# Generating CNNs using Genetic Algorithm

Lev Martin Zachar

August 2020

## 1 Introduction

The goal of this project is to explore the usability of GA (Genetic Algorithm) to generate CNN (Convolutional Neural Network) for image classification that would work reasonably well on multiple datasets such as MNIST, FASHION-MNIST, CIFAR-10 and CIFAR-100 (fine). One of the most popular approaches to generating neural networks via genetic algorithm is it's variation NEAT (Neuro Evolution of Augmented Topologies) [SM02], which is an algorithm set up to evolve minimal networks by initializing all networks with no hidden nodes. Each individual in the initial population of GA consists of input nodes, output nodes, and a series of connection genes between them. By itself, a network of this kind may not necessarily work, but when combined with the idea of speciation proves to be a powerful idea in evolving minimal, yet high-performing networks.

However, since NEAT leverages from evolving minimal networks (with no hidden nodes) makes the algorithm unusable for generating CNNs due to the fact that they need multiple hidden layers (convolutions). In the following chapters I shall provide a short description of CNNs in the context of generating its layers, various challenges to think of when implementing a GA to generate CNN and propose own solution.

### 1.1 Convolutional networks

A CNN consists of input, hidden and output layers. The hidden layers of CNN are mainly convolutional and pooling layers. Pooling layers are usually incorporated between two successive convolutional layers. The pooling layers reduce the number of parameters and computation by down-sampling the representation.

For the purposes of this project, I have been working with image datasets that often provided low-resolution images, lowest being 28x28 pixels and highest 32x32 pixels. This introduced a limitation on the number of hidden layers as they reduce dimensions. The lower count of CNN layers complements the idea of generating deepest possible networks for the data I am using, and at the same time, not too deep that the network would become computationally unfeasible - especially while training multiple species and generations. In order to reduce compute time even further, a method called *Early stopping* was used.

## 1.2 Genetic algorithm

The most common use of GA is to generate a high-quality, or very rarely an optimal solution to optimization and search problems by utilizing biologically inspired operators such as selection, crossover and mutation. For a long time, it was quite common to use a simple GS (Grid Search) for the tuning of neural network hyperparameters. However, this approach may not be feasible when one is not fully confident which hyperparameter space to set for the GS. When using a GA, one can set a more broad hyperparameter space while not prolonging the search time, that is if we were to compare GS and GA execution time.

## 2 Proposed algorithm

Main challenges to solve while coming up with a GA for CNN are species (chromosome) representation and crossover.

### 2.1 Species Representation (Encoding)

The way in which to encode species lays out the path for how an algorithm will handle the key evolutionary processes of selection, crossover, and mutation. Any encoding will fall into one of two categories, direct or indirect. [Hei20] I took a closer look on two direct encodings (binary and graph), and thought about a custom indirect encoding. Ultimately, I chose an implementation using indirect encoding.

#### 2.1.1 Binary encoding

When it comes to GA species representations, by far the most used one is binary encoding. With this data structure, one can easily perform the needed evolution operations such as crossover and mutation. However, implementing binary encoding representation for neural network, especially when using multiple hyperparameters, might not be the best approach as long bit codes may require additional overhead and be harder to debug.

#### 2.1.2 Graph

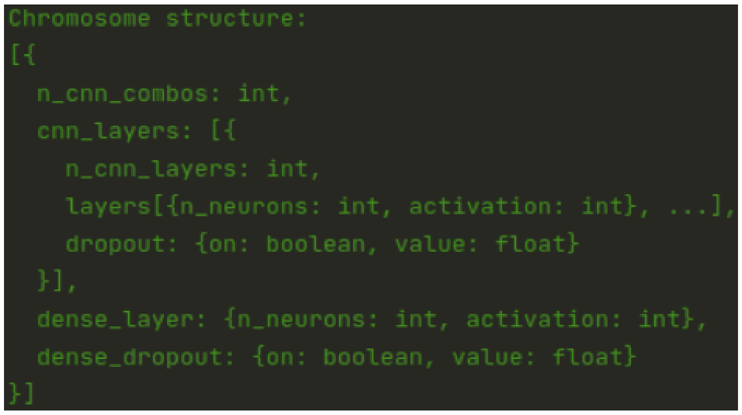
While a graph representation would make the most sense for representing neural networks, upon further research, to make this kind of representation work would take more effort. One of the main benefits this representation could provide is an often phenomenon of ”dying” ReLU units while training neural networks. With this approach, it’s more likely that during crossover, an event would occur in which one of the species would get some of it’s ”dead” ReLUs swapped or mutated, along with it’s connections and produce a truly advanced specimen.

### 2.1.3 Custom data structure

Custom indirect encoding in a form of a data structure introduced a benefit of not having to take into account encoding the weights and connections, just the architecture and hyperparameters.

In figure 1 we can see the structure of a chromosome, where:

- **n_cnn_combos** stands for the number of *combinations of CNN layers*.
- **cnn_layers** (*combinations*) contain the number of layers (*n_cnn_layers*), layers themselves represented by number of neurons and their encoded activations, and possible dropout regularization layer.
- **dense_layer** with *n_neurons* number of neurons and an encoded *activation.*
- **dense_dropout** possible dropout for the previous dense layer, *on* signifies whether to use this this dropout layer and *value* is the dropout rate.



```
Chromosome structure:
[{
  n_cnn_combos: int,
  cnn_layers: [{
    n_cnn_layers: int,
    layers[{n_neurons: int, activation: int}, ...],
    dropout: {on: boolean, value: float}
  }],
  dense_layer: {n_neurons: int, activation: int},
  dense_dropout: {on: boolean, value: float}
}]
```

Figure 1: Chromosome structure.

## 2.2 Crossover

There are three types of crossover operations I took into consideration - single point, two point and uniform. Due to the nature of chosen chromosome structure, the most fitting operation is single point crossover.

For two point crossover there are cases where could not be used, for instance, if a generated chromosome has only one convolutional layer and single dense layer, which is the most simple chromosome that can be generated, they are still desirable for crossover and there's only one crossover point available.

Uniform crossover would make the least sense, since the crossover would take additional overhead as the depths of CNNs may vary by multiple layers.

## 3 Implementation

The project was implemented in Python using libraries such as Keras, Tensorflow and Sklearn.

Key components of the GA implementation, as in any GA, is the chromosome generation, fitness calculation, selection, crossover and mutation.

During the initial period of testing various parameters for the algorithms were tried. Thanks to this testing, final parameters were selected that would used for all the datasets. These can be seen in the table 1.

| Parameter | Value |
|---|---|
| Genetic algorithm | |
| Generation size | 32 |
| New generation | 1/2 Tournament + 1/2 New blood |
| Crossover | Single point |
| Mutation | 10% |
| Neural network | |
| Epochs | 20 |
| Early stopping | Patiance = 4 |
| Optimizer | Adadelta |
| Batch size | 128 |

Table 1: Chosen GA and CNN parameters.

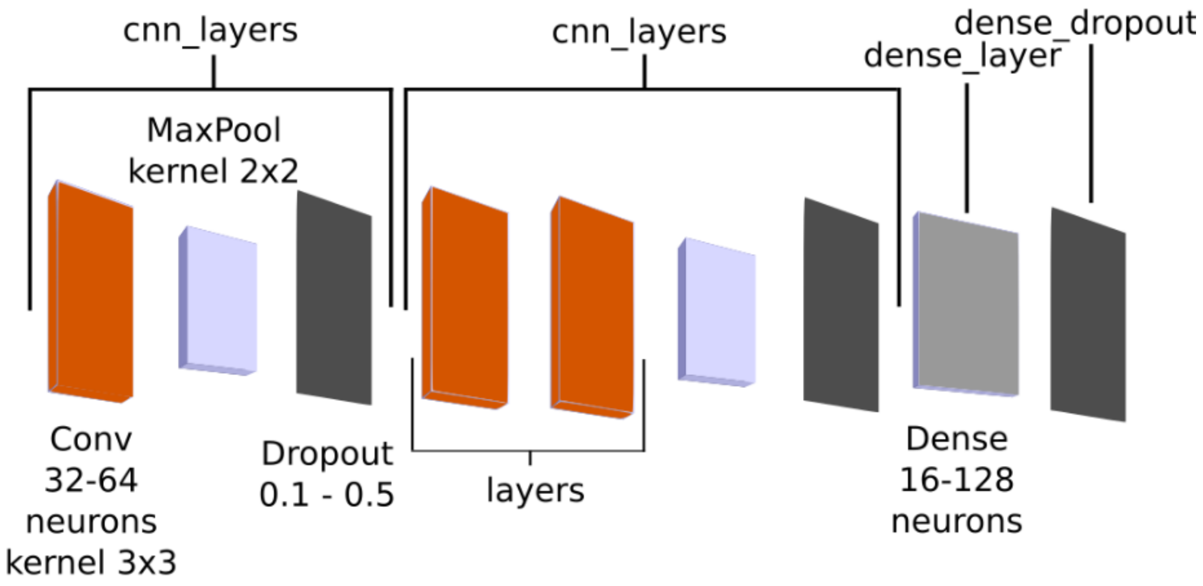


Figure 2: Visualized CNN chromosome structure.

## 3.1 Chromosome generation

Chromosome generation is implemented by simply assigning values from a pre-defined range to the variables of chromosome structure described earlier. The value ranges can be seen in the table 2 below.

| Variable | Value range |
|---|---|
| n_cnn_combos | 1-2 |
| n_cnn_layers | 1-3 |
| layers - n_neurons | 32-64 |
| layers - activation | 0-2 |
| dropout - on | 0-1 |
| dropout - value | 0.1-0.5 |
| dense_layer - n_neurons | 16-128 |
| dense_layer - activation | 0-2 |
| dense_dropout - on | 0-1 |
| dense_dropout - value | 0.1-0.5 |

Table 2: Value ranges of a chromosome.
Note: *Activation mapping: 0 - relu, 1 - elu, 2 - selu*

## 3.2 Fitness calculation

The fitness function is very simple, after each evaluated specimen (neural network), it's fitness (accuracy) is pushed into a *fitness* array. Once all the species are evaluated, generation fitness is calculated.

$$generationfitness = \sum_{n=0}^{len(fitness)-1} fitness[n]$$

## 3.3 Tournament selection

Selection method used is *tournament* which compares defined number of species, in this case a square root of population, and selects a winner (best accuracy). The process is repeated until we have selected a sufficient number of specimen for crossover.

## 3.4 Single point crossover

When it comes to crossover, only the layers (convolutional and dropout) are taken into account. These layers are extracted into a support array. After extraction, a crossover point is randomly selected from the range of:

*< 1, min(len(support_array_parent_a), len(support_array_parent_b)) - 1>*

Consequently, the crossover is executed by two specimen exchanging the latter portion of their chromosomes, followed by their reconstruction.

## 3.5 Mutation

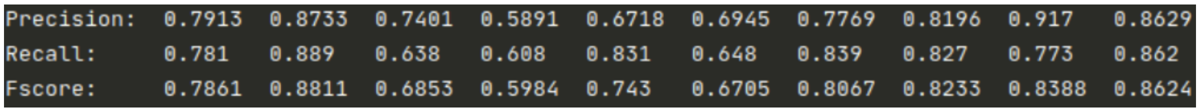
The probability of mutation is set to 10%, which may seem to be high, but after initial testing and due to the fact that there are less epochs, it needed to be a bit more drastic. Naturally, all of the values in the chromosome structure can be mutated. The values that the individual parts of chromosomes can mutate to are according to the value ranges in the table 2 above.

# 4 Testing and monitoring

Testing and monitoring were split into two different categories since each category has it's own distinctive indicators of success.

## 4.1 Convolutional neural networks

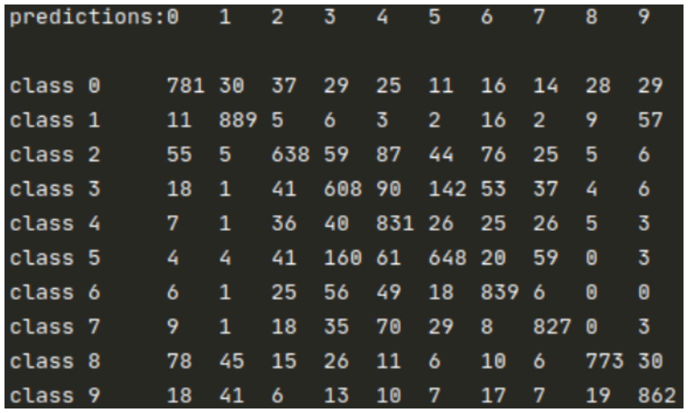
In order to assure quality of CNN classifications, a common method of cross validation was used. The data was split into train, test and validation sets. Next, metrics such as Precision, Recall and Fscore of each class were monitored, where the classes are represented by numbers, or in the case of figure 3 by order from left to right (the numbers are missing). Representation is as follows airplane(0), car(1), bird(2), cat(3), deer(4), dog(5), frog(6), horse(7), ship(8), and truck(9).



```
Precision:  0.7913  0.8733  0.7401  0.5891  0.6718  0.6945  0.7769  0.8196  0.917   0.8629
Recall:     0.781   0.889   0.638   0.608   0.831   0.648   0.839   0.827   0.773   0.862
Fscore:     0.7861  0.8811  0.6853  0.5984  0.743   0.6705  0.8067  0.8233  0.8388  0.8624
```

Figure 3: Precision, Recall and Fscore of each class.

Finally, the last metric which also helped visualising possible biases and problems of classifying specific classes, was a confusion matrix.



```
predictions:0   1   2   3   4   5   6   7   8   9


class 0     781 30  37  29  25  11  16  14  28  29
class 1     11  889 5   6   3   2   16  2   9   57
class 2     55  5   638 59  87  44  76  25  5   6
class 3     18  1   41  608 90  142 53  37  4   6
class 4     7   1   36  40  831 26  25  26  5   3
class 5     4   4   41  160 61  648 20  59  0   3
class 6     6   1   25  56  49  18  839 6   0   0
class 7     9   1   18  35  70  29  8   827 0   3
class 8     78  45  15  26  11  6   10  6   773 30
class 9     18  41  6   13  10  7   17  7   19  862
```

Figure 4: Confusion matrix.

## 4.2 Genetic algorithm

In addition to monitoring the fitness of species and generations, multiple execution time outputs were generated to ensure that the whole GA pipeline worked correctly. That being said, the evolution of fitness over the generations is still the single most important indicator, that the GA is working correctly. The most recurring fitness development can be seen in figure 5.

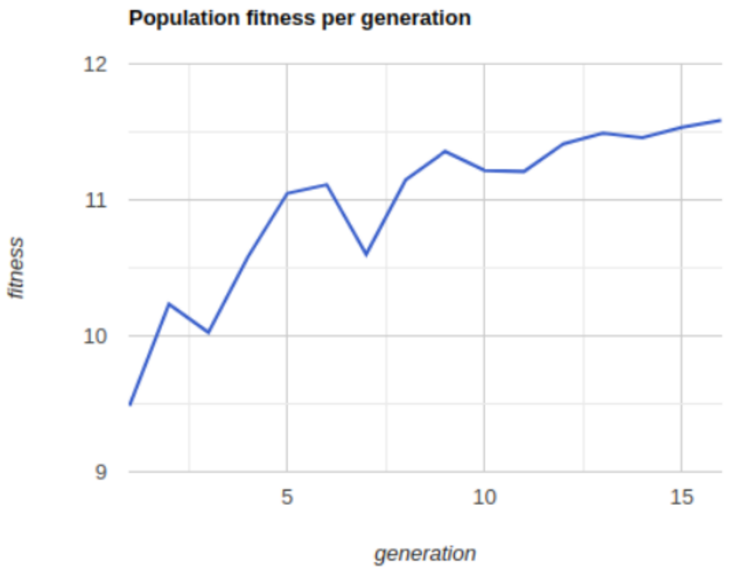


Figure 5: Fitness evolution on CIFAR-100 (fine) dataset.

# 5 Results

During the effort of achieving the best result on the most difficult task, that is **CIFAR-100 (fine)**, after the evolution finished, the best specimen was selected for further training and would be stopped by *Early stopping*. This process managed to increase the **accuracy to 48.72% and was trained for additional 34 epochs**. The other tasks such as MNIST, FASHION-MNIST and CIFAR-10 were more trivial, since a majority of reasonable CNN models are already able to achieve very good results on them. However, the GA algorithm proves to be most useful when one needs to tackle multiple small problems. It may relieve a lot of work and in some cases, the final model could often be a good enough solution. The results of the implementation and their comparison similar implementations are available in next paragraph and comparison to the state-of-the-art networks can be seen in the table 3.

| Dataset | My Genetic algorithm | State-of-the-art network |
|---|---|---|
| MNIST | 98-99% | 99%+ |
| FASHION-MNIST | 98-99% | 99%+ |
| CIFAR-10 | 75% | 99.37% |
| CIFAR-100 (fine) | 48.72% | 93.51% |

Table 3: Comparison of My Genetic algorithm and State-of-the-art networks

Comparison of CIFAR-10 results with similar implementations:

- Evolving Deep Neural Networks [Mii+17] (average network)
  - After 120 epochs reached 20% error
  - After 300 epochs reached 7.3% error
- Scalable Bayesian Optimization Using Deep Neural Networks [Sno+15] (best network)
  - After 120 epochs converged to 6.2% error
  - At 12 epochs already had only about 20% error
- My Genetic algorithm (best network)
  - After 20 epochs reached around 25% error

From my point of view the concept of using GAs to generate CNNs has a lot of potential. While it is true that it may be limited by computational resources, for some use cases, it may be a completely valid approach. The shortcomings of my implementation are mainly not using more hyperparameters and regularization such as *l1* and *l2*, as well as not having more diverse hyperparameter value ranges which was due to my limited computational resources.

## Installation and execution

The project was developed in Linux (Ubuntu) environment. In order to run the program, please follow these steps:

1. Have **Python3** installed.
2. **Unzip** the *Zachar_Lev_Martin_PV026_GA_CNN_Oprava.zip* file.
3. **Run** the Bash script *linux_run_neuroevolution.sh.*

   ```
   bash linux_run_neuroevolution.sh
   ```

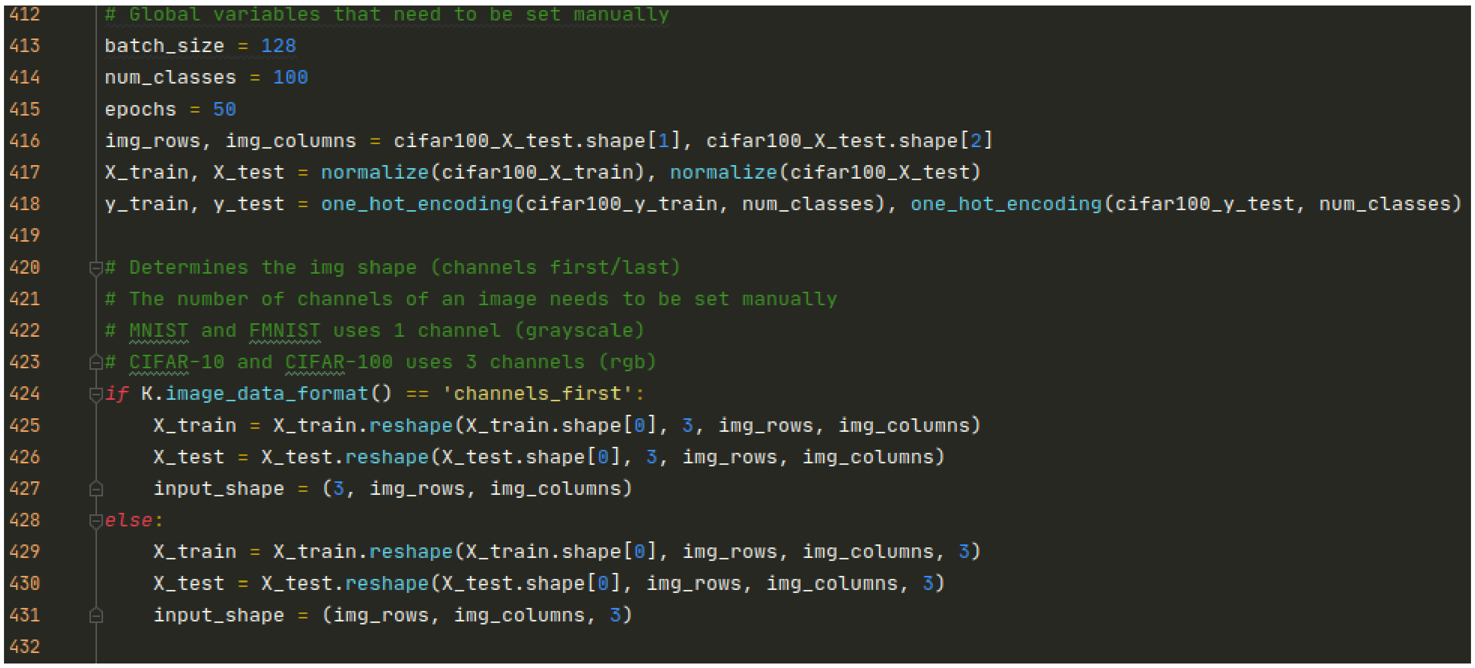
   Or, if you are using Windows open up a Bash terminal and run (this may not work and may need a little debugging).

   ```
   bash windows_run_neuroevolution.sh
   ```

These scripts should run the program in virtual environment which has all the libraries installed.

In order to run the program on GPUs, a custom CUDA installation is required based on graphics card in use.

After the program is runnable, depending on the dataset, the constants (except for number of epochs and batch size) need to be adjusted:



```
# Global variables that need to be set manually
batch_size = 128
num_classes = 100
epochs = 50
img_rows, img_columns = cifar100_X_test.shape[1], cifar100_X_test.shape[2]
X_train, X_test = normalize(cifar100_X_train), normalize(cifar100_X_test)
y_train, y_test = one_hot_encoding(cifar100_y_train, num_classes), one_hot_encoding(cifar100_y_test, num_classes)

# Determines the img shape (channels first/last)
# The number of channels of an image needs to be set manually
# MNIST and FMNIST uses 1 channel (grayscale)
# CIFAR-10 and CIFAR-100 uses 3 channels (rgb)
if K.image_data_format() == 'channels_first':
    X_train = X_train.reshape(X_train.shape[0], 3, img_rows, img_columns)
    X_test = X_test.reshape(X_test.shape[0], 3, img_rows, img_columns)
    input_shape = (3, img_rows, img_columns)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_columns, 3)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_columns, 3)
    input_shape = (img_rows, img_columns, 3)
```

Figure 6: Constants to be adjusted depending on the dataset.

## References

[SM02] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10 (2002).

[Sno+15] Jasper Snoek et al. *Scalable Bayesian Optimization Using Deep Neural Networks*. 2015. arXiv: `1502.05700 [stat.ML]`.

[Mii+17] Risto Miikkulainen et al. "Evolving Deep Neural Networks". In: *CoRR* abs/1703.00548 (2017). arXiv: `1703.00548`. URL: `http://arxiv.org/abs/1703.00548`.

[Hei20] Hunter Heidenreich. *NEAT: An Awesome Approach to NeuroEvolution.* 2020. URL: `https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f` (visited on 09/30/2010).