Finding the right network architecture for image classification using genetic algorithms

Nathan Loudjani
Exchange student in School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
nloudjan@andrew.cmu.edu

Introduction

In recent years, the development of deep learning systems has been greatly simplified by the rising of libraries like *Tensorflow* and *Keras*. However, choosing the right architecture and hyper-parameters is still a very hard task for non-specialists.
In this project, we will use genetic algorithms to try to "evolve" to the right architecture for any given image classification task.
Given an image dataset and labels, we want to be able to generate a network with the best performances to categorize this dataset.
For this paper, we will evaluate our performances over multiple datasets. The first one will be the very well known *MNIST* dataset. This dataset is composed of 70000 28×28 digit images labeled with hot-end encoding. The second one is *Fashion-MNIST* [1], this dataset was designed as a replacement of *MNIST* because the original *MNIST* was too widely used and easy to solve with current technologies. This dataset is composed of 70000 28×28 gray-scale images of Zalando articles associated with a label from 10 classes. Finally, the third dataset will be the much more complex *Cifar10* dataset, which is now a reference in image recognition problems. This dataset is composed of 60000 32×32 color images labeled in 10 classes.
To measure the performances of our genetic algorithm, we will compare the results of our generated architecture with the state-of-the-art and most used existing convolutional and dense architectures.

1 Background

1.1 Genetic algorithms

Genetic algorithms are generative processes inspired by biology. They allow approaching an optimal solution in a space where exhaustive search would be impossible because of the high dimensionality. Those algorithms are usually composed of 3 phases:
During the initialization phase, a population of individuals is randomly generated. At this point, the program does not have any idea about what the desired solution looks like, so those individuals are often all very different.
The individuals are then compared during the evaluation phase. For each individual, the metrics to optimize are measured (in our case, the validation accuracy). The individuals are then ranked according to their performance on these metrics.
Finally, during the selection and evolution phase, the individuals of the next generation are created. A small number of individuals from the previous generation is first selected, usually the best-ranked ones. These individuals are then bred to create the new generation, a randomly selected part of this new population can also be mutated to create more variance.
Evaluation and selection phases can be repeated in a loop until a performance objective is reached or until the overall performance stops improving.

10417 Class Project (2019).

1.2 Existing solutions

Multiple recent experimentations tried to build a genetic algorithm to discover the good architecture for image classification. One of the main difficulties raised by this problem is the great dimensionality of the research space. Indeed, a great number of parameters must be used to describe a deep neural network, for example, the number of convolutional layers, the number of fully connected layers, the activation of each of them, the size and the number of each kernel, the dropout value, etc...
To overcome this difficulty, Yanan Sun and al. [2] approach is to use only convolutional networks and to fix a great number of these parameters. For example, they fix every kernel size to 3×3 and all the strides to 1. In the same way, the pooling layers' size is 2×2.
Another approach is the work of Lingxi Xie and al. [3]. They decided to create a great number of human-defined network building blocks. Their algorithm then generates networks by choosing which blocks to include and which order to use. This also ensures that every generated network will be valid and will not have dimensionality issues.
Another difficulty when building deep neural networks is the risk of gradient vanishing. Because network depth is very important for complex tasks, Yanan Sun and al. [2] decided to add random skip connections in their models. This allows solving this problem although it adds more dimensions in the search space.

1.3 Midway baseline solution

As a baseline solution, we chose a very naive approach to this problem. We decided to let free all the parameters to be sure to find the optimal architecture. The generation of the architectures was fully randomized. The numbers of convolutional and fully connected layers were first randomly chosen between 0 and 10. We chose 10 as the maximal value to reduce the risk of gradient vanishing with too high depth. For each of these layers, the different parameters were then also randomly chosen.
The evaluation was done on the validation dataset accuracy, the top 5 architectures were selected and 5 more architectures were randomly chosen among the population. Those 10 individuals were bred to create a new generation of 100 individuals.
This approach allowed us to achieve a 40.6% accuracy and thus showed that genetic algorithms had the ability to find a non-random solution even with a huge search space.

2 Methods

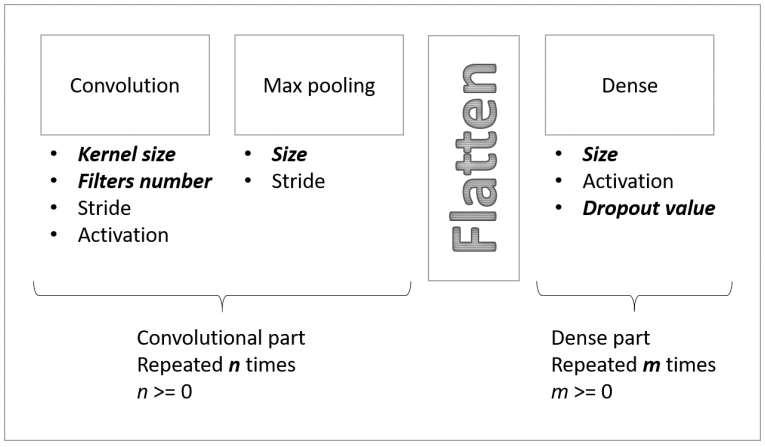

Figure 1: Architecture of the networks generated with our genetic algorithm. Variable parameters are written in italics.

The precedent approaches of genetic algorithm research of an image classification network tried to face the high dimensionality problem by fixing parameters or using pre-built blocks to reduce possibilities [2, 3]. We think that this could restrain the algorithm too much and reduce its ability to find the right architecture. More, choosing which parameters should be fixed and what value to use is again a tough task for non-specialists.

We decided to let the main part of the parameters free and use some general well know heuristics to choose and fix the others. Figure 1 shows the architecture of the networks generated with our algorithm and their variable parameters.
One big difficulty raised by this wide search space is that if architectures are purely randomly generated, a great part of them are invalid or are not able to use the input information to produce the output. This greatly reduces the algorithm's ability to find the right architecture, so we imagined an architecture creation algorithm that generates only valid and meaningful architectures.

2.1 Fixed parameters selection

Even if we want to let free the maximum number of parameters, we could notice that some of those parameters always have the same values. Exploring some of the state of the art architectures for different datasets, we noticed that those parameters very often have the same values. Because of this, we decided to fix them to simplify the solution search without degrading the final performance.
We decided to fix the convolution and pooling strides as well as the activation functions.
The convolution stride is fixed to 1 and the pooling stride is set to the pooling size. This choice is very widely spread and gives very good results. More, this simplifies the valid architecture generator algorithm described in the next section.
The activation function is *ReLU*. It appears that all the activation functions have the same capacity to solve a problem, so exploring this dimension is not useful to find the best architecture. We chose *ReLU* because it allows an easy back-propagation and thus accelerates the learning process.

2.2 Population initialization

As explained before, because of the presence of a great number of parameters, the naive approach of our baseline implementation leads to the generation of a lot of invalid architectures. The problem comes from the convolutional part of the model which can lead to dimensional problems. Using too many convolutions or too big ones leads to dimensionality mismatches or even to a programmatically valid architecture that looses too much input information.
To face this problem, we had to design a new random generation algorithm that would only create valid and meaningful architectures.

Algorithm 1 Valid convolutions generator

```
output_size ← network_input_size
while output_size > min_output_size do
    kernel_size ← random_int()
    filters_number ← random_int()
    new_conv ← conv(kernel_size, filters_number)
    network.append(new_conv)

    should_add_pooling ← random_bool()
    if should_add_pooling then
        pooling_size ← random_int()
        new_pool ← pool(pooling_size)
        network.append(new_pool)
    end if
    output_size ← network.output_size
end while
```

In algorithm 1, the `min_output_size` allows guaranteeing that the output of the convolutional part of the model still has some dimensionality to keep the information of the input data.

Once a valid convolutional part has been created, a flatten layer is added to the model to go from the 2D convolutional layers to the 1D dense layers.
The numbers of fully connected layers is then chosen between 0 and 10. As for our baseline implementation, value 10 has been chosen to reduce the risk of gradient vanishing. For each of these layers, the different parameters are then also randomly chosen.
Finally, a last dense layer corresponding to the desired network output size is added, with a categorical cross-entropy activation. We arbitrary fixed the initial population size to be 100.

2.3 Evaluation

The selection of the best networks is done on their validation accuracy. Each network in the population is trained for 25 epochs on the training set. If the validation loss stops decreasing before these 25 epochs, training is stopped to reduce global execution time.
After the training is complete, the networks are benchmarked on a validation set and ordered depending on their accuracy on this set.

2.4 Evolution

After evaluation, we select 10 individuals that will be used to create the new generation. The first 5 correspond to the five networks that showed the best performance on the validation dataset. The 5 others are randomly picked between the other networks. This allows generating new networks that will be better on average while keeping variability in the case where the best networks were only a local optimum. We then breed each pair of selected individuals to reach a new 100 individuals population.
As for generation, naive breeding would lead to a large number of invalid networks, thus we also imagined a breeding algorithm that generates only valid and meaningful architectures. As before, the invalid architecture problems only come from the convolutional part.

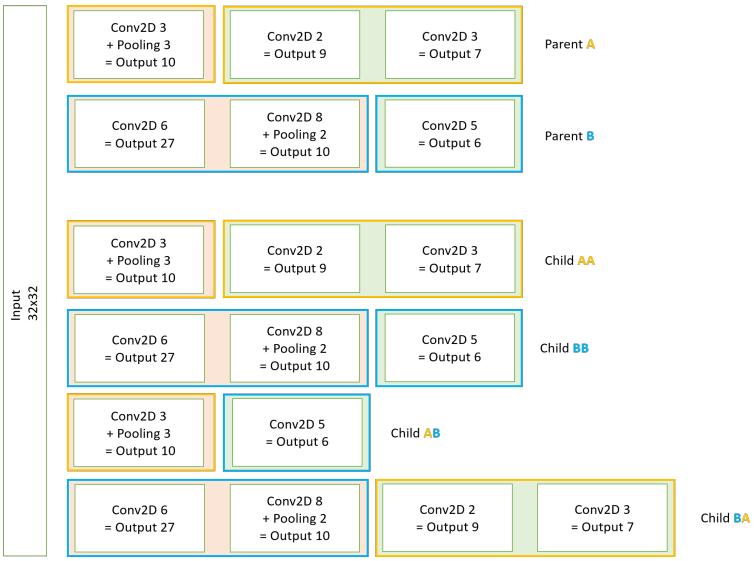

Figure 2: Example of a breeding between two parent networks. The blocks of the same background colors were aligned together. The children were build by randomly selecting blocks from parents.

An example of our solution is visible in figure 2, we first align the convolutions from the two architectures on their output sizes. The alignment is done with the dynamic time warping algorithm (also written DTW), for time efficiency, we used the fastDTW algorithm which is an approximation of DTW. This gives us aligned blocks that contain a variable number of layers, two aligned blocks have a similar output size. In figure 2, the red blocks are equivalent and the green blocks are equivalent.
Then we can randomly choose blocks from the two parents models to create the child model. In figure 2, children are built by taking one of the two red blocks and one of the two green blocks. If the two parents are valid, this algorithm has a much larger chance of creating valid children than the purely random naive approach.
The dense part can then be bred naively. For each of the parameters of each dense layer, we randomly choose one of the two parents' values to create the child model.
With some probability, a mutation can also be performed, some parameters are changed following a normal distribution around their previous values.

2.5 Performance evaluation

The main metric for this problem is the test dataset accuracy. To obtain reliable values, we divided all three datasets into 3 sub-datasets each. The training sets are used to train all the models generated by the genetic algorithm. The validation sets are used to compare networks' performances within the same generation. The test sets are used to give the test accuracy of the network chosen by our algorithm.
The usage of a validation dataset prevents the algorithm from choosing the network on its validation dataset results and 'overfit' the validation dataset.
The *MNIST* and *fashion-MNIST* training sets contain 60000 samples, the validation sets contain 8000 samples and the test sets contain 2000 samples.
The *Cifar10* training set contains 50000 samples, the validation set contains 8000 samples and the test set contains 2000 samples.

3 Results

With the midway baseline implementation, we could build a network that reached a 40.6% accuracy on the *MNIST* test dataset.
With the slight reducing of parameter number and the valid architectures generation, we rose the *MNIST* test dataset accuracy to 99.07%. The evolution of the validation and test performances is shown in figure 3. The architecture that achieved the best performance is shown in figure 4.

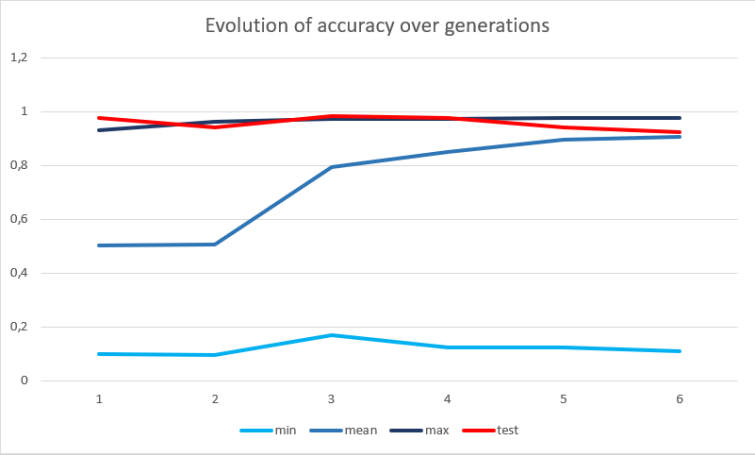

Figure 3: Evolution of minimum, mean, and maximum validation accuracies and test accuracy over the generations for the *MNIST* dataset.

On the *Fashion-MNIST* dataset, the test accuracy reached 90.01%. The evolution of the validation and test performances is shown in figure 5. The architecture that achieved the best performance is shown in figure 6.

On the *Cifar10* dataset, the test accuracy reached 65.96%. The evolution of the validation and test performances is shown in figure 5. The architecture that achieved the best performance is shown in figure 6.

Note that the three results given here were obtained by taking the best performing model generated by the genetic algorithm for each dataset and training it until convergence. This explains why the announced test accuracies may be higher than the validation and test accuracies shown in figures 3, 5 and 7.

4 Discussion

The 99.07% accuracy on *MNIST* is an extremely good result as it almost reaches the state of the art performance. The current convolutional and dense state of the art that does not use any pre-processing

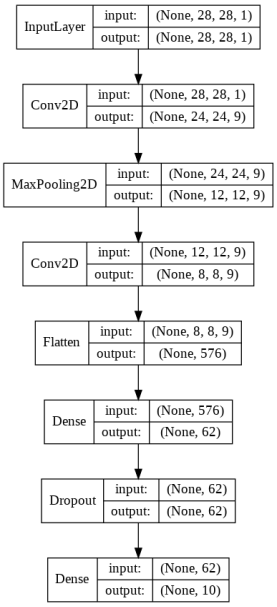

Figure 4: Architecture of the model generated with our genetic algorithm that achieved the best test accuracy (99.07%) on the *MNIST* dataset.

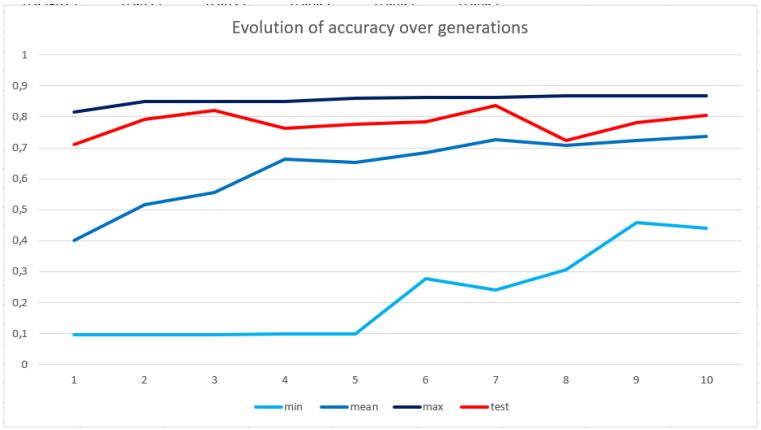

Figure 5: Evolution of minimum, mean, and maximum validation accuracies and test accuracy over the generations for the *Fashion-MNIST* dataset.

or data generation is 99.65% [4].
However, this result can be discussed; as shown in figure 3 the best test accuracy is reached after only three generations, then the algorithm starts to overfit on the validation dataset and the test accuracy decreases. The first generation already has extremely good performances, this may be because the *MNIST* dataset is quite simple, so looking for the best solution is fast. The algorithm not reaching the state of the art performance may be because some human tweaking is always necessary for achieving state of the art performances.

The 90.01% accuracy on *fashion-MNIST* is a very good result too. The current convolutional and dense state of the art that does not use any pre-processing or data generation is 93.4% [5].
As before, figure 5 shows that the max and test accuracies are very high from the first epoch, but this time the maximum test accuracy is reached at epoch 7. We can also observe that even if max accuracy stays on a plateau, the mean and minimum accuracies raise across the generations. This proves that the genetic algorithm tends to select only the best architectures and that the choices are not random.

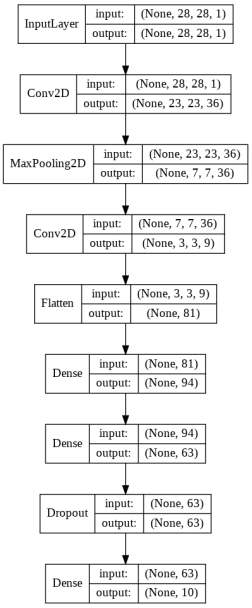

Figure 6: Architecture of the model generated with our genetic algorithm that achieved the best test accuracy (90.01%) on the *fashion-MNIST* dataset.

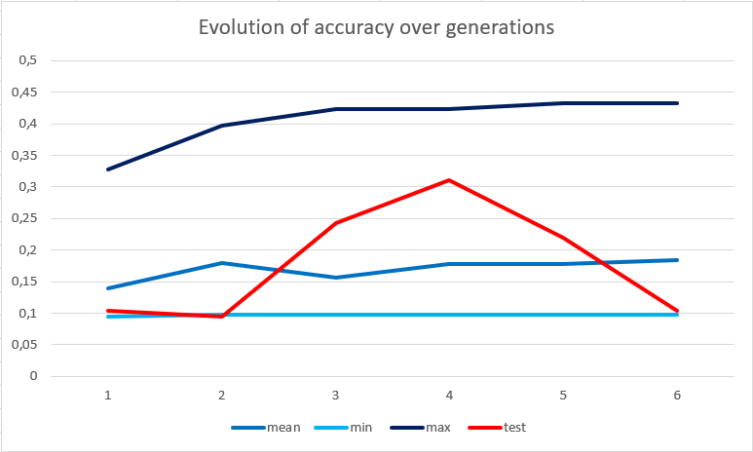

Figure 7: Evolution of minimum, mean, and maximum validation accuracies and test accuracy over the generations for the *Cifar10* dataset.

The 65.96% accuracy on *Cifar10* is a way worse performance than for the two first datasets. Although it is not that bad, it stays very far from the state of the art performance. The current convolutional and dense state of the art that does not use any pre-processing or data generation is 95.59% [6].
Two things can be observed in figure 7. First, the maximum validation accuracy seems to be around 45%. It seems strange because the state of the art network is not much more complex than the solutions proposed by our algorithm. One noticeable thing is that the algorithm trains all of our models for a maximum of 25 epochs, this may have a great influence on our results. This seems to be confirmed by the fact that our fully trained model achieved around 65% accuracy, which is way higher.
The second thing is that the test accuracy does not seem to follow the validation accuracy. It is first very low, then raises a bit without reaching the max validation accuracy and finally drops. It seems that our selection on the validation accuracy does not generalize well to the test accuracy.

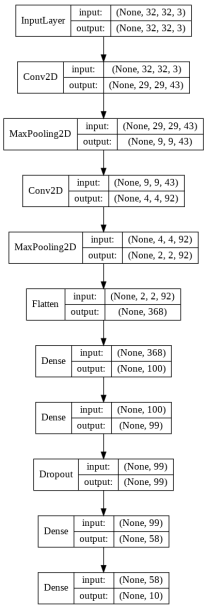

Figure 8: Architecture of the model generated with our genetic algorithm that achieved the best test accuracy (65.96%) on the *Cifar10* dataset.

Those results show us that our genetic algorithm can easily approach state of the art performances for easy datasets such as *MNIST* or *Fashion-MNIST*. For harder datasets such as *Cifar10*, the current version of the algorithm has more difficulties but still achieves decent results. This may be because of the small training time given for each model, this small time may not allow the models to show their ability to learn the dataset.
Our approach may be a good start point for unexperimented deep learning developers, although some supplementary work may be needed to achieve state of the art results. The big drawback of this technique is the great computing power and time needed for evolution. During our tests on small datasets such as *MNIST* with a training time of only 25 epochs by model, each generation took around 1 hour on a g4dn_xlarge Amazon GPU instance. This is for sure shorter than the time needed by a human to create a good architecture but it needed to be underlined.

5 Future work

A lot of work is still needed to improve this algorithm. Different directions can be explored. First, as a small epochs number seems to be a problem for difficult datasets, one solution may be to automatically determine the right trade-off between execution time and generation performance.
Another limitation of this algorithm is that the generated architectures are very conventional, a sequential model composed of a convolutional part and a dense part. However, for advanced datasets, those classical architectures are not enough, some advanced networks have branching and merging parts that allow extracting more data from the input. Exploring this kind of architecture would also be very interesting in the case of our project.

References

[1] Zalando research. Fashion-MNIST dataset, 2017.

[2] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G. Yen. Automatically designing CNN architectures using genetic algorithm for image classification. *CoRR*, abs/1808.03818, 2018.

[3] Lingxi Xie and Alan L. Yuille. Genetic CNN. *CoRR*, abs/1703.01513, 2017.

[4] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.

[5] khanguyen1207. Fashion-MNIST dataset state of the art solution, 2019.

[6] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*, 2015.