INTRODUCTION

Convolutional Neural Networks (CNNs) have recently attained human-level performance in tasks such as object recognition and detection [1, 2]. The advantage of using a neural network for the task of classification is that it is able to automatically extract high-level features from a given dataset in order to achieve it's target objective. However, one of the biggest downsides to neural networks is that their performance greatly depends on the chosen architecture. The task of selecting the network configuration is left to the user. This process is often guided by experimental trial and error and best practices shared amongst the deep learning community [3]. Several efforts have been made to automate the process of neural network generation [6-12]. These studies focus on genetic algorithms that allow for random mutations of the network architecture, network weights, or both and have shown that such an approach can help discover neural networks that are well-suited for the provided task.

Unlike many other neural network types, CNNs have been successful at extracting useful features for tasks they were not originally trained for, indicating that the CNNs filters can extract image features that are generally present in natural images [4, 5]. This quality of CNNs makes them a particularly interesting subject for Genetic Algorithm application, since network cross-over can, in this context, be well-defined as the transfer of convolutional filters from one trained network to another.

The goal of the project is to use adaptive evolutionary algorithms to automatically generate, test, and evolve CNNs with different architectures.

The goal of the project is to use genetic algorithms to automatically generate, test, and evolve CNNs with different architectures by mutating network architectures and preserving network weights from one generation to the next. The evolution of the network architectures is recorded and can be analyzed to provide insights into their impact on CNN performance. The deliverables for this project are an open source Python library that interfaces with TensorFfow and an evaluation of the genetic algorithm approach to creating CNNs using the CIFAR 10 dataset.

REVIEW OF EXISTING WORK

The idea of using evolutionary approaches to create neural networks has been explored in academic work (Table 1), but has not been widely adapted in practice (Table 2). The following is a brief summary of selected literature on the subject as well as current repositories on GitHub that offer implementations. The ECNN library introduces two novel approaches to evolving CNNs: preserving learned parameters from one generation to the next and crossing models by filter adoption.

Table 1: Academic Literature

Ref	Year	NN Type	Parameters Evolved	Data
6	2007	FF, RNN	architecture	XOR, word punctuation
7	2009	3 layer FF	weights, architecture	Iris, breast cancer
8	2007	FF	wights, architecture	Iris, wine, glass
9	2011	CNN	evolution parameters	MNIST
10	2015	CNN	weights	MNIST
11	2007	FF	architecture	credit card
12	2007	FF	wights, architecture	XOR, spirals

Table 2: Available Open Source Projects

Ref	Year	Language	Summary
13	2015	C#	self-implemented NN
14	2014	Java	simple NN
15	2015	Python	self-implemented NN
16	2016	C#	evolve structure and weights
17	2016	Python	FF nets, uses existing GA library

METHODS

The ECNN Python library, developed as part of this project, uses adaptive evolutionary algorithms to create CNNs best suited for a particular image recognition task. The library includes a simple interface with TensorFlow that gives users the ability to create and train TensorFlow models provided a minimal amount of information. The evolutionary algorithms used are based on a tournament model: during each generation, the best-performing CNNs are chosen and either mutated or crossed to create the next generation. If a mutation is leads to better performance, its probability increases during subsequent generations. The library allows for parallel CNN training across multiple GPUs.

Tournament

In order to configure the tournament, the user can specify the following tournament parameters: Number of Generations - the total number of generations. Population Size - the total number of models to be trained during each generation. Number Selected - how many models are selected to move on to the next generation. Fitness Criteria - the function used to select best models. The function can use one or more model attributes to select the best models. If more than one attribute is used, a weight can be assigned to each attribute. For example, validation accuracy can be used to select models that have best accuracy based on the validation set, or a combination of validation accuracy and number of trainable parameters can be used to add a penalty for models size. Population Restrictions - the user can specify restriction on the makeup of the model population. These include the minimum and maximum number of convolutional layers, maximum number of dense layers, restrictions

EVOLUTION

- Append convolutional layer
- Append dense layer
- Remove convolutional layer
- Remove dense layer
- Adopt filters from another model

During each generation, the selected models undergo mutation or cross-over, depending on probabilities drawn from a beta distribution. The mutation with the highest probability is chosen when producing a new model. Convolutional layers that have compatible filter sizes and input channels can be crossed.

on the filter size and number of filters as well as the size of the dense layers. In order to prevent overfitting, the validation set used to evaluate model performance is randomly selected for each tournament generation.

Mutation

During each generation, a model can be mutated or crossed with another model. Mutations include: appending a convolutional layer, removing a convolutional later, appending a dense layer, removing a dense layer. Model crossovers are performed by adopting filters from another model and is only done if the convolutional layers are compatible, that is have the same filter size and number of input channels. When mutating a model, the probability of each

Fitness Criteria

The user can specify performance metrics to be used in the model selection process. Multiple metrics can be combined by assigning weights to each. For example, validation accuracy the the number of trainable parameters could be combined to control the tradeoff between accuracy and model size.

mutation is drawn from a beta distribution and the mutation with the highest probability is selected. The parameters of the beta distribution for each mutation are adjusted after each generation in order to increase the probability of selecting mutations that lead to higher performance and reduce the probability of mutations that decrease model performance. During mutation or cross-over, the exponential moving averages of model weights is retained for all layers except those affected by the mutation.

Reusing Model Weights

The Exponential Moving Average of all weight variables is saved and restored in order to smooth over batch fluctuations and boost performance [18].

Adaptation

After each generation, the alpha and beta parameters, which control the skewness of the mutations probably distribution, are adjusted for each mutation. If a mutation lead to better performance as evaluated by the fitness criteria, its alpha parameter is increased, making the mutation more likely to occur in subsequent generations. Conversely, if a mutation lead to worse performance, its beta parameter is increased creating a positive skew in the mutation's probability distribution. Additionally, the layer parameter distributions are adjusted. For example, if models with larger dense layers outperform those with small dense layers, larger layers

become more likely in subsequent generations.

Training Functions

Another useful feature of the ECNN library is the functionalization of hyperparameters. Instead of using constant values, the user can specify function to be evaluated during training in order to adjust the hyperparameters dynamically. For example, the learning rate can be evaluated based on training loss history. Training functions are available for learning rate, batch size, stopping rule, number of training iterations, regularization strength, and dropout for dense and convolutional layers.

Experiments

Two tournaments were completed in order to compare the evolution and performance of models when different model fitness criteria are employed. The CIFAR 10 image dataset was used. The dataset consists of 60,000 images; 50,000 used for training and validation and 10,000 reserved for testing. The CIFAR 10 dataset contains images belonging to 10 image classes.

Both tournaments used following training functions.

Learning rate function: the learning rate was decreased by 25% if the loss fluctuated by more than 5% for four epochs in a row. If the loss decreased by less than 1% for four epochs in a row, the learning rate was increased by 1/3. The initial learning rate was set at 0.01 for new models or restored to its value from the previous generation of the model was not mutated.

Stopping rule: training was stopped if validation accuracy decreased for 4 consecutive epochs.

Batch size: 512

Training Iterations: 100 for selected models, 150 for models that were mutated.

Convolutional Layer Dropout: 0.2

Dense Layer Dropout: 0.3

Regularization Strength: 0.004

TRAINING FUNCTIONS

- Number of training iterations
- Learning rate
- Batch size
- Stopping rule
- Regularization strength
- Dropout

Instead of using constant hyperparameters, the ECNN library allows the user to create hyper parameter functions that can be evaluated during training in order to adjust the hyper parameters..

Tournament 1 Settings

Fitness Criteria: validation accuracy
Number of Generations: 10
Models selected: 4

Tournament 2 Settings

Fitness Criteria: the fitness criteria for Tournament 2 was changed to include a 5% penalty for model size.
Number of Generations: 8
Models selected: 4

For both tournaments, the training and validation sets were randomly selected during each generation. The training set consisted of 20,000 images and the validation set consisted of 1,000 images. After the final generation, the top-performing models were trained for an additional 1,000 epochs (with the training set restricted to 20,000 samples) before being evaluated on the test set.

RESULTS

Tournament Results

The highest previously attained test accuracy for the CIFAR 10 dataset was 96.53% at the time this project was completed [19]. The best-performing model produced in Tournament 1 achieved 81% test accuracy and would have been ranked in 44th place amongst recorded CIFAR 10 performance results since 2011. The best-performing model in Tournament 2 achieved 75.8% test accuracy and would have ranked in 50th place [20]. While these networks did not achieve state of the art performance, it is important to note that the resulting architectures were much smaller than state of the art networks. The winning models contained 58,242 and 8,827 parameters in Tournament 1 and

Tournament 1 winning CNN achieved

81%

test accuracy with

58,242

parameters.

Tournament 2 winning CNN achieved

75.8%

test accuracy with

8,827

parameters.

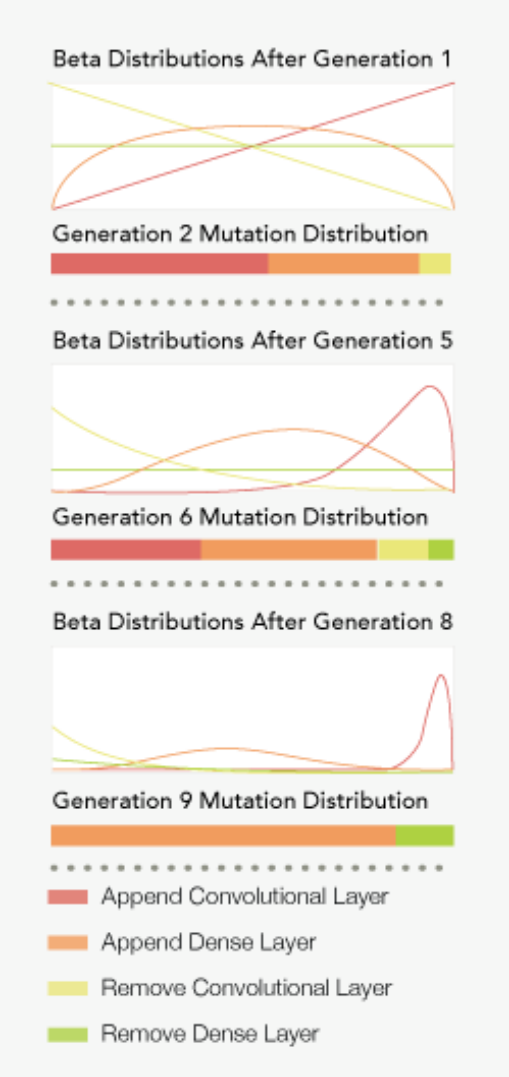
Tournament 2 respectively, while the state of the art networks are reported to have close to 1 million parameters [21]. Larger networks can be obtained using the presented approach by adjusting the limits on convolutional and dense layers when defining the population parameters and increasing the number of generations in the tournament.

Figure 2 shows a box plot of validation accuracy for each generation in Tournament 1. Earlier generations have a larger variation in performance, possibly due to larger diversity in the first several convolutional layers. In the third generation 17 out of 20 models shared the same generation 1 ancestor and therefore the same initial layer configurations as is illustrated in Figure 10. Figure 4 shows that average validation accuracy in Tournament 1 increased over time, with slight fluctuations, possibly due to random validation set selection.

Figure 3 shows that models tend to get larger over time. Figure 6 illustrates that models that achieve high validation accuracy have more parameters on average than smaller models in the same generation.

The results of Tournament 2, using a penalty for models size, show an increase in accuracy over time (Figure 6), however the increase in model size is less significant than in Tournament 1. In fact, the average number of parameters in selected models tends to be smaller than the average population (Figure 9).
It is interesting to note that the evolution of models in Tournament 1 and 2 are different due to the model fitness criteria. Tournament 1

Figure 1: Evolution of mutation probability distributions and resulting mutations



mutations consisted mostly of adding convolutional layers while Tournament 2 mutations consisted of adding and removing dense layers.

Figure 1 shows the evolution of the beta distributions for the 4 mutations as well as the resulting mutations in subsequent generation for Tournament 1. It can be seen that the probability distribution of the 'Append Convolutional Layer' becomes centered around higher values as the mutation results in higher validation accuracy. The mutation does not occur in generation 9 because all networks had reached the maximum allowed number of convolutional layers.

Methodology Evaluation Results

Does reusing weights improve model performance?

During each generation, the exponential moving averages of all layers not affected by mutation were restored for each mutated model. in order to see if reusing weights from a smaller model resulted in better model performance, models with preserved weights were compared to models with the same architecture using random weight initialization. Each mutation consisted of appending a new layer to the model. All models were trained for 150 epochs before being evaluated on the test set. Table 3 shows that using pre-trained layers significantly improves model convergence. The same test accuracy could be achieved without weight transfer, but, this would require an increase in the number of training epochs.

Does iteratively growing models lead to better model performance?

The evolutionary algorithm approach to creating CNNs assumes that small models that perform well in earlier generations will lead to higher performing larger models in subsequent generations after the addition of layers. In order to validate this hypothesis, the first two

Table 3: Test Accuracy for Models with Weight Transfer vs Random Initialization

Generation	1	2	6	10
Trainable Parameters	4,982	11,133	33,825	58,643
Epochs	150	150	150	150
Test Accuracy with Weight Transfer	51.65%	60.68%	65.37%	73.99%
Test Accuracy without Weight Transfer	34.77%	46.96%	30.62%	10.8%

convolutional layers (top-performing models in generation 1) of top performing models in generations 2 and 3 were replaced with poor-performing models in generation 1. The resulting models were retrained and evaluated on the test set. Table 4 shows that using better performing initial layers in larger models leads to better overall model performance.

Table 4: Initial Layer Replacement

Test Accuracy With Top-Performing Initial Layers	Test Accuracy With Low-Performing Initial Layers
58.4%	52.7%
69.6%	66.3%

Does crossing models lead to higher performance?

The assumption that transferring filters from another model would increase performance by adding more useful feature detectors was evaluated by comparing the performance of models with additional filters to those that did not undergo mutation. In Tournament 1, 74% models that adopted filters from another model performed better than their non-muted versions. In Tournament 2, filter adoption was successful 58% of the time. Figures 12 and 13 show that the winning models in each tournament underwent at least one filter adoption during their evolution.

CONCLUSION

Initial experiments suggest that evolutionary algorithms can help iteratively generate Convolutional Neural Networks with increasing performance. Transferring weights from a smaller network to a larger one can help the larger network converge faster and reduce training time. Without prior knowledge of the optimal network size, the strategy of ‘growing’ the network using mutation and cross-over while preserving the learned parameters is shown to produce higher-performing models over time. The ECNN library provides a flexible framework that allows users to configure tournament settings and define training functions in order to produce Convolutional Neural Networks for image classification.

Figure 2: Validation Accuracy for All Generations in Tournament 1

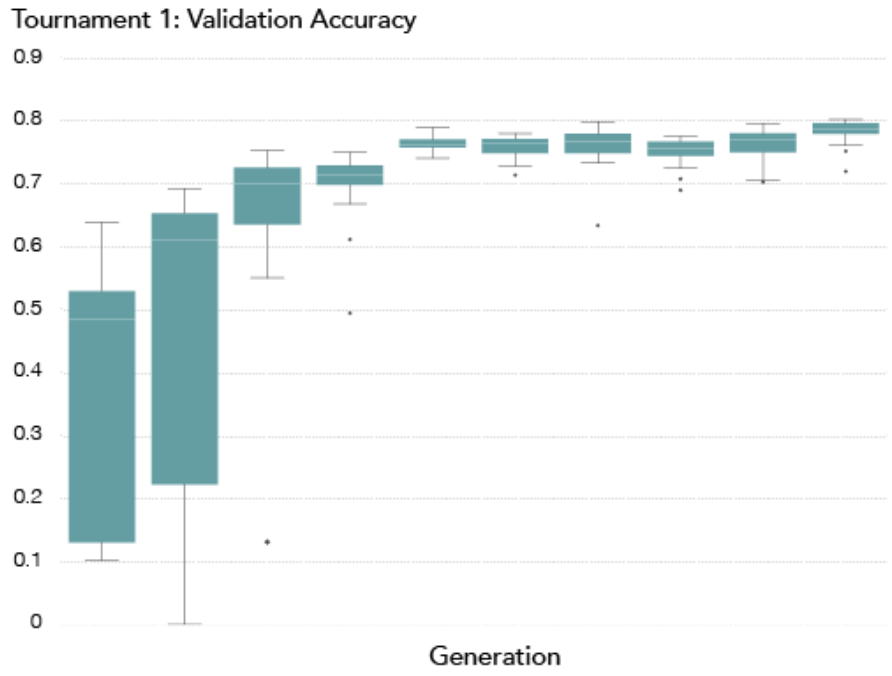


Figure 3: Number of Parameters for All Generations in Tournament 1

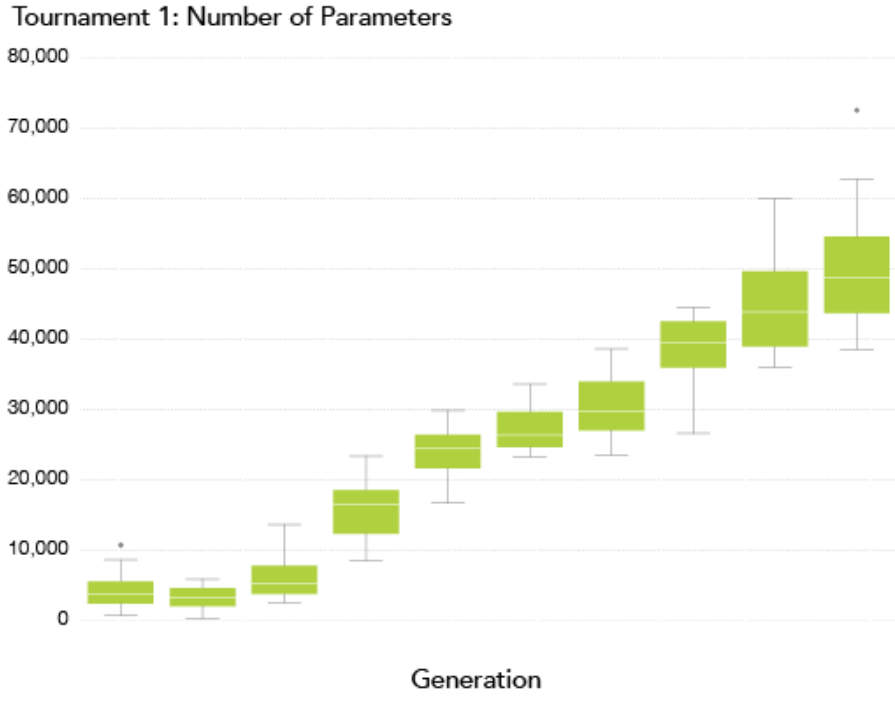


Figure 4: Average Validation Accuracy in Tournament 1

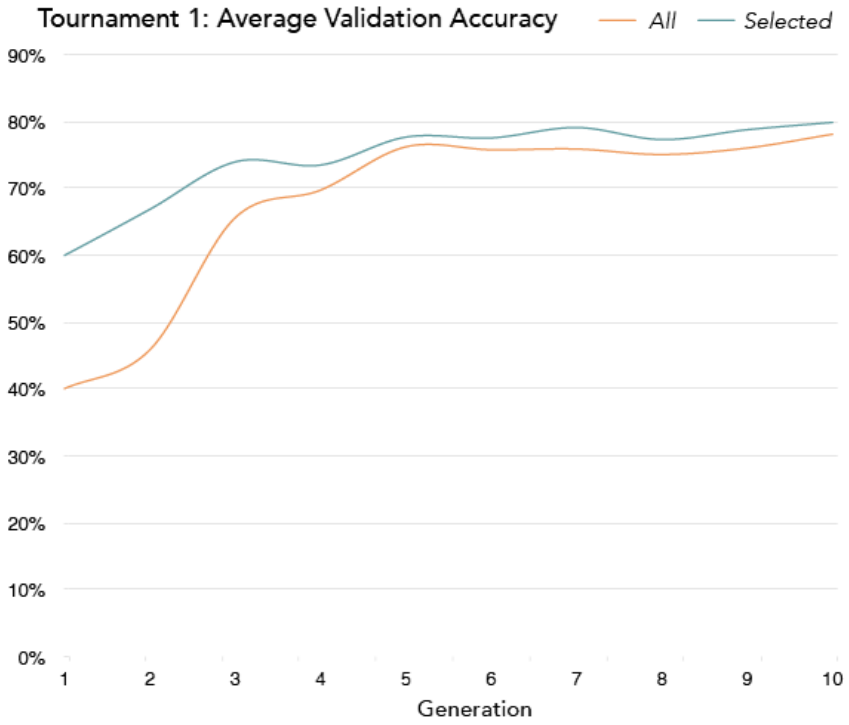


Figure 5: Average Model Parameters in Tournament 1

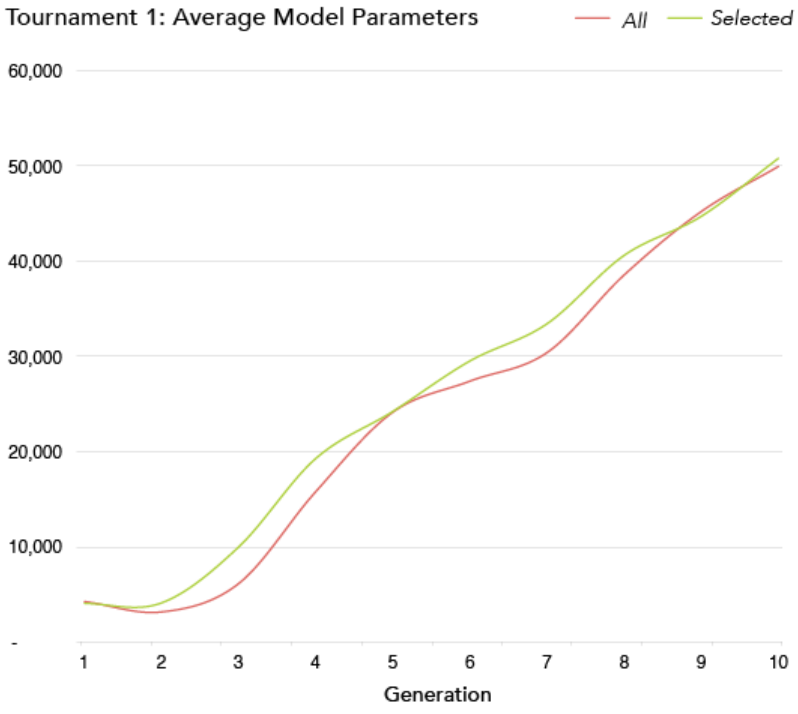


Figure 6: Validation Accuracy for All Generations in Tournament 2

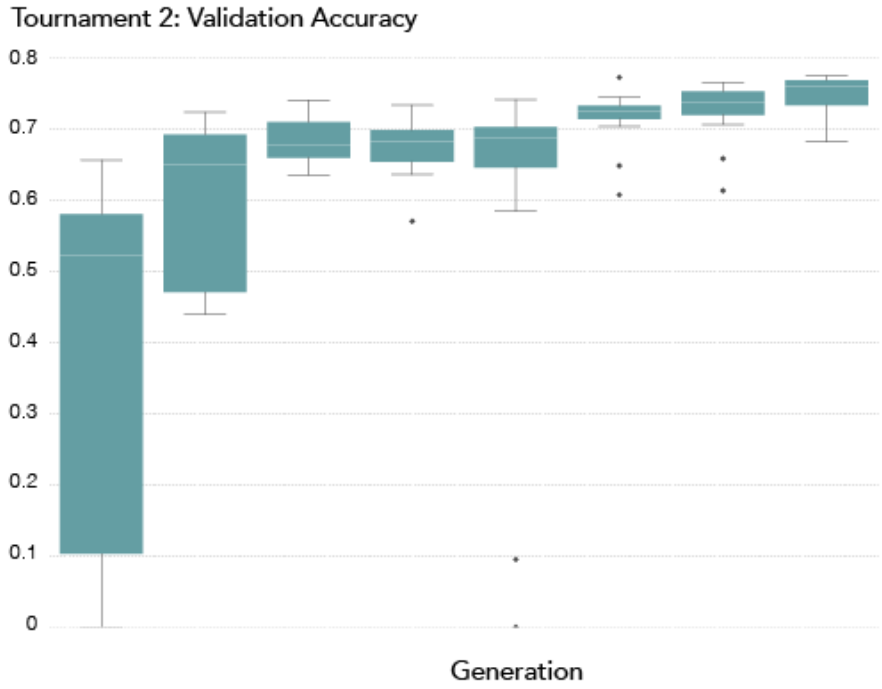


Figure 7: Number of Parameters for All Generations in Tournament 2

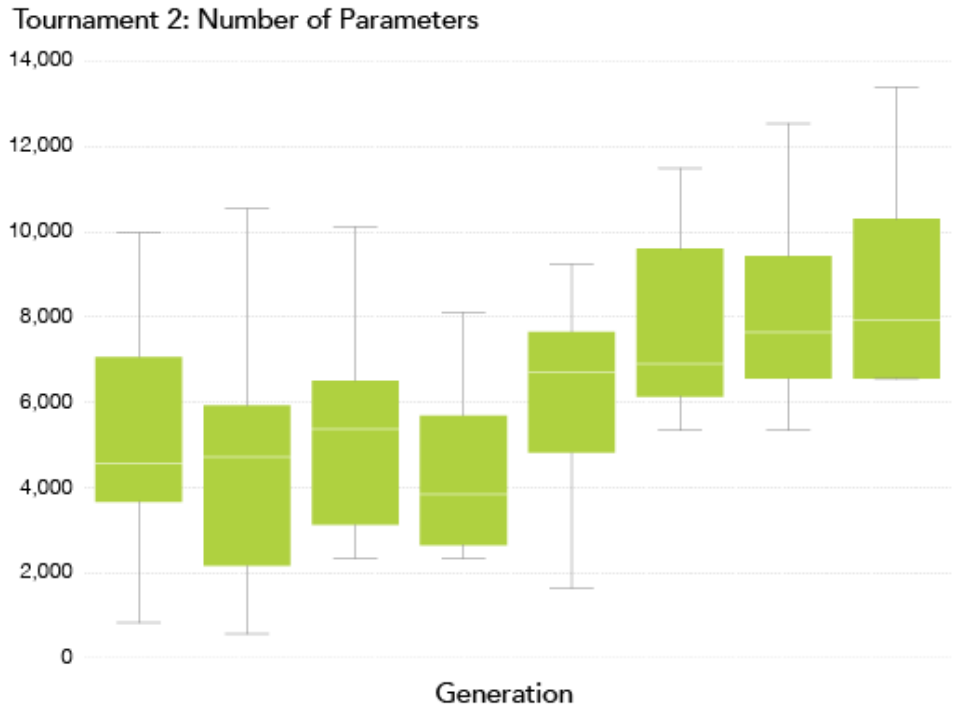


Figure 8: Average Validation Accuracy in Tournament 2

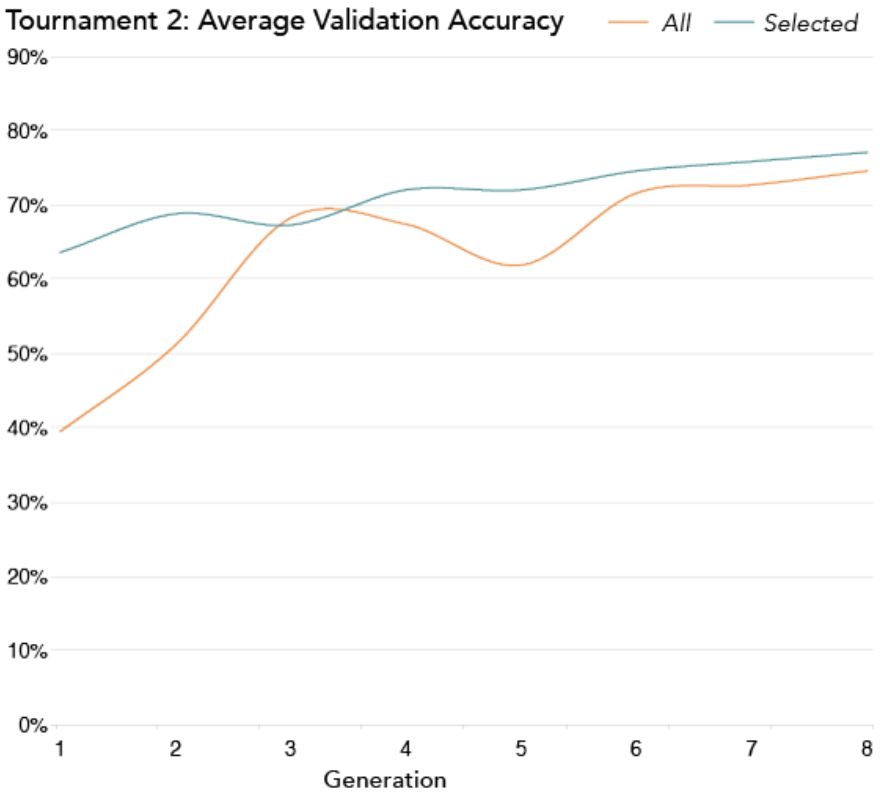


Figure 9: Average Model Parameters in Tournament 2

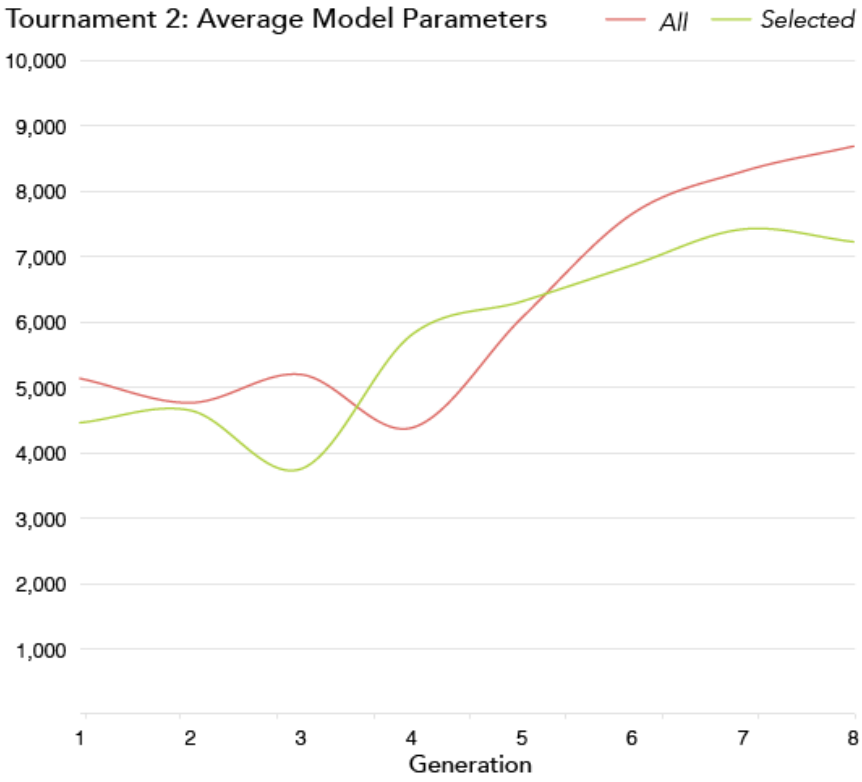


Figure 10: The Majority of Models Share the Same Ancestor after Generation 2

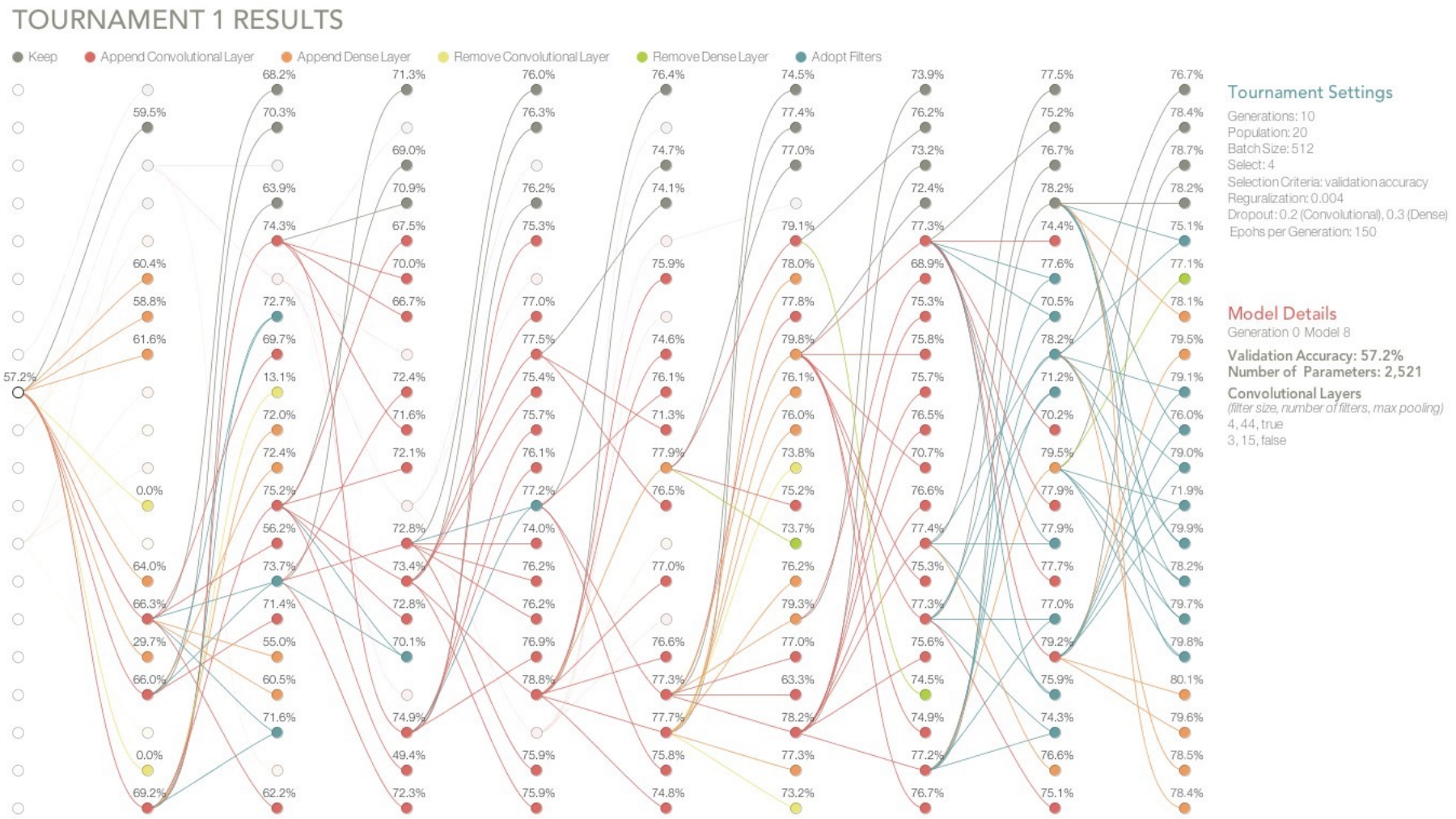


Figure 11: The Majority of Models Share the Same Ancestor after Generation 4

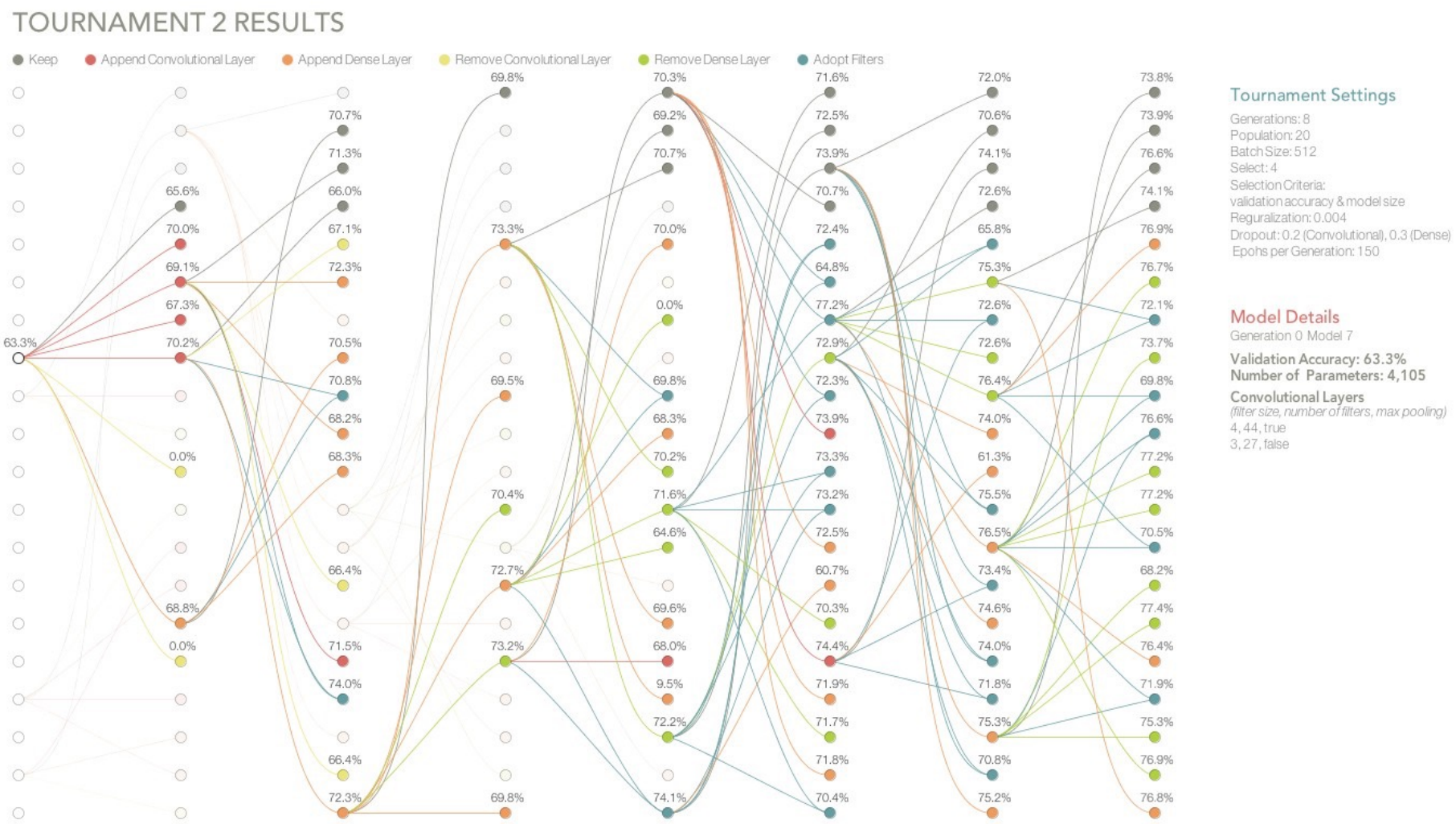


Figure 12: The Evolution of the Winning Model in Tournament 1

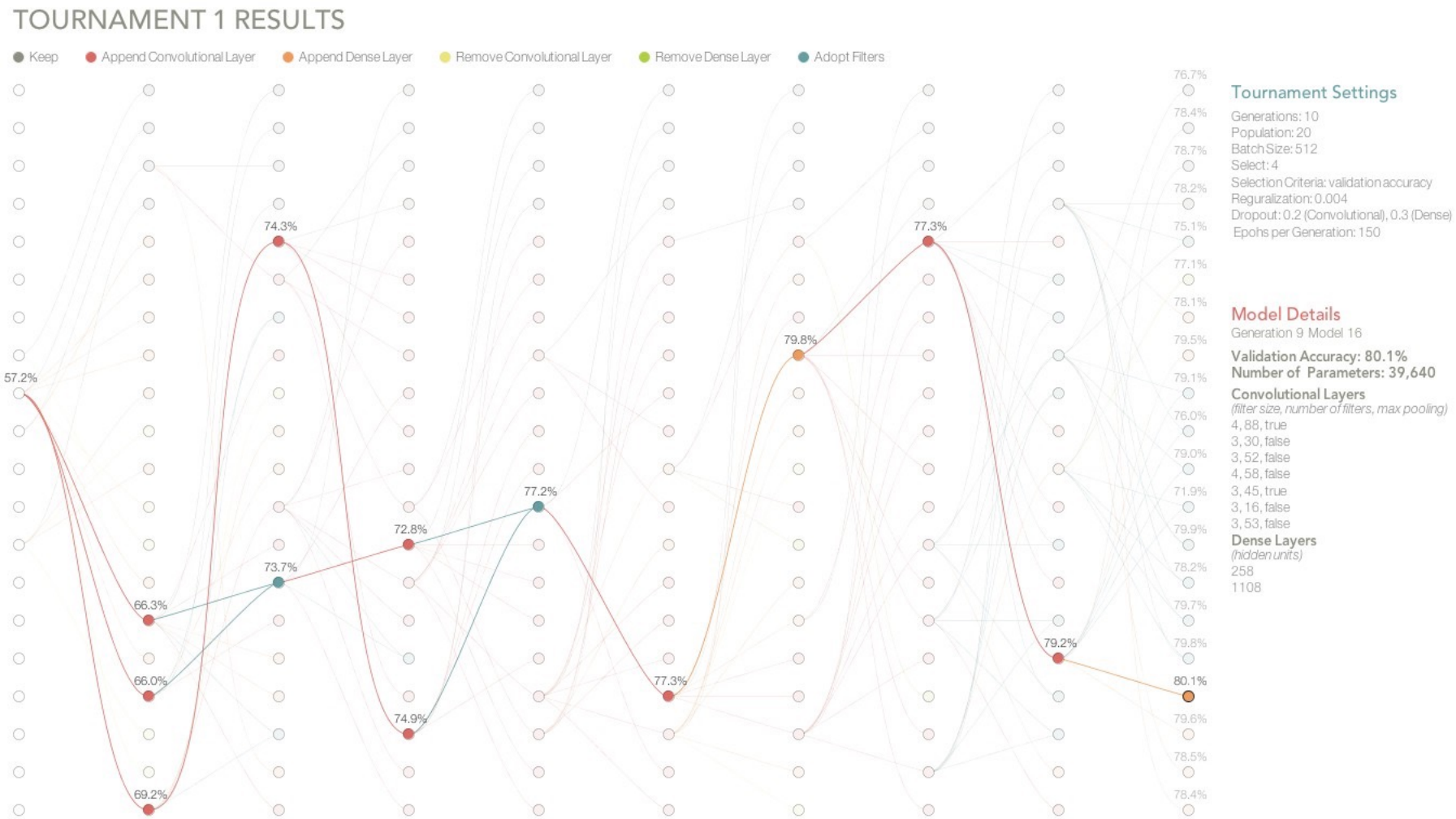
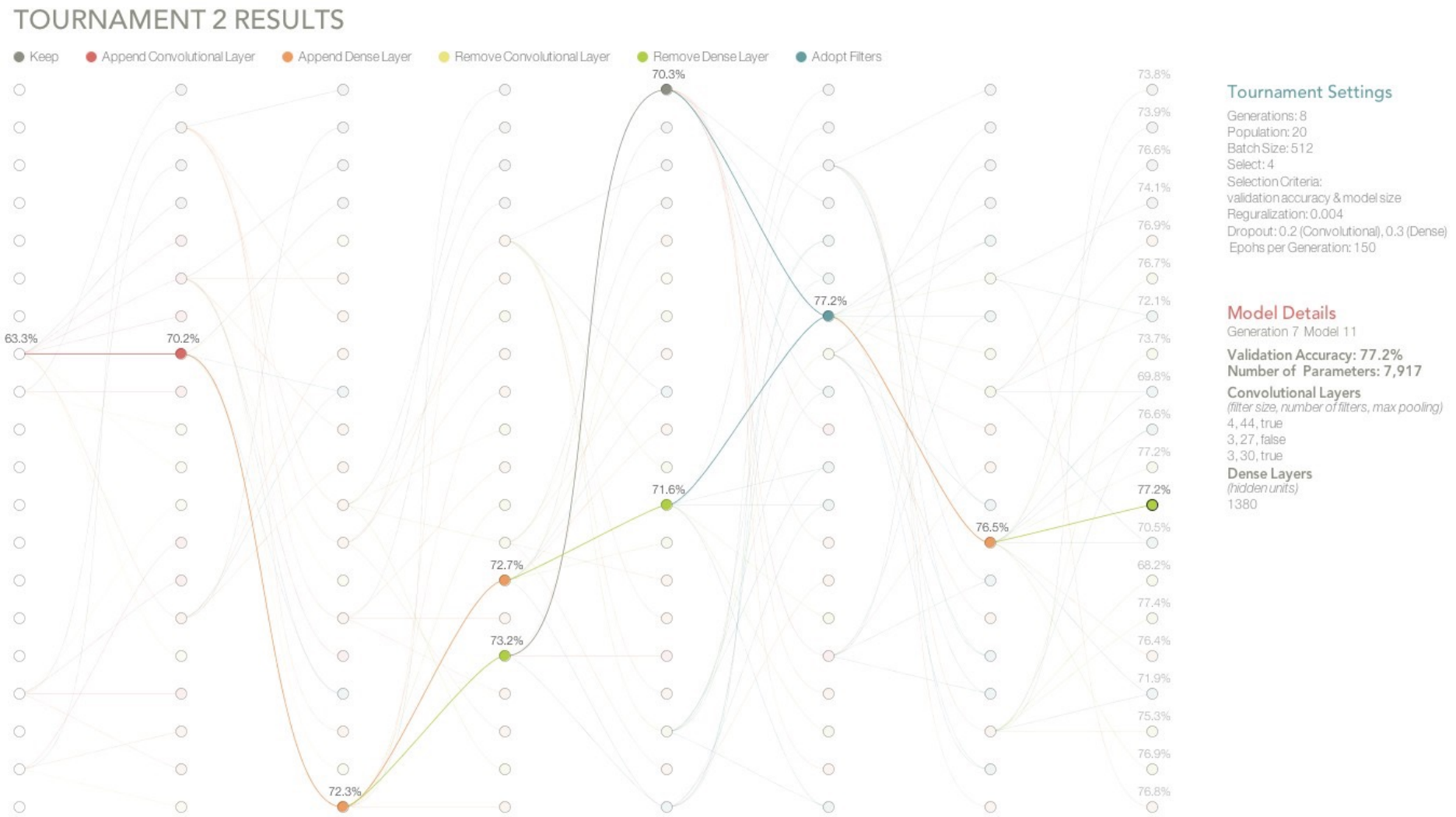


Figure 13: The Evolution of the Winning Model in Tournament 2



References

1. He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE International Conference on Computer Vision. 2015.
2. Dai, Jifeng, Kaiming He, and Jian Sun. "Boxsup: Exploiting bounding boxes to supervise convolutional networks for semantic segmentation." Proceedings of the IEEE International Conference on Computer Vision. 2015.
3. Bengio, Yoshua. "Practical recommendations for gradient-based training of deep architectures." Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 437-478.
4. Hong, Seunghoon, et al. "Learning transferrable knowledge for semantic segmentation with deep convolutional neural network." arXiv preprint arXiv: 1512.07928 (2015).
5. Sharif Razavian, Ali, et al. "CNN features off-the-shelf: an astounding baseline for recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. 2014.
6. Jung, Jae-Yoon, and James A. Reggia. The automated design of artificial neural networks using evolutionary computation., Success in Evolutionary Computation. Springer Berlin Heidelberg, 2008. 19-41.
7. Nadi, A., S. S. Tayarani-Bathaie, and R. Safabakhsh. Evolution of neural network architecture and weights using mutation based genetic algorithm. Computer Conference, 2009. CSICC 2009. 14th International CSI. IEEE, 2009.
8. NourAshrafoddin, Naser, Ali R. Vahdat, and Mohammad Mehdi Ebadzadeh. Automatic design of modular neural networks using genetic programming., International Conference on Artificial Neural Networks. Springer Berlin Heidelberg, 2007.
9. Cheung, Brian W. Hybrid Evolution of Convolutional Neural Networks. Diss. COOPER UNION, 2011.
10. Zhining, You, and Pu Yunming. The Genetic Convolutional Neural Network Model Based on Random Sample." International Journal of u-and e-Service, Science and Technology 8.11 (2015): 317-326.
11. Fiszelew, A., et al."Automatic generation of neural networks based on genetic algorithms." Revista Eletrônica de Sistemas de Informação 2.1 (2003): 1-7.

12. Vonk, E., Lakhmi C. Jain, and Ray P. Johnson. Automatic generation of neural network architecture using evolutionary computation. Vol. 14. World Scientific, 1997.
13. NeuralNetwork. <https://github.com/jobeland/GeneticAlgorithm/tree/master/NeuralNetwork.GeneticAlgorithm>
14. evo-neural-network-agents. <https://github.com/lagodiuk/evo-neural-network-agents>
15. GANN. <https://github.com/anelachan/GANN>
16. <https://github.com/AbhishekGhosh/Neural-Genetic-Algorithm>
17. EasyTensorflow. <https://github.com/calvinschmdt/EasyTensorflow>
18. TensorFlow Tutorials, Convolutional Neural Networks. <https://www.tensorflow.org/versions/r0.11/tutorials/deep_cnn/index.html>
19. Graham, Benjamin. "Fractional max-pooling." arXiv preprint arXiv:1412.6071 (2014).
20. Classification datasets results. <http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html>
21. Springenberg, Jost Tobias, et al. "Striving for simplicity: The all convolutional net." arXiv preprint arXiv:1412.6806 (2014).