Genetic Algorithm for Neural Network Hyperparameter Optimisation

Maxim Khovansky

December 2024

1 Best model overview

The architecture of my best model consists of 4 convolutional layers followed by 2 fully connected layers. The convolutional layers consist of a max-pooling operation applied to a convolution passed into an activation function, while the fully connected layers consist of a linear transformation passed into an activation function and include dropout for regularization.

The hyperparameters of the model are as given in the tables below (Table 1a and Table 1b):

Layer	**Filters**	**Kernel Size**	**Stride**	**Padding**	**Pool Size**	**Activation**
Layer 1	32	5	1	0	2	relu
Layer 2	16	7	2	1	3	relu
Layer 3	64	5	1	1	2	swish
Layer 4	16	3	1	0	2	swish

(a) Convolutional Layers

Layer	**Neurons**	**Activation**	**Dropout**
Layer 5	128	relu	0.1
Layer 6	32	sigmoid	0.1

(b) Fully Connected Layers

Table 1: Best model hyperparameters

The validation-set accuracy of the model is **0.9387** to 4 s.f. Indeed, both the genetic.py and best_net.py scripts agreed on this around this accuracy for the model.

2 Genetic algorithm methodology overview

Before configuring the initial population, I implemented the following:

- Instead of simply repeating the training data 10 times, I performed data augmentation by adding some random noise to the training data during each iteration.
- I defined the Net class to allow neural networks to contain both convolutional and fully connected layers: since the input data is audial, convolutions might help identify recurring features of the data.
- Defined the meta-hyperparameters, including population size (set to 10), number of generations (set to 4), number of parents to bear children (set to 4), mutation rate (set to 0.2), probability of a layer being convolutional (set to 0.8), and the noise level during data augmentation (set to 0.1).
- Defined the blueprint, from which all of the parameters in the initial population and most of the parameters in consequent generations were sampled. The blueprint contained the maximum initial

number of layers per genome (set to 6), the possible layer sizes (set to powers of 2 from 8 through 512), the possible activations (set to sigmoid, relu, leaky relu, and swish), and the possible dropout rates in a layer (set to 0.1, 0.2, 0.25); it also contained parameters for convolutional layers, namely the possible numbers of convolution filters (set to powers of 2 from 8 through 64), the possible kernel sizes (set to 3, 5, and 7), the possible strides (set to 1 and 2 for simplicity), the possible paddings (set to 0 and 1 for simplicity), and the possible max-pooling sizes (set to only 2 and 3 for simplicity).

To define the initial population of specified size N, I created N genomes, each being a list of a randomly selected total number of genes not exceeding the maximum number of layers specified in the blueprint. Each gene would represent the set of relevant hyperparameters of each layer of the neural network associated with the genome, and would consist of a randomly chosen proportion (with the mean of this proportion being specified by "conv_chance") of convolutional layers followed by a series of fully connected layers.

For the selection stage, I simply sorted the population by fitness, and then selected the number of genomes with the highest fitness specified by "num_parents".

For the crossover stage, I generated a child out of two parents by iterating over each gene in the smaller parent and creating a corresponding gene in the child by randomly selecting each parameter from one of the two parents. I then simply copied the genes that the larger parent had while the smaller parent lacked from the larger parent onto the child.

Finally, for the mutation stage, I first computed the size of the mutated genome by sampling from a normal distribution with the mean corresponding to the size of the original genome and the variance being proportional to the mutation rate, specified by "mutation_rate", and rounding to the nearest whole. For genes that already existed in the original genome, I changed each of the parameters with the probability specified by "mutation_rate" to one of the values specified in the blueprint. For new genes, I created them the same way that I created all the genes in the initial population.

3 Reproductions

I ran the code 10 times, and my best architecture was obtained only on the 9th reproduction.

Due to the very large number of possible configurations that I chose to allow, the optimal architecture only emerged once out of the 10 reproductions. However, similar architectures emerged a total 3 times. The architectures of the two additional similar architectures are displayed below:

Layer	**Filters**	**Kernel Size**	**Stride**	**Padding**	**Pool Size**	**Activation**
Layer 1	16	7	2	1	3	leaky_relu
Layer 2	32	5	2	0	2	leaky_relu
Layer 3	8	5	1	1	3	swish

(a) Convolutional Layers

Layer	**Neurons**	**Activation**	**Dropout**
Layer 4	128	swish	0.1
Layer 5	512	leaky_relu	0.1

(b) Fully Connected Layers

Table 2: Similar architecture 1 (Accuracy = 0.9186)

Both architectures used a series of convolutional layers followed by two fully connected neural networks, just like my best model, and both architectures used a version of relu for the earlier convolutional layers and swish for the later convolutional layers, also just like my best model.

The best architectures generated in the other reproductions were likely local minima: architectures which were better than all sufficiently similar architectures, but which were nonethless not the best architectures possible. Many of them were completely unrelated to my overall best architecture, as they started from different initial conditions and found local minima which were far away from the local

Layer	Filters	Kernel Size	Stride	Padding	Pool Size	Activation
Layer 1	32	3	1	1	3	relu
Layer 2	64	3	2	0	2	relu
Layer 3	32	7	2	0	3	swish

(a) Convolutional Layers

Layer	Neurons	Activation	Dropout
Layer 4	64	sigmoid	0.1
Layer 5	512	relu	0.25

(b) Fully Connected Layers

Table 3: Similar architecture 2 (Accuracy = 0.9158)

minimum of my overall best architecture - which in turn is likely close to the global minimum, as it was independently almost reproduced on 3 occasions out of 10.

4 Genetic algorithm meta-hyperparameters

Mutation rate and noise level in data augmentation were two important meta-hyperparameters that I used. Their importance is demonstrated by the table below, which specifies the validation accuracy given selected values of mutation rate and noise level:

Parameter	Value	Accuracy
Mutation Rate		
	0.1	0.8826
	0.2	0.9172
	0.3	0.9030
	0.4	0.8682
Noise Level		
	0.001	0.8468
	0.010	0.8834
	0.100	0.9080
	1.000	0.5850

Table 4: Mutation rate and noise level vs validation accuracy

Both the mutation rate and the noise level appear to have a bell-shaped distribution with respect to validation accuracy: the optimal mutation rate appears to be around 0.2, while the optimal noise level appears to be around 0.1. This makes conceptual sense: if the mutation rate is too high, then fit genomes cannot retain their advantageous genes throughout the generations, decreasing performance; on the other hand, if the mutation rate is too low, then the genomes would struggle to jump out of shallow local minima by exploring promising configurations. Meanwhile, if the noise level is too low, then there is not enough diversity in the data for the model to generalise efficiently, while if it's too high, the augmented data becomes too different from the original data and the training data stops being representative of the validation data. It is important to note that differences in accuracy between the tested mutation rates are small and might potentially be a simple statistical artifact; on the other hand, the differences in accuracy between the tested noise levels are significant and likely to represent true differences in performance.