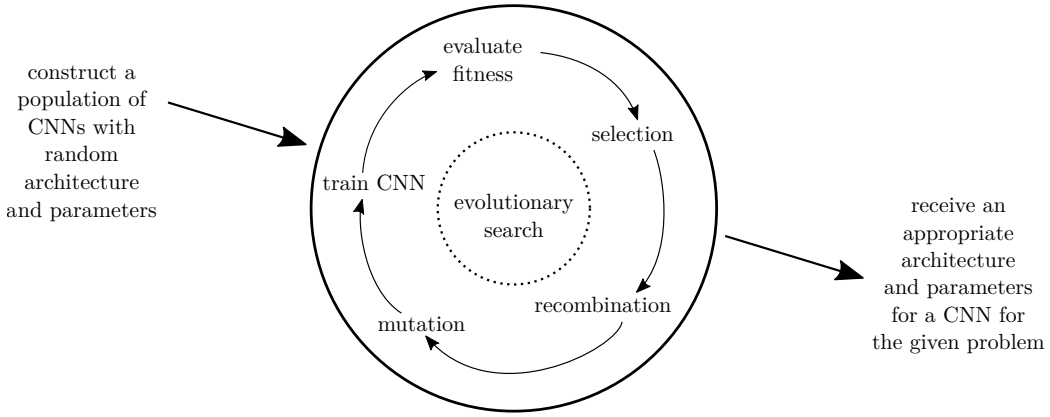

Figure 1: General procedure of the hyperparameter optimization algorithm.

parameters that are to be optimized are encoded as genes, while all other hyperparameters, such as the activation function, stay the same. The hyperparameters are initially drawn from a given distribution. However, through mutation, the hyperparameters can take values outside the original distribution during the optimization process. Table 1 presents the list of optimized and fixed hyperparameters used in each of our experiments.

We also adhere to common architectures and guidelines and ensure in a second step that a higher convolutional layer does not have fewer filters than the previous convolutional layer. Analogous to how we handle the filters in the convolutional layers we ensure that higher hidden layers do not have more hidden units than lower hidden layers. This is some domain knowledge for CNNs that is inserted into the optimization process to speed up the time until good results are found. See Figure 2 for an overview of the optimized hyperparameters.

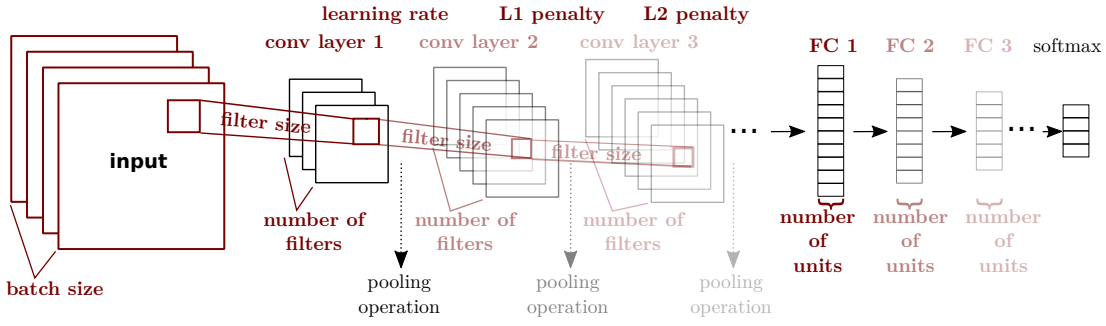

Figure 2: Overview of the hyperparameters that are optimized during the experiments.

Training and Fitness Evaluation Each CNN is trained using stochastic gradient descent for a maximum of 50 epochs. To shorten the overall training time we employ an early stopping strategy and abort training if the validation error does not decrease for five consecutive epochs of training. The CNN's fitness is its error on the validation set after training is completed. Since we want the validation error to be as small as possible, the overall goal is to minimize the CNNs' fitness values over time.

Parent Selection and Recombination To create the next generation of CNNs from the current generation we combine multiple methods. First, the best three CNNs are automatically added to the next generation, this technique is called *elitism*. Next, three randomly created "new" CNNs are also added to the next generation of individuals. Then, we randomly choose 44 parents from the current generation using tournament selection, by picking two random individuals from the current generation and ranking them according to their fitness. We then choose the better individual with a probability of 75%, while selecting the worse individual in 25% of the cases and add it to the pool of parents. These 44 parents are recombined to create 44 children, which fill up the pool of 50 individuals for the next generation. The recombination of two distinct parents is done via 1-point crossover. At the crossover point in the genome, the two parents are split up and the respective parts are recombined to form two new children. In order to retain some architectural features of a CNN we merge some individual hyperparameters during the recombination process. More precisely, all convolutional layers with associated filters and filter sizes, as well as all hidden layers with associated units are treated as individual hyperparameters.

Mutation After the recombination step, each individual's gene is mutated with a probability of 10%. Through this, the hyperparameters can take values that were not originally present. After the mutation step, it is again ensured that higher convolutional layers do not have fewer filters than lower layers and that higher hidden layers do not have more units than lower hidden layers.

Table 1: Overview of fixed and optimized hyperparameters during the experiments.

	Hyperparameters	Values
Fixed	Activation Function	Rectified Linear Unit
	Pooling Operation	Max Pooling
	Pool Size	(2,2)
	Weight Initialization	$\mathcal{N}(0, \frac{2}{n})^1$
	Momentum	0.9
	Early Stopping	5 Epochs w/o Improvement
	Max. Number of Epochs	50
Optimized	Filter Size	$\{(x,x) \mid (x > 2) \wedge (x \,\%\, 2 = 1)\}$
	Batch Size	$\{x \mid (x > 0) \wedge (x \,\%\, 10 = 0)\}$
	L1 Penalty	$\{0 \vee 10^{-x} \mid x \in \mathbb{N}_{>0}\}$
	L2 Penalty	$\{0 \vee 10^{-x} \mid x \in \mathbb{N}_{>0}\}$
	Learning Rate	$\{10^{-x} \mid x \in \mathbb{N}_{>0}\}$
	Number of Conv Layers	$\{x \mid x \in \mathbb{N}_{>0}\}$
	Filters per Conv Layer	$\{x \mid (x \in \mathbb{N}_{>0}) \wedge (x \,\%\, 10 = 0)\}$
	Number of Hidden Layers	$\{x \mid x \in \mathbb{N}_{>0}\}$
	Units per Hidden Layer	$\{x \mid (x \in \mathbb{N}_{>0}) \wedge (x \,\%\, 50 = 0)\}$

[1] n: number of incoming connections to one unit, see He et al. (2015)

Random Search and TPE For each setting, we also run random search and the TPE algorithm three times to optimize the hyperparameters. Since the GA consists of 30 gener-

ations of 50 individuals we give both random search and TPE the opportunity to evaluate 1500 hyperparameter settings. This makes it convenient to compare the progress in the optimization procedure for all three algorithms since we can split them into subgroups of 30×50 hyperparameter evaluations, which can be seen as an analogue to the concept of generations in the GA. In this context, the first generation of random search or the TPE algorithm consists of the first 50 hyperparameter evaluations. The second generation contains the hyperparameter evaluations 51-100, etc. For more details on the initialization and the search space of the two algorithms see Appendix A.

4. Experiment 1: Relationships between Hyperparameters

In the first experiment, we test the performance of our genetic algorithm (GA) on two distinct data sets. The extended Cohn-Kanade (CK+) data set (Lucey et al., 2010) consists of images depicting facial expressions of 210 adults and the task is to classify the displayed emotion. All images were converted to gray scale and resized to four different image sizes of 200x200, 128x128, 64x64 and 32x32 pixels respectively. For the hyperparameter optimization process, we split the data and use 70% for training and the remaining 30% for validation purposes. Since the classes do not have equal amounts of images we perform the split individually for each class, i.e. of each class we take 70% and add it to the training data, while the remaining 30% are added to the validation data.

The second data set used in experiment 1 is the STL-10 data set (Coates et al., 2011), which is made up from labeled images acquired from ImageNet. It consists of color images with size 96x96 pixels and contains ten classes. Each class has 500 images for training and 800 images for testing purposes. Similarly to the CK+ data set we converted the images to gray scale and resized them to sizes of 32x32 and 48x48 pixels. There exist ten pre-defined folds containing 100 images from each class for the training set. Training is performed using each of those folds at a time, i.e. using only 1000 images. The reported test set accuracy is then calculated as the average of the accuracy of each of the ten models on the test set. For the process of hyperparameter optimization, we follow the approach by Swersky et al. (2013) and use the first fold as training data while using the remaining 4000 images from the training set as our validation set.

4.1 Experimental Setup

We now describe the implementation and procedure of our GA. The initialization, recombination and mutation processes are identical for both data sets and are now described in more detail.

Initialization The initial learning rate is drawn from {1e-1, 1e-2, 1e-3}. These values somewhat constitute "default" suggestions for a good learning rate. The initial number of convolutional layers is drawn from {1, 2, 3} and can become any natural number greater than zero during the optimization process. While numerous recent CNNs feature many more convolutional layers we try to find a CNN with minimal complexity that is still able to solve the given problem. We, therefore, start with a relatively small number of convolutional layers, which can be increased by the algorithm over time if deemed advantageous. The same is true for the number of filters per convolutional layer, which can have a major impact

on the training time. Consequently, the number of filters per convolutional layer is also quite small in the beginning and drawn from {10, 20, 30, 40, 50}. We insert a max-pooling layer after each convolutional layer.

The same motivation, to keep the CNNs as small as possible, means that we initialize the number of hidden layers in the same way as the initial number of convolutional layers. Again, if necessary the number of hidden layers can increase during the optimization process. The number of units per hidden layer is also quite small in the beginning and is initialized as a multiple of 50 between 50 and 500.

The filter sizes are initialized as uneven numbers between three and seven. All filters on a given convolutional layer are initialized with the same quadratic filter size, i.e. 3x3, 5x5, or 7x7. These values are commonly seen in the literature (e.g. Cox and Pinto (2011) and Khorrami et al. (2015)), while larger filter sizes are usually reserved for bigger sized input images, typically greater than 200x200 pixels. However, if necessary the filter sizes can grow larger than 7x7 during the optimization process.

There is very little data on how to find a "good" batch size for a specific problem. However, Breuel (2015) empirically found that smaller batch sizes commonly perform better. We, therefore, start with comparatively small batch sizes drawn from {10, 20, 30, 40, 50}. Finally, the L1 and L2 regularization parameters for each CNN are drawn from {0, 1e-1, 1e-2, 1e-3, 1e-4}, which correspond to commonly seen "default" values (Bengio, 2012; Sermanet et al., 2013).

Recombination Figure 3 illustrates an example including only three hyperparameters: the learning rate, the convolutional layers and the hidden layers. All other individual hyperparameters, such as the regularization penalties, are treated identically and can come from either parent, depending on the location of the crossover point.

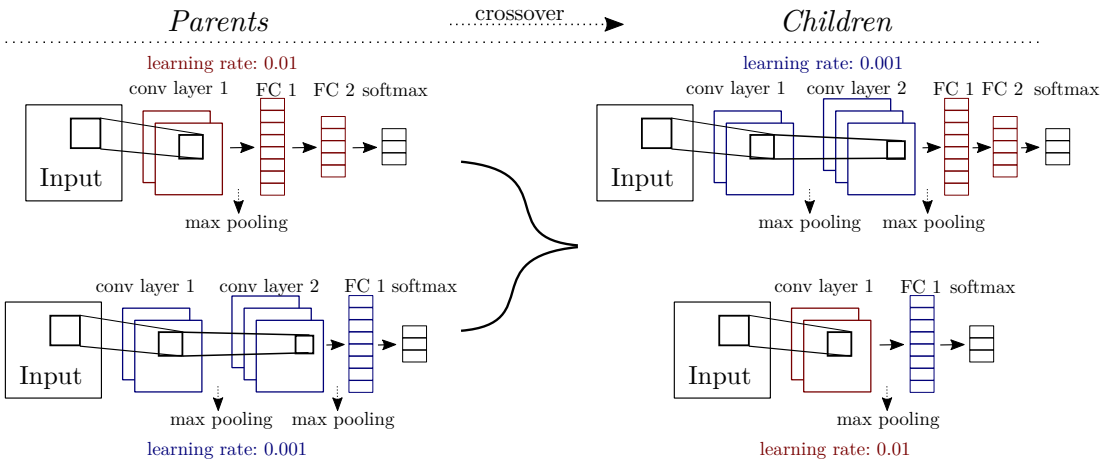

Figure 3: The two CNNs on the left are the parents with according architecture and learning rate. Each parent has one or two convolutional layers, one or two hidden, fully connected layers (FC) and one Softmax layer for the classification. After performing the crossover operation we obtain the two children on the right side.

Mutation Table 2 gives an overview over the mutation probabilities for the different hyperparameters. We have a slightly higher probability for decreasing the learning rate because we can already sample the biggest allowed learning rate of 0.1 from the very beginning. The tendency to decrease the learning rate is, therefore, advantageous if smaller learning rates need to be found. If they are mutated the number of the convolutional or hidden layers is in- or decreased by one with a probability of 50% each.

The L1 and L2 regularization parameters are mutated in the same manner as the learning rate, but need some special treatment in case they become zero. If the penalty is zero, there is a 40% chance that it will be increased. If this is the case it is either set to 0.001 or 0.0001, each with a probability of 50%. If one of the penalties becomes smaller than 1e-5, it is set to zero. The mutation parameters are set manually in the beginning of the optimization process. They are not fine-tuned for the specific optimization task, but rather represent intuitive knowledge about how hyperparameters are usually adapted during manual or grid search. The exact value of the mutation parameters does most likely not have an impact on the final solution, but may have an impact on how quickly the algorithm converges.

Table 2: Mutation parameters for the first experiment. The columns describe the probability of the state changes of a given hyperparameter in case it is mutated.

hyperparameter	10%	30%	50%	hyperparameter	10%	40%
learning rate L1 penalty L2 penalty	$\times 10^2$ / $\times 10^{-2}$	$\times 10$	$\times 10^{-1}$	# filters # hidden units filter size batch size	± 20 ± 100 $\pm(4\times 4)$ ± 20	± 10 ± 50 $\pm(2\times 2)$ ± 10

4.2 Results

The results obtained by the genetic algorithm are now compared to the random search algorithm, as well as the Tree of Parzen Estimators (TPE) algorithm. We will examine which algorithms find the best hyperparameters, how long it takes them and how the found hyperparameters differ between algorithms. Finally, we will look at the importance of the different hyperparameters for different resolutions and the similarity of hyperparameters values across different resolutions of the same images. See Appendix B for a detailed listing of the hyperparameter values for all algorithms and resolutions. As described at the end of the methodology section, we compare the results with the concept of generations where we define one generation as 50 consecutive hyperparameter evaluations.

Comparison of Hyperparameters

Time and Validation Error As expected the average time per generation increases along with the input size. For the GA, the highest resolution takes up to four times longer per generation than the smallest resolution. This is even more pronounced for random search and the TPE algorithm, for both of which the difference is up to eight-fold. We can

see that the GA is the fastest algorithm per generation, followed by random search. The TPE algorithm on average takes the longest to complete the equivalent of one generation.

While the difference is less pronounced for the smaller input size of 32x32 pixels, it becomes very obvious for bigger inputs. While the GA on average takes less than 21 minutes per generation for an input size of 200x200 pixels from the CK+ data set, random search takes more than 42 minutes and the TPE algorithm takes almost an hour. For the STL-10 data set and an input of 96x96 pixels the GA takes about 30 minutes per generation, random search takes a bit over 47 minutes and the TPE algorithm takes more than 160 minutes.

For the CK+ data set the resolution of the images does not seem to have too much of an impact on the accuracy, as the validation errors stay quite consistent across different resolutions. Images from the STL-10 data set, on the other hand, seem to benefit from higher resolutions, as the best average accuracy is always obtained with the highest resolution for all optimization algorithms. For the CK+ data set the validation error between the GA and the TPE algorithm is very similar. Both of them significantly outperform random search. The average validation error of the GA is $1.2\% - 3.5\%$ better than that of a random search. TPE on average performs $1.8\% - 3.2\%$ better than random search.

The difference in the validation errors between the GA and TPE are much smaller. In fact, except for the input size of 200x200 pixels, the differences are not statistically significant. Only for the biggest input size does the TPE algorithm perform better, outperforming the GA by 1.7%. For the STL-10 data set the TPE algorithm usually achieves the best average validation error. However, aside from the input size of 48x48 pixels, for which TPE has on average a 2.6% lower validation error than the GA, the difference is usually only $0.3\% - 0.6\%$. Both the GA and TPE again significantly outperform random search. See Tables 5, 6 and 7 in Appendix B for more details on the average errors on the validation set.

Learning Rate, Batch Size and Regularization Penalties All algorithms choose a similar range of learning rates for each given data set. For the CK+ data set the learning range is usually between $0.005 - 0.05$, for the STL-10 the learning rate is mostly $10 - 100$ times smaller. While the value ranges do differ to some extent between the algorithms, they are always consistent for any given algorithm and data set, irrespective of the image resolution for which the hyperparameters were optimized. Figure 4 was created using all data gathered by the three algorithms on the respective image resolutions and shows the impact of the learning rate on its own on the average validation error of the CK+ data set. We can see that the learning rate's impact is very similar across all input settings. The best learning rate seems to be located somewhere between 0.01 and 0.1 for all resolutions.

The same is true for the batch size, which also seems to be mainly dependent on the data set, but not on the specific resolution of the images. The L1 regularization penalty is often set to zero for most resolutions in both data sets, indicating that at least for these two data sets L1 regularization is detrimental to the final performance. The L2 penalty, on the other hand, is usually greater than zero, especially for the STL-10 data set.

Convolutional Layers All algorithms usually choose the same amount of convolutional layers for each resolution and data set. The number of convolutional layers increases with the resolution, moving from one/two up to five/four convolutional layers for the CK+/STL-

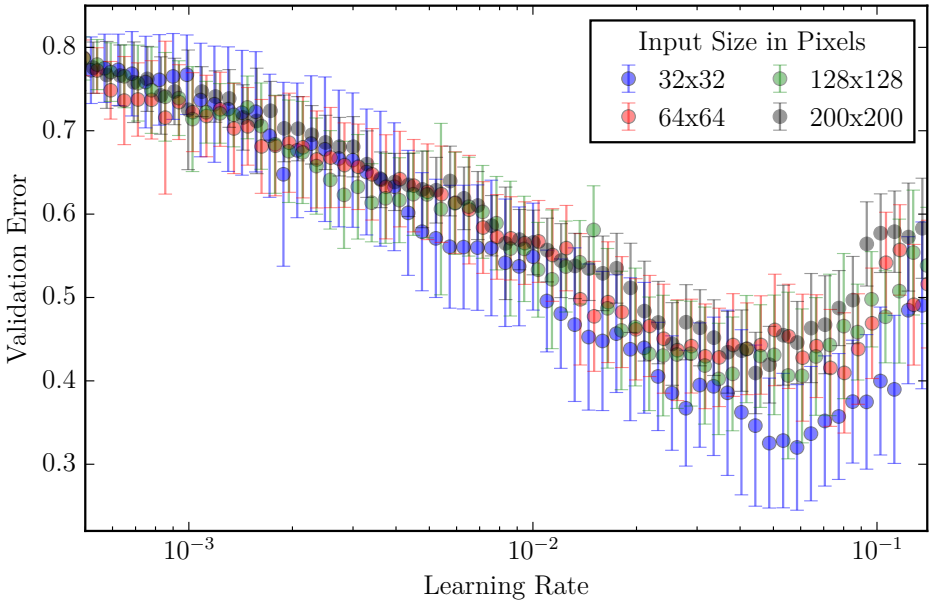

Figure 4: Relationship between the learning rate and the validation error (CK+ data set), aggregating the data of all three algorithms on the respective image resolutions.

10 data set. The filter size is also consistent between the algorithms. While the exact filter size seems to be of little importance for the CK+ data set, usually fluctuating between 3x3, 5x5 and 7x7, it is much more important for the STL-10 data set, almost always taking the value 3x3 for all layers and resolutions, see Tables 8, 9 and 10 in Appendix B.

However, there are considerable differences between the algorithms in the number of filters per convolutional layer. While the GA usually has a smaller number of filters ($10 - 100$), both random search and TPE use much more filters on each layer (up to 800 on the highest convolutional layer). However, the number of filters per layer stays again consistent when we only look at one specific algorithm and data set.

Hidden Layers The number of hidden layers is the same for all algorithms. For the CK+ data set one hidden layer seems to be enough, while the STL-10 data set seems to need two hidden layers. The number of hidden units is similar across all algorithms and resolutions. The exact value seems to be of little importance, as long as it does not fall below a certain threshold, roughly 100 for the CK+ data set and roughly 300 for the STL-10 data set, see Appendix B.

Importance of Hyperparameters

We now look at the impact of various hyperparameter subsets on the performance across the different input sizes. For this, we use a variant of analysis of variance (ANOVA), called functional ANOVA (Hooker, 2007) to analyze the importance of different subsets of hyperparameters as suggested by Hutter et al. (2014).

Figure 5 shows an overview of the – on average – most important hyperparameter subsets of the CK+ data set and their impact on the validation error. We can observe that the importance of hyperparameter subsets stays somewhat constant across the different input sizes. The only subsets that deviate strongly from this are the subsets that include the number of convolutional layers. However, this is to be expected as these are directly dependent on the input dimension. Other hyperparameters, such as the batch size and regularization parameters, are similarly important for all input sizes. Indeed, the top five and top ten of the most important hyperparameter subsets are virtually identical for all input sizes.

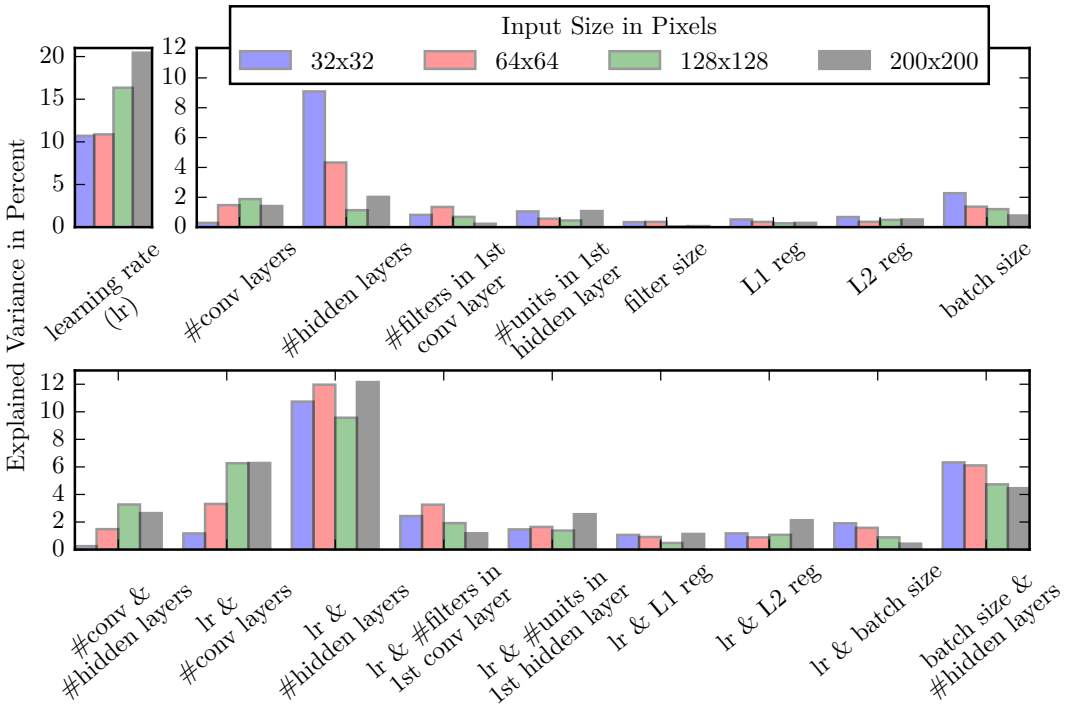

Figure 5: Explained variance of the validation error in percent on the CK+ data set. Depicted are the effects of the 18 most important hyperparameter subsets.

Figure 6 depicts the impact of the combination of the learning rate and the number of hidden layers on the validation error of the CK+ data set. Again this is very similar across all input sizes, identifying a learning rate between 0.1 and 0.01 together with one hidden layer as good parameters. Both Figure 4 and Figure 6 further indicate that not only the importance of hyperparameters is the same for different input sizes, but even the general "best" value for a given hyperparameter seems to be closely correlated. These results are closely reflected on the STL-10 data set. See Appendix B for visualizations of the impact of the hyperparameters on the STL-10 data set.

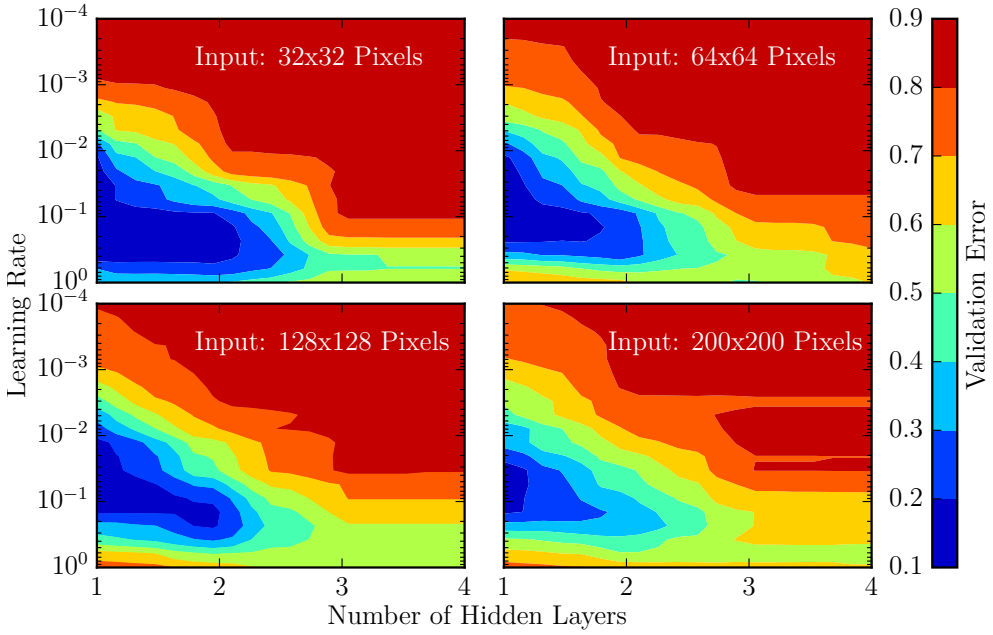

Figure 6: Relationship between the learning rate, the number of hidden layers and their impact on the validation error for the CK+ data set.

Comparison of Test Set Performance

For a more unbiased evaluation of the hyperparameters, we also test some hyperparameter settings on held-out test sets. For these tests, we choose the hyperparameter settings of each algorithm that performed best on the respective validation set. We adhere to the common guidelines about the test sets as detailed in Khorrami et al. (2015) for the CK+ data set and Coates et al. (2011) for the STL-10 data set. Table 3 gives an overview of the results without any additional regularization, as well as the best results found with additional regularization methods, such as dropout (Srivastava et al., 2014), max pooling dropout (Wu and Gu, 2015) and batch normalization (Ioffe and Szegedy, 2015).

Since the STL-10 data set only has 100 images per class for training it tends to overfit very quickly. We, therefore, used data augmentation with the combination of regularization methods that yield the best results on the test set. To create additional training data we took the images and applied some random transformations, such as mirroring, rotation by up to 30 degrees, shifting by up to 20 pixels and zooming into the images. In this way, we increase each training fold by a factor of ten. These additional images are stored so that each hyperparameter setting is trained on the exact same augmented data set. In this case, the training set then consists of the 1000 images of the original fold and 9000 additional images created from that fold through the above-mentioned techniques.

Table 3 shows that both the GA and the TPE algorithm significantly outperform random search when no additional regularization methods are applied. With the best combination of regularization methods the difference gets somewhat smaller, but especially for the STL-

Table 3: Results of 10-fold cross-validation of different hyperparameters on the CK+ and the STL-10 data set. See Table 11 in Appendix B for more details on the specific hyperparameters that were chosen for each *Setting*.

Data set	Found by	Setting	Regularization methods	Val error
Extended Cohn Kanade	Genetic Algorithm	Setting 1	none	0.057 ± 0.005
			dropout	**0.046 ± 0.003**
		Setting 2	none	0.057 ± 0.005
			dropout	**0.046 ± 0.002**
	Random Search	Setting 1	none	0.077 ± 0.041
			dropout	**0.043 ± 0.004**
	TPE	Setting 1	none	0.062 ± 0.001
			dropout+bn	**0.033 ± 0.001**
		Setting 2	none	0.079 ± 0.002
			dropout	**0.053 ± 0.003**
STL-10	Genetic Algorithm	Setting 1	none	0.544 ± 0.001
			dropout+aug	**0.467 ± 0.003**
	Random Search	Setting 1	none	0.569 ± 0.001
			dropout+aug	**0.482 ± 0.001**
	TPE	Setting 1	none	0.529 ± 0.001
			aug	**0.473 ± 0.001**

10 data set random search is still inferior. Additionally, we also observe that the GA is competitive with the TPE algorithm for both unregularized and regularized settings. All in all, the results are similar to the results obtained on the validation sets, which indicates that the hyperparameters are not fit specifically to the validation set, but rather are appropriate hyperparameters for this kind of input.

Discussion

Figure 7 shows the development of the best average validation error found by the GA and the TPE algorithm over time. For this, we take the average best validation error for each generation in the GA and the average best validation error for groups of 50 consecutive evaluations of the TPE algorithm. While the GA usually takes less time to evaluate one generation of individuals, the question remains if the TPE algorithm possibly needs fewer generations to find good solutions. Figure 7 shows that this is not generally the case. Indeed, the development of the average best validation error over time is very similar between the two algorithms. Only for an input size of 200x200 pixels the TPE algorithm significantly outperforms the GA both in the quality of the solution and in the number of trials that is needed to find it. The better validation errors that are achieved by the TPE algorithm might be due to the much bigger number of used filters on the third, fourth and fifth convolutional layer. However, with a longer runtime for the GA or a different initialization of the number of filters on the convolutional layers it is likely that this drawback of the GA could be overcome.

For the random search algorithm, this comparison does not provide much information, as no knowledge besides the predefined search space is used in the algorithm. Each sample, therefore, has the same probability of being a "good" solution. Nevertheless, we observe that the random search algorithm often finds its best solution within the first 500 evaluations, which is not the case for both TPE and the genetic algorithm. However, even though the random search algorithm on average seems to reach its best solution in fewer evaluations, this solution is often worse than the solutions found by both the GA and TPE after 500 evaluations and significantly worse than the solutions found by the GA and TPE after all 1500 evaluations.

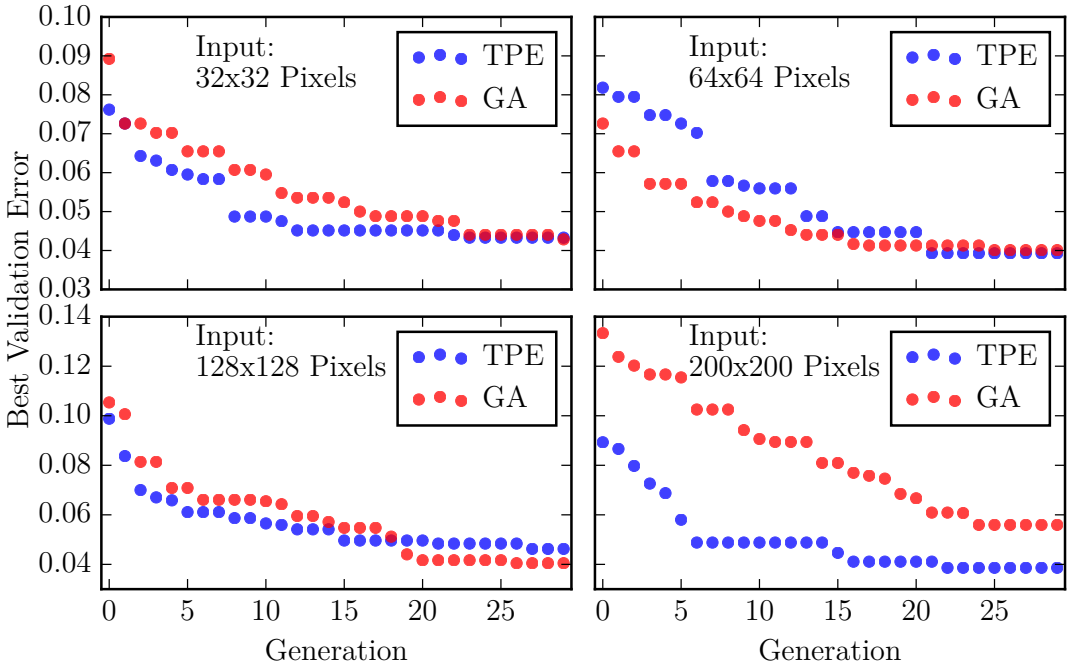

Figure 7: Best average validation error after each generation for the genetic algorithm (GA) and the TPE algorithm on the CK+ data set.

In general, we can say that both the GA and the TPE algorithm outperform random search when we look at the average validation error and no additional regularization (which is the setting for which the hyperparameters were optimized). While the TPE algorithm also outperforms the GA, the differences are, except for few exceptions, small. We can also see that all algorithms tend to use hyperparameters in a similar range of respective values. The only major difference is the number of used filters per convolutional layer. This difference is probably why the TPE algorithm takes so much longer than the GA and also why it outperforms the GA.

However, this difference in the number of filters per convolutional layer is likely not an inherent weakness of the GA, but simply a result of the initialization. Since in the GA all layers only have $10 - 50$ filters, in the beginning, it takes time for this number of filters to

grow, especially to a number as big as 700. It is, therefore, likely that a bigger number of filters would also be chosen by the GA if we either initialize it with a greater variance and higher numbers or if we simply let it run for longer. On the other hand, the high overlap in the general hyperparameter choices between the TPE algorithm and the GA indicates that these are indeed appropriate hyperparameters for the given problem.

We can also see that the different hyperparameters' importance is approximately consistent across the different input sizes. The only exception to that are hyperparameters that include the number of convolutional layers. Since the maximum number of convolutional layers is limited by the input size, CNNs with smaller inputs cannot have as many convolutional layers as CNNs with bigger inputs. Additionally, e.g. for an input size of 32x32 pixels, a CNN might still perform reasonably well with only one convolutional layer, even if two layers lead to an improvement of performance. For an input size of 96x96 pixels, or even 200x200 pixels, on the other hand, a CNN with only one convolutional layer does not perform well at all. This is an inherent problem of the optimization process across different input dimensions and most likely means that the optimal number of convolutional layers has to be found for each specific input size. A good starting point for the number of filters per convolutional layer, on the other hand, can be inferred from smaller input dimensions, at least for convolutional layers that are present in CNNs for smaller inputs.

5. Experiment 2: Complex Data

The previous experiment indicated that the GA is a promising approach to optimize the hyperparameters of CNNs. Additionally, it suggests that hyperparameters are approximately of the same importance, independent of the image resolution and consistency of good hyperparameter values across image resolutions. The second experiment aims to test these findings from the previous experiment on a new and independent data set, so as not to be biased by the hyperparameters that were found in the previous experiment. For this, we choose the 102 Flowers data set (Nilsback and Zisserman, 2008). While this data set offers images of very high resolution (minimum 500x500 pixels), many practitioners rescale the images to a size of 221x221 pixels (Sermanet et al., 2013; Sharif Razavian et al., 2014) to reduce the number of inputs and the amount of time needed to train the model. Therefore, we also use 221x221 pixels as our "maximum" input size, i.e. the input for which the hyperparameters should be optimized are RGB images of size 221x221 pixels. There are 8189 images all in all. Nilsback and Zisserman (2008) provide a predefined data split, which gives 2040 images for training and validation, while the remaining 6149 images are used as a test set. The 2040 training and validation images are further split into two equally sized groups, each of which contains 10 images of each flower category. We follow the protocol by Nilsback and Zisserman (2008) and use the first 1020 images as a training set to optimize the hyperparameters, evaluating their performance on the other 1020 images.

The traditional approach is to take the data as we have it and then run some algorithm to optimize the hyperparameters for the given model. This will be our benchmark and we will use the TPE algorithm to optimize the hyperparameters for a CNN trained on the color images of size 221x221 pixels, again with a maximum of 1500 evaluations. Our algorithm, on the other hand, will not always work with these "big" images. Instead, we rescale the images to sizes of 128x128 and 64x64 pixels.

Our pipeline for the optimization process is then the following: We use our GA to optimize the hyperparameters of a CNN that gets as input images of size 64x64 pixels for 10 generations. The hyperparameters obtained through this are then used to initialize the GA for the optimization process on the images of size 128x128 pixels for another 10 generations. Finally, these hyperparameters are used to initialize the GA to optimize the hyperparameters for images of size 221x221 pixels for the final 10 generations. With this strategy we expect to arrive at hyperparameters that are comparable in performance to those obtained by the TPE algorithm. However, we expect our approach to take less time since most of the optimization is carried out on lower-dimensional input.

5.1 Experimental Setup

Several researchers have achieved good results on the 102 Flowers data set using common CNNs with many layers and filters (Sharif Razavian et al., 2014; Azizpour et al., 2015). These CNNs are based on some of the winners of ImageNet Competitions of the last years. Due to hardware limitations, we are not able to use them directly, however, their hyperparameters can give us some suggestions for an initial search space. Since the 102 Flowers data set contains many more classes than the previous data sets we tested and the exemplary CNNs that achieve good results are very big, we change the initialization for the GA. We now describe the initializations for the GA optimization runs on the different input sizes. Subsequently, we give an overview of the search space that is used by the TPE algorithm. The initialization and handling of hyperparameters that are not mentioned explicitly are the same as in the previous experiment.

For the initial search space we use the hyperparameters of Sermanet et al. (2013), Sharif Razavian et al. (2014) and Azizpour et al. (2015) as coarse direction, but reduce the number of layers, filters and hidden units due to hardware constraints. The initializations that are made for higher input sizes for the GA are based on the results of the previous, smaller input size. The initialization and mutation parameters were then adapted to reduce the variance in the respective hyperparameters. The idea behind this is that the optimization process on the smaller data sizes should already choose "good" areas in the hyperparameter space which should then be examined in more detail. While we have a big search space for the initialization of the smallest input size, for higher input sizes we reduce the space from which different hyperparameters are initialized. Additionally, the relative change in the size of the hyperparameter values caused by mutations is usually also reduced when the input size increases. This forces the GA to focus on promising areas of the hyperparameter search space while ignoring less promising areas. For more details on the exact mutation parameters that are used for the different resolutions see Table 12 in Appendix D

Genetic Algorithm – Input Size: 64x64 Pixels – Generations: 10 To accommodate for the complex data set, the initialization of the number of filters per convolutional layer is increased to $50 - 200$ per convolutional layer. The number of hidden units per layer is also increased to $100 - 1000$, as is the batch size, which is initialized to be between $20 - 100$. If they are mutated the number of the convolutional or hidden layers is in- or decreased by one with a probability of 50% each. Table 12 gives an overview over all mutation parameters for the three different resolutions.

Genetic Algorithm – Input Size: 128x128 Pixels – Generations: 10 Resulting from the hyperparameters found by the first 10 generations on 64x64 pixels images we initialize the hyperparameters for the next 10 generations. The mutation parameters are also adapted to focus more on the current area in the hyperparameter space. This decreases the exploration process but increases the exploitation of already good hyperparameter values. The learning rate is initially set to 0.01 for all CNNs since this was identified as the best learning rate in the first 10 generations. The number of convolutional layers is set to $3-4$ and the number of hidden layers is set to one. Both these values are also determined by the results of the first 10 generations.

Looking at the number of filters per convolutional layer after the 10 first generations they are initialized to be $50-150$, $100-150$, $150-200$ and $200-250$ per convolutional layer respectively. The initial number of units in the hidden layer is set to $500-800$, to focus on the most promising area of the previous generations. The batch size is also reduced to be between $20-60$ and the L1 regularization penalty is set to zero in 80% of the cases and to a small value of 0.0001 in all other cases.

Genetic Algorithm – Input Size: 221x221 Pixels – Generations: 10 For the hyperparameter initialization on the final input size, we again use the results of the previous 10 generations on the input size of 128x128 pixels. The learning rate is initialized to a value between 0.005 and 0.05, reflecting on good values of the previous generations. The number of convolutional layers is initially set to five and the number of hidden layers is one. The number of filters per convolutional layer is $50-100$, $80-150$, $150-200$, $170-250$ and $200-300$, as illustrated by the previous generations. The number of hidden units is between $500-800$ as before. The L1 regularization penalty is set to zero for all CNNs and the batch size is further reduced to $30-50$.

TPE Algorithm – Input Size: 221x221 Pixels – Generations: 30 The search space for the TPE algorithm is based on the same literature. The learning rate is between $0.00001-0.1$, the number of convolutional layers is between $1-5$ and the number of hidden layers is between $1-3$. The number of filters is up to 100, 200, 300, 400 and 500 on each of the respective convolutional layers and the filter size is either 3x3, 5x5 or 7x7. The number of units per hidden layer is between $50-1000$ and the batch size is between $10-250$. The L1 and L2 regularization penalties are between $0.0-0.1$.

With these hyperparameter settings, we proceed as in the previous experiments. The GA gets 10 generations for each input size, while the TPE algorithm again gets 1500 evaluations to find an appropriate hyperparameter setting. Similarly, as for section 4, we perform each optimization run three times and the best five solutions of each run are kept for further analysis. Since the CNNs can become quite complex with the given hyperparameter values there is also the possibility that we run out of memory. If this happens, it is not possible to actually evaluate the performance of the CNN, since we cannot train it with the given hardware. Therefore, hyperparameter settings that cannot be evaluated, but instead throw a memory error are assigned the worst possible fitness of "1.0".

5.2 Results

We will now have a look at the hyperparameters that were found by the GA and the TPE algorithm. After this, we will compare the two optimization procedures, in general, to see how much time and how many evaluations it usually takes them to find good solutions. Finally, we will evaluate the best settings found by each algorithm on the test set. For a detailed overview over the found hyperparameters see Table 14 in Appendix D.

Comparison of Hyperparameters

When we look at the hyperparameter values of the three different input sizes on the GA, we see that the progression between them is quite smooth. While the number of hidden layers stays consistently at one for all input sizes, the number of convolutional layers increases from three for an input size of 64x64 pixels to five for the maximum input size. The different solutions have between $60 - 90$ filters in the first layer and $250 - 300$ filters in the top layer, independent of how many convolutional layers they have in total. The filter size also stays at 3x3 for all inputs. Even the number of units in the hidden layer stays mostly between $500 - 700$ for all settings.

The batch size decreases from $40 - 60$ to $20 - 30$ with an increasing input size, while the L1 regularization penalty is zero for all settings. For the L2 penalty, the values settle somewhere in the middle between the initial values for 64x64 pixels and the values found for 128x128 pixels. We also see that the average generation time increases drastically, from roughly 30 minutes per generation for an input of 64x64 pixels to almost 4.5 hours for an input of 221x221 pixels. The average validation error, on the other hand, decreases over time from 0.697 to 0.644.

Comparing the final hyperparameter values of the GA to those found by the TPE algorithm shows a very big overlap. The only major difference is the number of filters in the third and fourth convolutional layer. While the GA chooses around $200 - 250$ filters, the TPE algorithm only chooses $20 - 60$ filters on the third convolutional layer and either $60 - 90$ or $340 - 380$ filters on the fourth convolutional layer. All other hyperparameters are very similar to each other. Even the average best validation error is almost identical, with a value of 0.644 for the genetic algorithm and 0.647 for the TPE algorithm.

Comparison of Optimization Processes

Figure 8 shows how the average best validation error developed during the different optimization processes. For the TPE algorithm, we again partitioned the 1500 trials into blocks of 50, which then constituted one "generation". We then took the hitherto best validation value of the three individual runs and averaged over them. The top graph of Figure 8 shows this development over individual generations. The bottom graph shows the same information but here the x-axis depicts the amount of elapsed time until the respective results were obtained. The same information is also visualized for the results of the GA. Here, we depict the results of the hyperparameter optimization on an input size of 64x64, 128x128 and 221x221 pixels in intervals of ten generations each.

The GA improves its solution continuously until the last generations. While the progression from input sizes is quite smooth at the step from 64x64 to 128x128 pixels (generation 10), we see an obvious "bump" when we progress from 128x128 to 221x221 pixels. Here, the

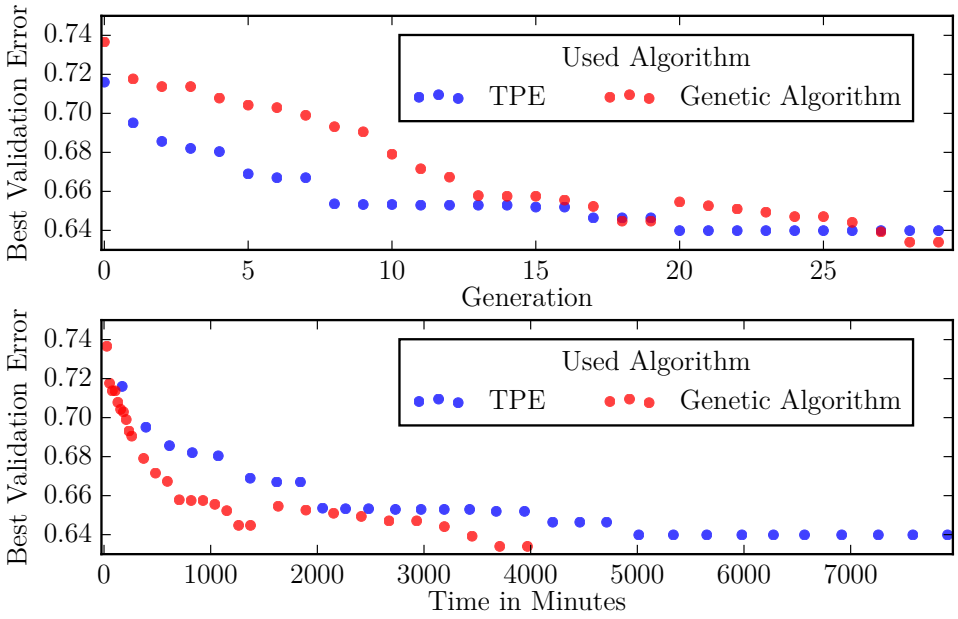

Figure 8: Best average validation error for the GA and the TPE algorithm on the 102 Flowers data set. The progress is visualized over the number of generations (top graph) and the amount of elapsed time in minutes (bottom graph)

average best validation error first increases but subsequently decreases below the previous best value. The TPE algorithm usually achieves its final best solution already after around 20 generations or approximately 5000 minutes. This means that it takes fewer evaluations than the GA to achieve a good solution.

However, we can also look at the progression of the validation error over time. This is arguably the more important aspect, as the required time is usually more important for a practitioner than the number of evaluations. Here, we can see that the GA achieves good solutions much quicker than the TPE algorithm. In fact, the GA achieves better average validation errors at any given time during the optimization. The main reason why it produces better solutions so much quicker is because it spends two-thirds of its evaluations on smaller inputs, which take less time to process. Indeed, as visualized in the bottom graph of Figure 8, the first twenty generations using the smaller input sizes take less time (about 1500 minutes) than the final ten generations on the biggest input size (about 2500 minutes). As a consequence, the GA used with smaller input sizes is able to make progress much faster, achieving good results in less time than the standard approach with the TPE algorithm.

Comparison of Test Set Performance

Finally, we test the hyperparameters found by the different algorithms on the test set of 6149 images. We follow the approach by Chai et al. (2011) and Yuan et al. (2012) and split the training set in the same manner as for the hyperparameter optimization. This way we obtain

1020 images for the training and validation set respectively, each with exactly 10 images per class. The test set is not balanced, with different classes containing between 20 and more than 200 examples. Following the standard approach (Nilsback and Zisserman, 2008) we report the average classification error averaged over all classes, not over all images on the test set. Table 4 gives an overview of the test errors achieved with different hyperparameter values and regularization methods.

Table 4: Test set errors of different hyperparameters on the 102 Flowers data set for input sizes of 221x221 pixels. The best results are highlighted in bold. See Table 13 in Appendix D for more details on the specific hyperparameter that were chosen for each *Setting*.

Found by	**Setting**	**Regularization methods**	**Val error**
Genetic Algorithm	Setting 1	none	0.678 ± 0.007
		dropout + bn	**0.587 ± 0.006**
	Setting 2	none	0.687 ± 0.006
		dropout + bn	**0.571 ± 0.006**
TPE	Setting 1	**none**	**0.670 ± 0.006**
		dropout	0.685 ± 0.008
	Setting 2	none	0.677 ± 0.005
		dropout + bn	**0.601 ± 0.003**

As we can see, all settings perform similarly well when no additional regularization methods are applied. The best solution found by the TPE algorithm significantly outperforms the solutions found by the GA, while the second best TPE solution significantly outperforms only the second best solution by the GA. However, once we apply common regularization techniques such as dropout (Srivastava et al., 2014) and batch normalization (Ioffe and Szegedy, 2015), the solutions found by the GA significantly outperform those from the TPE algorithm. It seems that the reduction in the number of filters especially on the third convolutional layer has a detrimental impact when regularization is applied to this layer. This can be seen in *Setting 1* and *Setting 2* for the TPE algorithm. While *Setting 1* only has 20 and 60 filters on the third and fourth convolutional layer, *Setting 2* has 80 and 250 filters respectively. When we apply batch normalization to those layers, we see that too few filters are detrimental to the performance. This effect is so strong that for *Setting 1* of the TPE algorithm, dropout on its own outperforms the combination of dropout and batch normalization. This is most likely because dropout is only applied to the hidden layer, whereas batch normalization is applied to all layers.

Discussion

In our second experiment, we showed that the GA works even for more complex data sets. The hyperparameters found by the GA are overall very similar to the ones found by the TPE algorithm. We could also illustrate the power of using smaller input sizes to find promising areas in the hyperparameter space, before optimizing the hyperparameters on the final input size. This approach was able to find comparative results in approximately

half the times as the TPE algorithm. This is a very promising result since this approach is independent of the underlying optimization algorithm. It can instead be applied to any optimization methodology, for which it is possible to reduce the dimensionality of the data in a meaningful way.

We can also see that the two optimization processes handle the given memory constraints differently. The TPE algorithm restricts the size of the CNNs by using only $20 - 60$ filters on the third convolutional layer. This "frees up" enough memory to use CNNs which have more than 400 filters on the fifth convolutional layer. The GA, on the other hand, does not create a bottleneck in one of the convolutional layers, but instead increases the number of filters roughly in intervals of 50 per convolutional layer, with around $250 - 300$ filters on the fifth convolutional layer.

However, the bottleneck that TPE introduces in the convolutional layers is not optimal if additional regularization methods, such as batch normalization, are used in the convolutional layers. The results of Table 4 clearly show that the GA's architectures perform much better if regularization methods are applied to the convolutional layers.

6. Conclusion

In this work, we present two novelties in the context of hyperparameter optimization. The first is a genetic algorithm (GA), which optimizes the hyperparameters specifically of convolutional neural networks (CNNs). Some knowledge about "common" CNN architectures is preserved by the GA: higher convolutional layers cannot have fewer filters than lower convolutional layers and higher hidden layers cannot have more units than lower hidden layers. Where it makes sense the search space is discretized, e.g. the learning rate is usually optimized on a logarithmic scale. Comparisons to Random Search and the Tree of Parzen Estimators (TPE) as other optimization algorithms show that the GA outperforms Random Search and usually performs on a par with the more sophisticated TPE algorithm. An additional advantage of the GA is that it is easily parallelizable since all hyperparameter settings of a given generation can be evaluated in parallel.

The second novelty is of a more general character. One of the main challenges of optimizing hyperparameters in CNNs is that the training process can take a very long time. This makes it expensive to evaluate many different hyperparameter combinations. To evade this problem we propose to first find promising hyperparameter values on a lower dimensional representation of the original data. To test this, we rescale images to a lower resolution and optimize the hyperparameters of CNNs on those smaller images. The results of this are then used to initialize the hyperparameter space of the optimization process on the original-sized images. In theory, this process can be repeated several times, i.e. the hyperparameters can be optimized on multiple smaller representations that increase in size until we reach the original data size. We investigate this approach on a more theoretical level in the first experiment. Here, we optimize the hyperparameters independently on the same images but with different resolutions. We observe that a significant number of hyperparameters overlap in their values, independent of the image resolution. Additionally, we find that the importance of different hyperparameter subsets in relation to each other stays roughly the same across different resolutions. This can be used to quickly identify important hyperparameters on lower-dimensional data.

These results are tested on a new data set in the second experiment and we show that an approach using this concept finds good hyperparameters roughly two times faster than the original approach employing the TPE algorithm. This concept is generally applicable to many different hyperparameter optimization methodologies and it is not restricted to the GA. Moreover, it is not only applicable to images but to any input whose dimensions can be reduced in a meaningful way, e.g. through common dimensionality reduction algorithms like PCA. However, while we show that it works well on CNNs in conjunction with image classification, future work needs to test if this is also the case for other tasks and other ways of dimensionality reduction. Furthermore, this technique is easily extensible with other methods for speeding up the hyperparameter optimization process, such as extrapolating learning curves (Domhan et al., 2015) or using weight features for performance predictions (Yamada and Morimura, 2016)

Even though the GA already finds good solutions in a reasonable amount of time there are still areas that we have not analyzed yet. One interesting question is whether different fitness functions affect the algorithm's performance. In our work, the fitness is solely based on the validation error which is to be minimized. However, many other fitness functions are conceivable, which might take into account other factors such as the network's complexity or the training time. We also did not do much optimization related to the hyperparameters guiding the GA, for which we used the default values by Navarro-Guerrero et al. (2017). Mutation rates are mostly based on intuition and experience and the crossover operation only uses a single crossover point.

While the GA as such requires the practitioner to specify more hyperparameters than for example random search or the TPE algorithm, this is not necessarily a drawback. Both random search and TPE require a specification of the search space. If the search space is too small, none of the algorithms might find an appropriate solution. It is, therefore, advisable to have a big search space, which leads to an increase in runtime. The GA, on the other hand, gets an initial search space through the hyperparameter initialization of the first generation's population. Through mutations, individuals of subsequent generations can leave this "search space" if necessary. Additionally, the practitioner can somewhat guide the search with his choice of the GA's hyperparameters. If the practitioner is quite certain about what hyperparameter values constitute good ones, the search space can be small from the very start and the scale of the mutations can also be kept small. If he is unsure, however, the first generation's hyperparameters can be initialized with a great variance and mutations can change hyperparameter values drastically. As we showed, the scale of the mutations can be reduced during the optimization process, leading from an exploration phase in the first generations to more of an exploitation in later generations.

We showed that it is possible to achieve very good results even with only basic mutation and crossover operations and "default" values for the GA's hyperparameters. Further improvements of the GA might lead to both better results and shorter runtimes. The idea of using smaller sized inputs in the beginning to find good hyperparameter spaces before proceeding on the original input is a promising approach for speeding up the optimization process in general. In its most basic form, this is independent of the used algorithm and the data. Combining both methods is a promising concept to improve the process of automatic hyperparameter optimization for various inputs and different forms of artificial neural networks.

Acknowledgments

The authors gratefully acknowledge partial support from the German Research Foundation DFG under project CML (TRR 169), the European Union under project SECURE (No 642667), and the Hamburg Landesforschungsförderungsprojekt CROSS.

Appendix A. Definition of the Search Space for Random Search and TPE

Here, we provide information about the search space of both random search and the TPE algorithm in the first experiment (section 4). Similar to the genetic algorithm both algorithms were given 1500 evaluations.

Random Search For the implementation of the random search algorithm, we use the Hyperopt library (Bergstra et al., 2013a) to randomly sample hyperparameters from a previously defined search space. The learning rate is sampled as a float value between 0.1 and 0.00001 and the number of hidden layers is between 1 and 3, each with 50 to 500 units sampled as multiples of 50. A max-pooling layer is inserted after each convolutional layer and the maximal number of convolutional layers, therefore, depends on the input size. It is always between 1 and max_i with i representing the input size and max_i being the maximum possible number of convolutional layers for that input size. Hence, e.g. the number of convolutional layers for an input of 32x32 pixels is sampled as an integer between 1 and 2, while max_i for an input of 200x200 pixels is 5.

To decide on an appropriate search space to sample the number of filters per convolutional layers from we look at the approach by Khorrami et al. (2015). This work delivers state-of-art results on the CK+ data set and is, therefore, a good point of reference. They use a CNN with three convolutional layers, with 64, 128 and 256 filters per layer respectively to classify images of size 96x96 pixels. Since they do not elaborate at how they arrived at these values we double the number and use that as a guideline for the maximum number of filters per convolutional layer to sample from. Similar to the GA we sample the number of filters per convolutional layer as multiples of 10. The first convolutional layer has therefore between 10 and 150 filters and the second convolutional layer, if existent, has between 10 and 260 filters. Due to computational limitations the third convolutional layer, if existent, only has a maximum of 400 filters.

For input sizes of 128x128 pixels and bigger we have to reduce the number of possible filters due to memory limitations. Thus, the maximum number of possible filters in the first convolutional layer is reduced to 100 for input size 128x128 pixels, while a potential fourth convolutional layer could have a maximum of 600 filters. For an input size of 200x200 pixels, the maximum number of filters per convolutional layer is 50, 100, 200, 300 and 400 respectively.

For the STL-10 data set, the search space is changed tom some extent. Since the input sizes differ from the CK+ data set images we have now a maximum of two, three and four convolutional filters for the input sizes of 32x32, 48x48 and 96x96 pixels respectively. Most of the work on the STL-10 data set does not work with raw pixels as input, but instead with samples features with some sort of k-means algorithm, which are then fed to a CNN.

Hence, there are few guidelines for the hyperparameters that should be used for the STL-10 data set with a vanilla CNN approach. However, compared to the CK+ data set

the STL-10 data set has 10 different classes with very different objects in them. In the CK+ data set all images contain centered faces, reducing the number of necessary filters, whereas this is not the case for the STL-10 data set. Due to this, the number of possible filters per layer was increased to 200, 400, 600 and 800 filters respectively per convolutional layer. The search space for the learning rate, the number of hidden layers and the number of units per hidden layer is the same as for the CK+ data set.

For the CK+ data set the batch size is drawn from {10, 20, 30, 40, 50}. Since the STL-10 data set possesses more classes than the CK+ data set its batch size is drawn from a slightly bigger search space, i.e. {10, 20, ..., 70, 80}. The filter size on each convolutional layer can be either 3x3, 5x5 or 7x7 for each data set. The L1 and L2 penalties are sampled from [0.1, 1e-7].

TPE The "Tree of Parzen Estimators" (TPE) algorithm (Bergstra et al., 2011) is a sequential model-based optimization (SMBO) strategy based on Bayesian probabilities. We use the implementation of TPE provided by Hyperopt (Bergstra et al., 2013a, 2014). It takes as input a specification of the initial search space for the hyperparameters, a loss function, which maps specific hyperparameters to a real value and an experimental history of values of the loss function (Bergstra et al., 2013b). The algorithm then returns a suggestion for which hyperparameter setting should be evaluated next, given the current search history. The next hyperparameter setting is then evaluated using the loss function, added to the search history and the next suggestion is calculated based on the updated history. Again, the five best performing hyperparameter settings are kept for further analysis and comparison. The definition of the search space is the same as for the random search algorithm.

Appendix B. List of best Hyperparameters per Algorithm and Data Set

Here we provide a detailed overview of the hyperparameters that were found by the various optimization algorithms for the different data sets in the first experiment (section 4). Tables 5, 6 and 7 describe the hyperparameters found for the CK+ data set, while Tables 8, 9 and 10 describe the hyperparameters for the STL-10 data set and Table 11 describes the hyperparameters used during the evaluation on the test sets.

Table 5: Results of the best CNNs found by the **genetic algorithm** for the extended **Cohn-Kanade** data set. The table shows results averaged over three independent optimization runs on each resolution. For each of the four resolutions the genetic algorithm was initialized randomly from the same search space. The crossover and mutation parameters stay the same across all resolutions.

input size	32x32px	64x64px
avg generation time	5 min 06 sec	7 min 33 sec
avg validation error	0.048 ± 0.007	0.043 ± 0.011
learning rate	0.01 (100%)	0.1 (13%), 0.01 (87%)
batch size	10 (100%)	10 (80%), 50 (13%)
L1 regularization	0.1 (27%), 0.01 − 0.001 (47%)	0.01 − 0.1 (33%), 0.0001 (53%)
L2 regularization	0 (60%), 0.01 − 0.1 (20%)	0 (13%) 0.01 − 0.1 (87%)
number of conv layers	1 (73%), 2 (27%)	2 (100%)
number of hidden layers	1 (100%)	1 (100%)
number of filters		
1st conv layer	20 − 30 (47%), 50 − 80 (53%)	10 (100%)
2nd conv layer		20 − 40 (100%)
filter size		
1st conv layer	5 (80%), 9 (20%)	5 (100%)
2nd conv layer		3 (67%), 5 (33%)
units in 1st hidden layer	200 − 300 (47%), 400 − 450 (47%)	200 − 250 (67%), 350 − 400 (33%)
input size	128x128px	200x200px
avg generation time	14 min 05 sec	20 min 44 sec
avg validation error	0.047 ± 0.011	0.058 ± 0.013
learning rate	0.01 (100%)	0.01 (100%)
batch size	10 (100%)	10 (73%), 40 (13%)
L1 regularization	0.001 (20%), 0.01 − 0.1 (73%)	0 − 0.001 (60%), 0.01 − 0.1 (33%)
L2 regularization	0 (60%), 0.1 (40%)	0.001 (33%), 0.01 − 0.1 (60%)
number of conv layers	3 (40%), 4 (60%)	4 (67%), 5 (33%)
number of hidden layers	1 (100%)	1 (87%)
number of filters		
1st conv layer	10 (67%), 60 (33%)	10 − 20 (93%)
2nd conv layer	30 − 50 (67%), 90 (33%)	10 − 30 (87%)
3rd conv layer	60 − 80 (67%), 90 − 100 (33%)	40 − 70 (87%)
4th conv layer	100 (100%)	50 − 70 (47%), 80 − 100 (53%)
filter size		
1st conv layer	5 (67%), 9 (33%)	5 (67%), 7 (33%)
2nd conv layer	3 (67%), 5 (27%)	3 (67%), 5 (33%)
3rd conv layer	3 (93%)	3 (100%)
4th conv layer	3 (100%)	3 (100%)
units in 1st hidden layer	200 − 300 (27%), 350 − 450 (73%)	300 − 350 (27%), 400 − 500 (60%)

Table 6: Results of the best CNNs found by **random search** for the extended **Cohn-Kanade** data set. The table shows results averaged over three independent optimization runs on each resolutions. For each optimization run of the four resolutions, random search sampled 1500 hyperparameter settings from the search space and evaluated their performance.

input size	32x32px	64x64px
avg generation time	5 min 26 sec	11 min 42 sec
avg validation error	0.064 ± 0.007	0.078 ± 0.005
learning rate	0.008 – 0.08	0.001 – 0.04
batch size	10 – 20 (53%), 30 – 40 (47%)	10 – 20 (53%), 30 – 40 (40%)
L1 regularization	0.0 (47%), 0.001 – 0.01 (33%)	0.0 (60%), 0.002 – 0.02 (33%)
L2 regularization	0.0 (33%), 0.01 – 0.1 (40%)	0.0 (33%), 0.001 – 0.05 (47%)
# conv layers	1 (80%), 2 (20%)	2 (27%), 3 (73%)
# hidden layers	1 (67%), 2 (20%)	1 (80%), 2 (20%)
number of filters		
1st conv layer	30 – 80 (60%), 90 – 120 (27%)	30 – 70 (60%), 100 – 120 (27%)
2nd conv layer		20 – 60 (40%), 70 – 120 (33%)
3rd conv layer		100 – 200 (45%), 210 – 300 (36%)
filter size		
1st conv layer	5 (33%), 7 (60%)	5 (47%), 7 (40%)
2nd conv layer		3 (20%), 5 (47%), 7 (33%)
3rd conv layer		3 (27%), 5 (18%), 7 (55%)
units in 1st hidden layer	100 – 300 (33%), 400 – 500 (47%)	100 – 300 (60%), 400 – 500 (40%)

input size	128x128px	200x200px
avg generation time	37 min 13 sec	42 min 42 sec
avg validation error	0.07 ± 0.007	0.07 ± 0.01
learning rate	0.003 – 0.05	0.003 – 0.06
batch size	10 – 20 (40%), 30 – 40 (47%)	10 – 20 (40%), 30 – 40 (47%)
L1 regularization	0.0 (40%), 0.0001 – 0.01 (47%)	0.0 (53%), 0.001 – 0.01 (33%)
L2 regularization	0.01 – 0.1 (40%) 0.0001 – 0.009 (40%)	0.001 – 0.01 (27%) 0.01 – 0.1 (47%)
# conv layers	3 (73%), 4 (27%)	4 (73%), 5 (27%)
# hidden layers	1 (73%), 2 (27%)	1 (80%), 2 (20%)
number of filters		
1st conv layer	20 – 80 (87%)	10 – 50 (100%)
2nd conv layer	100 – 150 (47%), 170 – 230 (47%)	20 – 70 (33%), 80 – 130 (53%)
3rd conv layer	200 – 270 (40%), 300 – 360 (33%)	50 – 100 (27%), 120 – 200 (60%)
4th conv layer		150 – 300 (73%)
filter size		
1st conv layer	3 (27%), 5 (27%), 7 (46%)	5 (20%), 7 (73%)
2nd conv layer	3 (47%), 5 (40%), 7 (13%)	5 (47%), 7 (40%)
3rd conv layer	3 (33%), 5 (40%), 7 (17%)	3 (20%), 5 (53%), 7 (27%)
4th conv layer		3 (20%), 5 (33%), 7 (47%)
units in 1st hidden layer	100 – 250 (80%)	100 – 250 (53%), 300 – 450 (27%)

Table 7: Results of the best CNNs found by the **TPE algorithm** for the extended **Cohn-Kanade** data set. The table shows results averaged over three independent optimization runs on each resolution. For each of the four resolutions, the TPE algorithm employed the same search space. Similar to random search the TPE algorithm got 1500 evaluations per optimization run and resolution.

input size	32x32px	64x64px
avg generation time	7 min 03 sec	12 min 10 sec
avg validation error	0.046 ± 0.007	0.046 ± 0.01
learning rate	0.006 – 0.009	0.01 – 0.05
batch size	10 (60%), 20 (20%)	10 (47%), 40 (33%)
L1 regularization	0.0 (47%), 0.003 – 0.03 (27%)	0.0 (47%), 0.0001 – 0.0005 (40%)
L2 regularization	0.005 – 0.05 (80%)	0.0 (40%), 0.02 – 0.06 (33%)
number of conv layers	1 (67%), 2 (33%)	2 (67%), 3 (33%)
number of hidden layers	1 (67%), 2 (33%)	1 (93%)
number of filters		
1st conv layer	30 – 60 (53%), 100 – 130 (40%)	20 – 40 (93%)
2nd conv layer	150 – 180 (100%)	100 – 150 (47%), 160 – 180 (33%)
filter size		
1st conv layer	5 (47%), 7 (53%)	5 (87%), 7 (13%)
2nd conv layer	5 (100%)	5 (53%), 7 (27%)
units in 1st hidden layer	250 – 350 (60%), 400 – 500 (27%)	100 – 200 (47%), 300 – 350 (40%)
input size	128x128px	200x200px
avg generation time	51 min 18 sec	59 min 07 sec
avg validation error	0.051 ± 0.008	0.041 ± 0.01
learning rate	0.007 – 0.04	0.01 – 0.05
batch size	30 (47%), 40 (33%)	10 (33%), 30 (60%)
L1 regularization	0.0 (53%), 0.002 – 0.02 (40%)	0.0 (60%), 0.01 – 0.1 (33%)
L2 regularization	0.0 (27%), 0.02 – 0.09 (53%)	0.0 (27%), 0.01 – 0.1 (53%)
number of conv layers	3 (40%), 4 (60%)	4 (67%), 5 (33%)
number of hidden layers	1 (100%)	1 (80%)
number of filters		
1st conv layer	10 – 20 (100%)	10 – 20 (53%), 30 – 50 (47%)
2nd conv layer	10 – 30 (47%), 40 – 70 (40%)	110 – 150 (87%)
3rd conv layer	120 – 170 (67%), 200 – 260 (27%)	80 – 120 (67%), 200 – 240 (27%)
4th conv layer	470 – 570 (56%)	200 – 270 (27%), 330 – 390 (60%)
filter size		
1st conv layer	5 (60%), 7 (33%)	5 (67%), 7 (33%)
2nd conv layer	3 (27%), 7 (53%)	3 (53%), 5 (33%)
3rd conv layer	3 (53%), 7 (33%)	3 (33%), 7 (40%)
4th conv layer	3 (44%), 7 (33%)	5 (33%), 7 (40%)
units in 1st hidden layer	100 – 200 (53%), 250 – 300 (33%)	50 – 150 (47%), 400 – 500 (47%)

Table 8: Results of the best CNNs found by the **genetic algorithm** for the **STL-10** data set. The table shows results averaged over three independent optimization runs on each resolution. For each of the three resolutions, the genetic algorithm was initialized randomly from the same search space. The crossover and mutation parameters stay the same across all resolutions.

input size	32x32px	48x48px	96x96px
avg generation time	13 min 07 sec	17 min 56 sec	29 min 56 sec
avg validation error	0.580 ± 0.001	0.579 ± 0.001	0.539 ± 0.002
learning rate	0.001 (100%)	0.001 (100%)	0.001 (100%)
batch size	20 (20%), 30 (20%) 40 (27%), 50 (33%)	20 (53%), 30 (13%) 40 (13%), 50 (13%)	10 (20%), 30 (27%) 40 (27%), 50 (20%)
L1 regularization	0.0 (47%) 0.001 – 0.01 (47%)	0.001 – 0.01 (47%) 0.1 (47%)	0.0 (67%) 0.01 (33%)
L2 regularization	0.0 (33%) 0.1 (67%)	0.001 – 0.01 (53%) 0.1 (40%)	0.0 (33%) 0.01 – 0.1 (53%)
# conv layers	2 (100%)	3 (100%)	4 (100%)
# hidden layers	2 (100%)	2 (100%)	2 (100%)
# filters			
1st conv layer	40 (93%)	20 – 30 (67%) 40 – 50 (33%)	40 – 50 (100%)
2nd conv layer	50 (93%)	50 – 60 (73%) 70 – 80 (27%)	30 (33%) 60 – 70 (67%)
3rd conv layer		60 – 70 (47%)	60 – 80 (80%)
4th conv layer		80 – 90 (53%)	100 (80%)
filter size			
1st conv layer	3 (100%)	3 (100%)	3 (67%), 5 (33%)
2nd conv layer	3 (100%)	3 (100%)	3 (100%)
3rd conv layer		3 (100%)	3 (100%)
4th conv layer			3 (100%)
# hidden units			
1st hidden layer	350 – 400 (47%) 450 – 500 (53%)	400 – 450 (47%) 500 – 600 (47%)	300 – 350 (87%)
2nd hidden layer	100 (33%), 200 (33%) 300 (33%)	100 – 200 (53%) 450 – 500 (47%)	200 (60%) 350 (33%)

Table 9: Results of the best CNNs found by **random search** for the **STL-10** data set. The table shows results averaged over three independent optimization runs on each resolutions. For each optimization run of the three resolutions, 1500 hyperparameter settings were randomly sampled from the search space and their performance was evaluated.

input size	32x32px	48x48px	96x96px
avg generation time	13 min 22 sec	27 min 54 sec	1h 07 min 42 sec
avg validation error	0.594 ± 0.005	0.58 ± 0.013	0.573 ± 0.008
learning rate	$0.0001 - 0.003$	$0.0002 - 0.001$	$0.0001 - 0.001$
batch size	10 (20%), 20 (20%) 40 (27%), 70 (20%)	10 (20%), 20 (33%) 30 (33%)	10 (30%), 20 (20%) 50 (40%)
L1 regularization	0.0 (47%) $0.001 - 0.01$ (40%)	0.0 (33%) $0.001 - 0.05$ (40%)	0.0 (40%) $0.001 - 0.05$ (50%)
L2 regularization	0.0 (53%) $0.01 - 0.02$ (33%)	$0.0001 - 0.001$ (40%) $0.01 - 0.1$ (27%)	0.0 (50%) $0.001 - 0.06$ (50%)
# conv layers	2 (100%)	3 (93%)	3 (20%), 4 (80%)
# hidden layers	2 (60%), 3 (40%)	2 (80%), 3 (20%)	2 (70%), 3 (20%)
# filters			
1st conv layer	$50 - 70$ (33%) $80 - 110$ (40%)	$30 - 50$ (40%) $140 - 160$ (33%)	$50 - 80$ (40%) $100 - 150$ (50%)
2nd conv layer	$100 - 190$ (27%) $200 - 300$ (53%)	$100 - 200$ (33%) $210 - 300$ (40%)	$150 - 200$ (60%) $210 - 250$ (20%)
3rd conv layer		$200 - 300$ (33%)	$300 - 500$ (60%)
4th conv layer		$400 - 600$ (42%)	$500 - 800$ (75%)
filter size			
1st conv layer	3 (80%), 5 (20%)	3 (80%), 5 (13%)	3 (40%), 5 (40%)
2nd conv layer	3 (67%), 5 (33%)	3 (80%), 7 (14%)	3 (40%), 5 (40%)
3rd conv layer		3 (57%), 5 (21%)	5 (60%), 7 (20%)
4th conv layer			3 (63%), 5 (25%)
# hidden units			
1st hidden layer	$250 - 350$ (40%) $400 - 500$ (53%)	$250 - 350$ (40%) $400 - 450$ (47%)	$100 - 250$ (50%) $350 - 450$ (50%)
2nd hidden layer	$100 - 150$ (20%) $200 - 350$ (67%)	$200 - 300$ (33%) $350 - 450$ (53%)	$150 - 250$ (67%) $400 - 500$ (22%)

Table 10: Results of the best CNNs found by the **TPE algorithm** for the **STL-10** data set. The table shows results averaged over three independent optimization runs on each resolution. For each of the three resolutions, the TPE algorithm employed the same search space. Similar to random search the TPE algorithm got 1500 evaluations per optimization run and resolution.

input size	32x32px	48x48px	96x96px
avg generation time	18 min 52 sec	31 min 33 sec	2h 41 min 33 sec
avg validation error	0.577 ± 0.002	0.553 ± 0.004	0.533 ± 0.003
learning rate	0.0003 – 0.0005	0.0002 – 0.0008	0.0003 – 0.0008
batch size	10 (20%), 20 (80%)	10 (100%)	40 (100%)
L1 regularization	0.0 (100%)	0.0004 – 0.0009 (80%)	0.0 (100%)
L2 regularization	0.09 – 0.13 (80%)	0.04 – 0.1 (100%)	0.001 – 0.009 (100%)
# conv layers	2 (100%)	3 (100%)	4 (100%)
# hidden layers	3 (100%)	2 (100%)	2 (100%)
# filters			
1st conv layer	150 – 200 (100%)	90 – 110 (100%)	180 – 200 (100%)
2nd conv layer	340 – 390 (100%)	190 – 210 (80%)	50 – 90 (40%) 140 – 150 (60%)
3rd conv layer		400 – 420 (60%)	360 – 450 (80%)
4th conv layer			570 – 690 (100%)
filter size			
1st conv layer	3 (100%)	3 (100%)	3 (100%)
2nd conv layer	3 (100%)	3 (100%)	3 (100%)
3rd conv layer		3 (80%)	3 (100%)
4th conv layer			5 (60%), 7 (40%)
# hidden units			
1st hidden layer	450 – 500 (100%)	450 – 500 (100%)	450 – 500 (80%)
2nd hidden layer	200 – 250 (80%)	400 – 450 (100%)	450 – 500 (80%)

Table 11: Results of **10-fold cross-validation** on the extended **Cohn-Kanade** and the **STL-10** data set. Each setting was tested both without any additional regularization and with regularization methods such as dropout and batch normalization (bn). The best results are highlighted in bold.

Data set	Found by	Hyperparameters	Regularization	Val error

Extended Cohn Kanade	Genetic Algorithm	learning rate = 0.01, batch size = 10
		# conv layers = 3, filter size = [5,3,3]
		filters = [10,40,60]
		# hidden layers = 1, units = [350]
		L1 = 0.0001, L2 = 0.1
			none	0.057 ± 0.005

			dropout	**0.046 ± 0.003**
			dropout max dropout	0.716 ± 0.011

		learning rate = 0.01, batch size = 10
		# conv layers = 4, filter size = [5,3,3,3]
		filters = [10,40,80,100]
		# hidden layers = 1, units = [300]
		L1 = 0.01, L2 = 0.1
			none	0.057 ± 0.005

			dropout	**0.046 ± 0.002**
			dropout max dropout	0.627 ± 0.065

	Random Search	learning rate = 0.018, batch size = 20
		# conv layers = 3, filter size = [7,5,3]
		filters = [50,200,320]
		# hidden layers = 1, units = [500]
		L1 = 0.0, L2 = 0.1
			none	0.077 ± 0.041

			dropout	**0.043 ± 0.004**
			dropout max dropout	0.813 ± 0.000

	TPE	learning rate = 0.029, batch size = 30
		# conv layers = 3, filter size = [7,7,3]
		filters = [70,150,340]
		# hidden layers = 1, units = [150]
		L1 = 0.013, L2 = 0.02
			none	0.062 ± 0.001

			dropout	0.066 ± 0.007
			dropout **bn**	**0.033 ± 0.001**

		learning rate = 0.032, batch size = 30
		# conv layers = 4, filter size = [5,7,3,3]
		filters = [10,140,220,470]
		# hidden layers = 1, units = [200]
		L1 = 0.0, L2 = 0.0001
			none	0.079 ± 0.002

			dropout	**0.053 ± 0.003**
			dropout bn	0.06 ± 0.003

STL-10	Genetic Algorithm	learning rate = 0.001, batch size = 40
		# conv layers = 4, filter size = [5,3,3,3]
		filters = [50,70,100,140]
		# hidden layers = 2, units = [350,350]
		L1 = 0.0, L2 = 0.01
			none	0.544 ± 0.001

			dropout	0.527 ± 0.001
			dropout + bn	0.862 ± 0.003
			dropout + **data augm**	**0.467 ± 0.003**

	Random Search	learning rate = 0.00087, batch size = 50
		# conv layers = 4, filter size = [7,5,5,5]
		filters = [110,170,400,330]
		# hidden layers = 2, units = [250,450]
		L1 = 0.0, L2 = 0.006
			none	0.569 ± 0.001

			dropout	0.556 ± 0.015
			dropout + bn	0.579 ± 0.001
			dropout + **data augm**	**0.482 ± 0.001**

	TPE	learning rate = 0.00087, batch size = 40
		# conv layers = 4, filter size = [5,3,3,3]
		filters = [200,140,380,690]
		# hidden layers = 2, units = [400,450]
		L1 = 0.0, L2 = 0.0155
			none	0.529 ± 0.001

			dropout	0.579 ± 0.049
			dropout + bn	0.541 ± 0.012
			data **augmentation**	**0.473 ± 0.001**

Appendix C. Meta-Analysis of GA, TPE, and Hyperparameters

Here, we present visualizations of the impact of various hyperparameters on the STL-10 data set, similar to the results shown in subsection 4.2 for the CK+ data set. Figure 9 presents an overview of the progress of the optimization procedures, similar to Figure 8 in section 5. Figures 10, 11 and 12 show the importance of various subsets of hyperparameters, the impact of the learning rate and the impact of the learning rate together with the number of hidden layers respectively for the STL-10 data set.

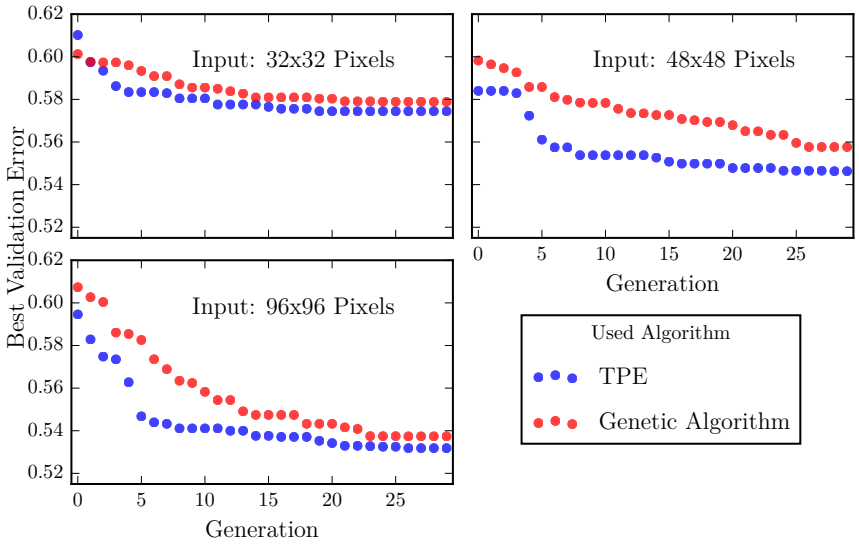

Figure 9: Best average validation error after each generation for the genetic algorithm and the TPE algorithm on the STL-10 data set over a total of 30 generations.

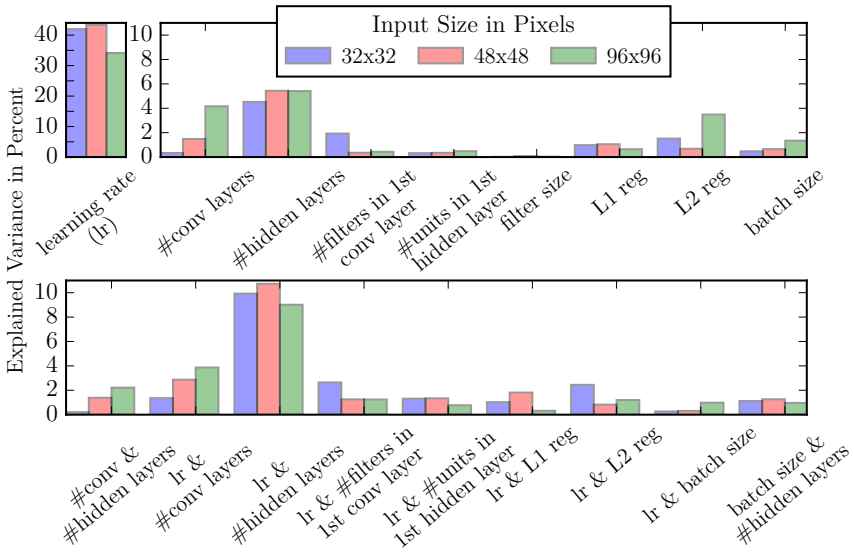

Figure 10: Explained variance of the validation error in percent on the STL-10 data set.

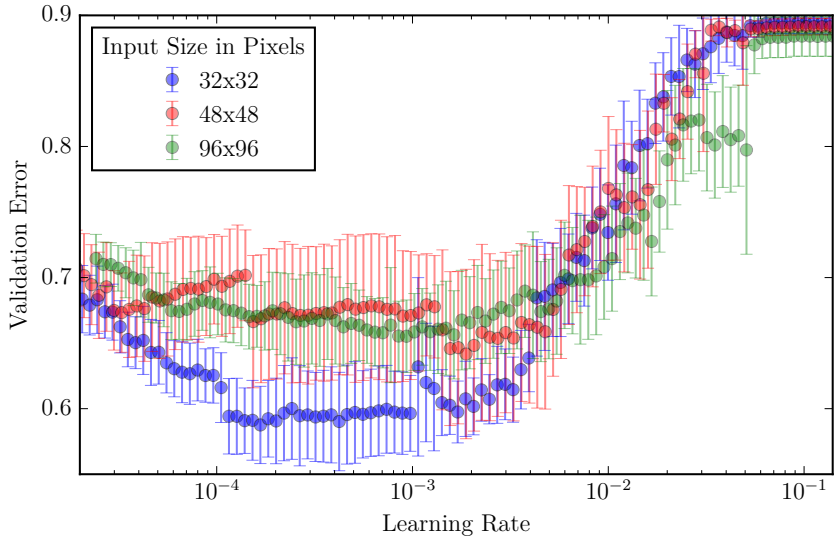

Figure 11: Merging the results of all three optimization algorithms shows the relationship between the learning rate and the validation error for different image resolutions on the STL-10 data set. We can see that the best learning rate is similar for all resolutions, usually between 10^{-3} and 10^{-4}.

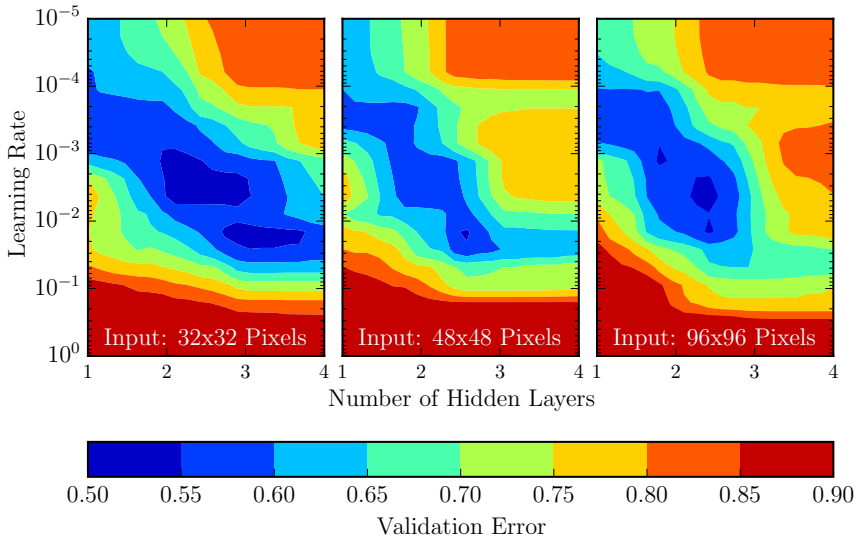

Figure 12: Merging the results of all three optimization algorithms shows the relationship between the learning rate, the number of hidden layers and the validation error for different image resolutions on the STL-10 data set.

Appendix D. Hyperparameters for the 102 Flower Data Set

Here, we provide more details to the experiment conducted in the second experiment (section 5). Table 12 describes the mutation parameters for the different resolutions that were used in the second experiment. Table 13 presents the hyperparameters that were used for the different *settings* to test the performance on the test set of the 102 Flowers data set. Table 14 presents the hyperparameters that were found by the genetic algorithm and the TPE algorithm for different resolutions on the 102 Flowers data set.

Table 12: The first horizontal block describes the mutation parameters for all hyperparameters for the smallest resolution of 64x64 pixels. The second and third horizontal blocks describe the mutation parameters that changed relative to the previous, smaller resolution. The different columns describe the probability with which different mutations are applied to any given hyperparameter.

64x64px	10%	20%	30%	40%	50%
learning rate L1 penalty L2 penalty	$\times 10^{2}$ / $\times 10^{-2}$		$\times 10$		$\times 10^{-1}$
# filters	$+30/+50$	± 10 / ± 20			
# hidden units	± 200 / ± 100		± 50		
filter size	$\pm(4 \times 4)$			$\pm(2 \times 2)$	
batch size	± 30 / ± 20		± 10		
128x128px	10%	20%	30%	40%	50%
learning rate		$\times 10$ / $\times 10^{-1}$	$\times 5$ / $\times 5^{-1}$		
# filters	± 10	± 20 / ± 30			
# hidden units	± 100			± 50	
221x221px	10%	20%	30%	40%	50%
learning rate		$\times 5$ / $\times 5^{-1}$	$\times 2$ / $\times 2^{-1}$		
batch size					± 10

Table 13: **Test set errors** on the **102 Flowers** data set for input sizes of 221x221 pixels. The best results are highlighted in bold. Each setting was tested both without any additional regularization and with regularization methods such as dropout and batch normalization (bn). The best results are highlighted in bold.

Found by	**Hyperparameters**	**Regularization**	**Val error**

Genetic Algorithm	learning rate = 0.005, batch size = 30
	# conv layers = 5, filter size = [3,3,3,3,3]
	filters = [80,150,220,280,300]
	# hidden layers = 1, units = [700]
	L1 = 0.0, L2 = 0.0001
		none	0.678 ± 0.007

		dropout	0.662 ± 0.008
		dropout **bn**	**0.587 ± 0.006**

	learning rate = 0.005, batch size = 20
	# conv layers = 5, filter size = [3,3,3,3,3]
	filters = [100,150,200,250,300]
	# hidden layers = 1, units = [700]
	L1 = 0.0, L2 = 0.0001
		none	0.687 ± 0.006

		dropout	0.668 ± 0.008
		dropout **bn**	**0.571 ± 0.006**

TPE	learning rate = 0.001, batch size = 40
	# conv layers = 5, filter size = [3,3,3,3,3]
	filters = [100,100,20,60,320]
	# hidden layers = 1, units = [550]
	L1 = 0.0002, L2 = 0.032
		none	**0.670 ± 0.006**

		dropout	0.685 ± 0.008
		dropout bn	0.706 ± 0.006

	learning rate = 0.0097, batch size = 70
	# conv layers = 5, filter size = [3,3,3,3,3]
	filters = [100,120,80,250,380]
	# hidden layers = 1, units = [700]
	L1 = 0.0005, L2 = 0.00014
		none	0.677 ± 0.005

		dropout	0.660 ± 0.006
		dropout **bn**	**0.601 ± 0.003**

Table 14: Results of the best CNNs found by the **genetic algorithm** and the **TPE algorithm** for the **102 Flowers** data set. The arrows between the input sizes for the genetic algorithm highlight that the search space of the genetic algorithm is initialized with the results of the genetic algorithm on the previous, smaller resolution. The procedure is described in more detail in section 5. The TPE algorithm optimizes the hyperparameters on the highest resolution for the full 1500 evaluations.

Algorithm	Genetic Algorithm			TPE
input size	64x64px →	128x128px →	221x221px	221x221px
# generations	10	10	10	30
avg gen time	29min 55sec	1h 48min 58sec	4h 29min 10sec	4h 10min 38sec
avg val error	0.697 ± 0.011	0.652 ± 0.008	0.644 ± 0.006	0.647 ± 0.005
learning rate	0.01 (100%)	0.01 (100%)	0.005 (100%)	0.001 − 0.01
batch size	40 (33%) 60 (47%)	30 (60%) 40 (27%)	20 (20%) 30 (67%)	10 (60%) 40 (20%)
L1 reg	0.0 (100%)	0.0 (100%)	0.0 (100%)	0.0 (47%) 0.01 − 0.09 (33%)
L2 reg	0.0 (33%) 0.01 − 0.1 (60%)	0.0001 (40%) 0.00001 (40%)	0.001 (27%) 0.0001 (60%)	0.0 (40%) 0.02 − 0.09 (40%)
# conv layers	3 (80%)	5 (100%)	5 (87%)	5 (100%)
# hidden layers	1 (100%)	1 (100%)	1 (100%)	1 (100%)
# filters				
1st conv layer	60 − 90 (47%) 120 − 150 (47%)	50 − 60 (33%) 90 (60%)	80 (40%) 100 (60%)	90 − 100 (93%)
2nd conv layer	140 − 150 (67%) 240 (33%)	80 − 120 (53%) 140 − 170 (47%)	110 − 120 (33%) 140 − 150 (67%)	100 − 110 (47%) 130 − 150 (47%)
3rd conv layer	150 − 170 (62%) 250 (33%)	130 − 160 (53%) 190 − 220 (33%)	150 (20%) 200 (80%)	20 − 30 (40%) 50 − 60 (40%)
4th conv layer		170 − 180 (33%) 230 − 240 (67%)	180 − 210 (40%) 250 (60%)	60 − 90 (33%) 340 − 380 (53%)
5th conv layer		190 (33%) 240 − 280 (67%)	270 − 290 (100%)	300 − 330 (33%) 430 − 460 (60%)
filter size				
1st conv layer	3 (80%)	3 (67%), 5 (33%)	3 (100%)	3 (93%)
2nd conv layer	3 (87%)	3 (100%)	3 (100%)	3 (87%)
3rd conv layer	3 (100%)	3 (100%)	3 (100%)	3 (67%), 5 (33%)
4th conv layer		3 (100%)	3 (100%)	3 (67%), 5 (33%)
5th conv layer		3 (100%)	3 (100%)	3 (93%)
# hidden units				
1st hidden layer	500 − 600 (33%) 700 − 800 (67%)	500 − 650 (67%) 750 − 900 (33%)	550 − 600 (33%) 650 − 700 (67%)	550 − 650 (53%) 700 − 800 (33%)

References

Saleh Albelwi and Ausif Mahmood. Automated Optimal Architecture of Deep Convolutional Neural Networks for Image Recognition. In *15th IEEE International Conference on Machine Learning and Applications*, pages 53–60, 2016.

Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. From Generic to Specific Deep Representations for Visual Recognition. In *Conference on Computer Vision and Pattern Recognition Workshops*, pages 36–45, 2015.

Rémi Bardenet, Mátyás Brendel, Balaázs Kégl, and Michéle Sebag. Collaborative Hyperparameter Tuning. In *International Conference on Machine Learning (2)*, volume 28, pages 199–207. ACM Press, 2013.

Yoshua Bengio. Practical Recommendations for Gradient-Based Training of Deep Architectures. In *Neural Networks: Tricks of the Trade*, number 7700, pages 437–478. Springer Berlin Heidelberg, 2012.

James Bergstra and Yoshua Bengio. Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research*, 13(1):281–305, February 2012. ISSN 1532-4435.

James Bergstra, Rmi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

James Bergstra, Dan Yamins, and David Cox. Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms. In *Python in Science Conference*, pages 13–20. Citeseer, 2013a.

James Bergstra, Daniel Yamins, and David Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning (1)*, pages 115–123, 2013b.

James Bergstra, Brent Komer, Chris Eliasmith, and David Warde-Farley. Preliminary Evaluation of Hyperopt Algorithms on HPOLib. In *International Conference on Machine Learning*, 2014.

Mátyás Brendel and Marc Schoenauer. Instance-based Parameter Tuning for Evolutionary AI Planning. In *Annual Conference Companion on Genetic and Evolutionary Computation*, pages 591–598. ACM, 2011.

Thomas M. Breuel. The Effects of Hyperparameters on SGD Training of Neural Networks. *arXiv:1508.02788*, August 2015.

Yuning Chai, Victor Lempitsky, and Andrew Zisserman. BiCoS: A Bi-Level Co-Segmentation Method for Image Classification. In *International Conference on Computer Vision*, pages 2579–2586, 2011.

Marion Chevalier, Nicolas Thome, Matthieu Cord, Jérôme Fournier, Gilles Henaff, and Elodie Dusch. LR-CNN for Fine-Grained Classification with Varying Resolution. In *International Conference on Image Processing*, pages 3101–3105, September 2015.

Adam Coates, Andrew Y Ng, and Honglak Lee. An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In *International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011.

David Cox and Nicolas Pinto. Beyond Simple Features: A Large-Scale Feature Search Approach to Unconstrained Face Recognition. In *IEEE International Conference on Automatic Face Gesture Recognition and Workshops*, pages 8–15, March 2011.

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

A Fiszelew, Paola Britos, Alejandra Ochoa, Hernán Merlino, Enrique Fernández, and Ramon García-Martínez. Finding Optimal Neural Network Architecture Using Genetic Algorithms. *Advances in Computer Science and Engineering Research in Computing Science*, 27:15–24, 2007.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

Giles Hooker. Generalized Functional ANOVA Diagnostics for High-Dimensional Functions of Dependent Variables. *Journal of Computational and Graphical Statistics*, 16(3):709–732, September 2007.

Frank Hutter, Holger Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An Efficient Approach for Assessing Hyperparameter Importance. In *International Conference on Machine Learning*, pages 754–762, 2014.

Frank Hutter, Jörg Lücke, and Lars Schmidt-Thieme. Beyond Manual Tuning of Hyperparameters. *Künstliche Intelligenz*, 29(4):329–337, July 2015.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning (ICML)*, volume 37, pages 448–456. JMLR: W&CP, 2015.

Pooya Khorrami, Thomas Paine, and Thomas Huang. Do Deep Neural Networks Learn Facial Action Units When Doing Expression Recognition? In *IEEE International Conference on Computer Vision*, pages 19–27, 2015.

Patrick Lucey, Jeffrey Cohn, Takeo Kanade, Jason Saragih, Zara Ambadar, and Iain Matthews. The Extended Cohn-Kanade Dataset (CK+): A complete dataset for action unit and emotion-specified expression. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 94–101, June 2010.

Nicolás Navarro-Guerrero. *Neurocomputational Mechanisms for Adaptive Self-Preservative Robot Behaviour*. PhD, Universität Hamburg, May 2016.

Nicolás Navarro-Guerrero, Robert Lowe, and Stefan Wermter. Improving Robot Motor Learning with Negatively Valenced Reinforcement Signals. *Frontiers in Neurorobotics*, 11 (10), 2017.

Maria-Elena Nilsback and Andrew Zisserman. Automated Flower Classification over a Large Number of Classes. In *Conference on Computer Vision, Graphics Image Processing*, pages 722–729, 2008. doi: 10.1109/ICVGIP.2008.47.

Nicolas Pinto, David Doukhan, James J. DiCarlo, and David D. Cox. A High-Throughput Screening Approach to Discovering Good Forms of Biologically Inspired Visual Representation. *PLOS Computational Biology*, 5(11):e1000579, November 2009.

Carl Edward Rasmussen. Gaussian Processes in Machine Learning. In *Advanced Lectures on Machine Learning*, number 3176 in Lecture Notes in Computer Science, pages 63–71. Springer Berlin Heidelberg, 2004.

Matthias Reif, Faisal Shafait, and Andreas Dengel. Meta-Learning for Evolutionary Parameter Optimization of Classifiers. *Machine Learning*, 87(3):357–380, April 2012.

Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *arXiv:1312.6229 [cs]*, December 2013.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN Features Off-the-Shelf: An Astounding Baseline for Recognition. In *Conference on Computer Vision and Pattern Recognition*, pages 806–813, 2014.

Sean C. Smithson, Guang Yang, Warren J. Gross, and Brett H. Meyer. Neural Networks Designing Neural Networks: Multi-objective Hyper-parameter Optimization. In *Proceedings of the 35th International Conference on Computer-Aided Design*, pages 104:1–104:8, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mostofa Ali, Ryan P Adams, et al. Scalable Bayesian Optimization using Deep Neural Networks. In *International Conference on Machine Learning*, 2015.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-Task Bayesian Optimization. In *Advances in Neural Information Processing Systems 26*, pages 2004–2012. Curran Associates, Inc., 2013.

Chris Thornton, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM, 2013.

Haibing Wu and Xiaodong Gu. Towards Dropout Training for Convolutional Neural Networks. *Neural Networks*, 71:1–10, 2015.

Yasunori Yamada and Tetsuro Morimura. Weight Features for Predicting Future Model Performance of Deep Neural Networks. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 2231–2237, 2016.

Steven R. Young, Derek C. Rose, Thomas P. Karnowski, Seung-Hwan Lim, and Robert M. Patton. Optimizing Deep Learning Hyper-parameters Through an Evolutionary Algorithm. In *Machine Learning in High-Performance Computing Environments*, pages 4:1–4:5. ACM, 2015.

Xiao-Tong Yuan, Xiaobai Liu, and Shuicheng Yan. Visual Classification with Multitask Joint Sparse Representation. *IEEE Transactions on Image Processing*, 21(10):4349–4360, 2012.

Liang Zheng, Yali Zhao, Shengjin Wang, Jingdong Wang, and Qi Tian. Good Practice in CNN Feature Transfer. *arXiv preprint arXiv:1604.00133*, 2016.

Barret Zoph and Quoc V Le. Neural Architecture Search with Reinforcement Learning. *arXiv preprint arXiv:1611.01578*, 2016.