# AUTOMATED MACHINE LEARNING TO GENERATE OPTIMAL NEURAL ARCHITECTURES

*Raül Pérez i Gonzalo (s192634)* *Anshul Chauhan (s192164)* *Meenal Malik (s190304)*
Wenjing Dai (s192897) Sushanth Kapisthalam (s190031)

## ABSTRACT

Neural Networks were inspired by the way our brain processes information based on its neurons. They are viewed as exceptionally ground-breaking, just as effective models to train, identify and predict future outcomes. However, at the same time, they are also very complex to design and implement. The idea behind this project is to reduce the human endeavors in designing neural networks which can, in place, be designed by the machines itself. In this paper, a controller, RNN, is trained using **Policy Gradient** algorithm to structure model descriptions of the neural networks, i.e., by reinforcing the RNN to maximize the accuracy (reward) obtained by the child network. Starting from the scratch, Half Moon dataset was used for the training of the child network to design optimal neural architectures which can compete with the best, already existing neural networks for this dataset. Furthermore, we actualized our model for a limited number of hyperparameters, that is, the number of hidden units, number of hidden layers and the non-linearities. These hyperparameters have been tuned in a way that best possible neural architecture can be accomplished. Code can be found on https://github.com/RualPerez/AutoML.

***Index Terms*—** Policy gradient, Recurrent neural network, Child network, Hyperparameters, Controller.

## 1. INTRODUCTION

In the field of deep neural networks, we have witnessed rapid development in the last decade. It has successfully shown an enormous evolution in many applications like computer games, robots, self-driving cars, image and speech recognition, machine translation, etc. Especially, when it comes to deep reinforcement learning, one can really aspire to reach a new level of amazing advances in artificial intelligence (AI).

Following this, we are addressing a problem of designing complex neural networks for different datasets which can itself be a very time-consuming task because of its complexity. Besides, hyper-parameter optimization requires preexperience based on a large number of experiments. To address this issue, we propose AutoML, which can automatically design the structure and implement hyperparameter optimization in the child network from a predefined search space. The main point is to use a recurrent neural network (LSTM) as a controller to generate an optimal child network [1]. To train this network, we are using a policy gradient method and the validation accuracy from a child network as a reward signal to update the controller parameters. In our experiments, we trained the controller to fine-tune the number of layers, the number of hidden units in each layer and the non-linearities (activation functions).

Our experiments reveal that neural networks built by the above-mentioned method can design optimal models from scratch. Also, this can be asserted that these models can outperform human-designed models, as the search space is enormously big which is far exceeding the limits of a classic grid search strategy.

## 2. RELATED WORK

Hyperparameter optimization is a challenging task that explores the optimal structure of the neural network automatically. Zoph and Barret et al. [2] proposed a neural architecture search algorithm by using reinforcement learning, which predicts network hyperparameters (filter height, filter width, stride height, stride width and number of filters) and gives the optimal neural network architecture based on search space. The model is built from these hyperparameters stochastically.

The hyperparameters are considered as actions and are produced repeatedly up to a certain number of times and then the model is built, trained and the validation accuracy is calculated which is taken as a reward. The controller parameters are updated based on this reward until policy gradient exploits the same neural architectures, that is, the action probabilities clearly induce to pick the same hyperparameters at each rollout.

Related to our research, the other way of doing the same is using the idea of learning to learn, known as meta-learning, is demonstrated in paper "A meta learning-based framework for automated selection and hyperparameter tuning for ma-

chine learning algorithms" [3]. The idea is to compare and update the knowledge base of a framework which contains information of meta features for all the processed datasets along with the corresponding performance of different classifiers and their tuned parameters.

In terms of having fast and inexpensive approach for the

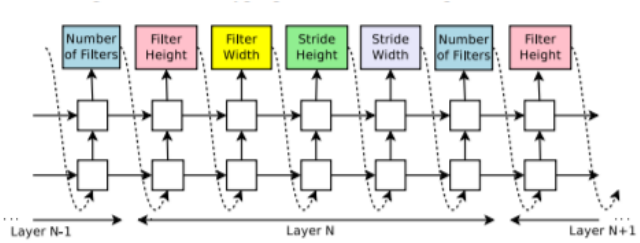


**Fig. 1**. Layer model architecture

optimal neural architectures, the paper, "Efficient neural architecture search via parameter sharing" [1], has shown some striking results. The idea behind the research is to use graph algorithms for searching an optimal subgraph within a large computational graph. Basically, by using this policy, a controller is trained to select a subgraph that maximizes the expected reward as well as to minimize the cross entropy loss. Another catch is to use the parameter sharing technique between the child models which makes it inexpensive in terms of GPU hours than almost all of the existing model design techniques.

Basic algorithmic understanding for the implementation of policy gradient algorithms has been drawn from one of the finest articles, "An introduction to Policy Gradients with Cartpole and Doom" [4]. This article is composed of mathematical understanding behind the policy gradient implementation, using some game examples like cartpole and doom. Apart from this, the article also talks about some other policy based methods and refers to various value based reinforcement learning techniques.

## 3. METHODOLOGY

In this section, we explain how our controller RNN is choosing the best blend of hyperparameters so as to define our child network model - a feed forward neural network (FFNN) model in our case. Following this, we demonstrate that the RNN model can be trained by using the REINFORCE policy gradient method which maximizes the accuracy of FFNN model architecture.

### 3.1. Use of Recurrent neural network as a controller

In order to find the best combination of hyperparameters from the search space, LSTM (Long short term memory), an exceptional kind of RNN, is used as a controller.

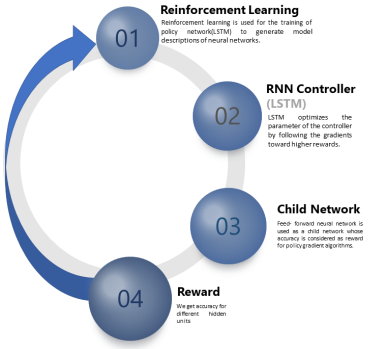


**Fig. 2**. Model structure

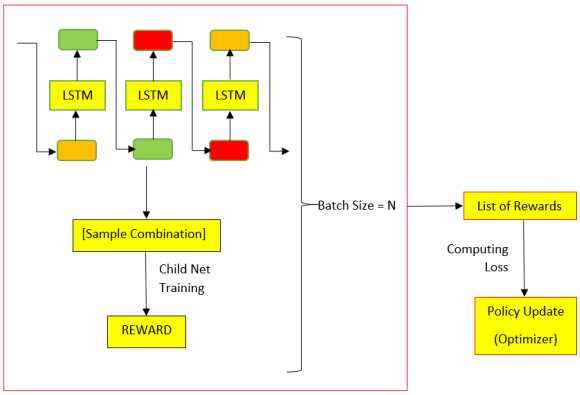


**Fig. 3**. Implementation of network using hidden units as one hyperparameter

Given the hidden units, layers and activation functions as our hyperparameter space, we have a hyperparameter server containing all the possible combinations for the selected hyperparameters. Let's say, for instance, we want to design a feed-forward neural network with the optimization of three hyperparameters, that is, the hidden units, the hidden layers and the activation functions. For this, we need to search for all different combinations of all the parameters in the search space and each LSTM iteration will specify us a childnet hyperparameter. More precisely, the LSTM controller takes a zero vector initializer as input and gives an output in terms of probabilities - thanks to a softmax stuck - which hyperparameter should be picked next. But in order to explore more, rather than taking the value with the highest probability, a random hyperparameter will be selected based on the softmax probabilities and then fed into the next time step as the input.

This process will carry forward up to some predetermined number of layers for LSTM cell or can stop at an early stage if 'EOS' (End Of Sequence, a possible hyperparameter/action) got selected in between. This process eventually ends on having the best combination of hidden units with their optimal number of hidden layers and activation functions, assuming

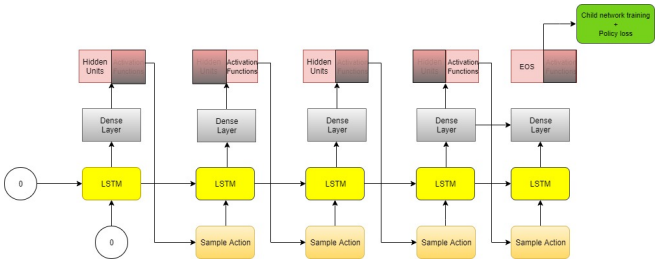


**Fig. 4**. Cluster of Recurrent Nodes, using 2 hyperparameters

the global maximum is reached.
As can be seen from the flow diagram of the policy gradient algorithm in figure 4, the policy gradient is structured in the form of a recurrent neural network where each memory cell has an input in the form of a vector consisting of 2 lists: the number of hidden units and the type of activation function to be used for each layer. The lists will be switched depending on the nature of the iteration step, so, for an odd number, the number of hidden units will be fed as the input and for an even number, the type of the activation function chosen is fed as the input to the memory cell of the recurrent neural network. Then, the sampled combination is picked for each episode and that particular childNet combination is fed to obtain a reward. Various childNet architectures are obtained in the controller's forward pass, then trained individually and their validation accuracies are recorded and sent to the policy gradient network to compute the controller's backward pass. So, all these steps are computed in one go (mini-batch gradient descent), except the childNet training. Also, these parameters have quite an impact on the behavior of the network performance, subsequently, giving an accurate idea about the state-of-the-art technique being used.

### 3.2. Reinforce Policy Gradient Approach

This approach is used to train the LSTM neural network (which is our policy network) to get some optimal actions for selecting the combinations of hyperparameters.

#### *3.2.1. Step-by-step Explanation*

- Let the Neural Network policy learn and train the childnet several times (equal to the batch size) and each step would compute the gradients that would make the chosen action even more likely, but of course, these are still not the final gradients yet, so they are not applied as it is.
- After running the above algorithm for several episodes, each action's score is computed by looking at the gradient calculated in the previous step and thus assigning a probability to each action.
- Looking at the action's score computed in the last step, if the score turns out to be positive, it means the action was good and the gradients corresponding to that action is applied to make the action even more likely to be chosen in the future. However, if the score for a particular action turns out to be negative, it means the action was not good enough and thus, to reduce the likelihood of this action, the corresponding gradients are inversed.
- The idea is basically to multiply each gradient vector, computed in the first step, by the corresponding action's score.
- The final step is to compute the mean of all the resulting gradient vectors and use it to perform a Gradient Descent step.

#### *3.2.2. Mathematical Approach*

This process has been taken place in two steps:

- By measuring the quality of our policy network ($\pi$) using policy score function ($J(\theta)$)
- By the use of policy gradient ascent to find the best parameter ($\theta$) which improves our policy ($\pi$).

In simple words, $J(\theta)$ will tell how good our policy network is and policy gradient ascent will help us to find the best policy parameters in order to maximize the reward from good actions.

**What is $J(\theta)$ ?**

- Calculates the expected reward from policy network
- Depends on the environment and objective that we have

Starting from the very first episode, we will calculate $J(\theta)$ as:

$$J(\theta) = E_\pi[G_1 = R_1 + \gamma * R_2 + \gamma * R_3 + .... + \gamma * R_n] = E_\pi(V(S_1))$$

where $G_1$ is Cumulative discounted reward starting at initial state, $\gamma$ is discount rate and $E_\pi(V(S_1))$ is value of first state.

$$\text{As, } J_{avg}(\theta) = E_\pi(V(S)) = \sum d(s)V(s).$$

This expression can be rewritten as :

$$J_{avg}(\theta) = E_\pi(r) = \sum d(s) \sum \pi\theta(s,a)R_s^a$$

where $\sum d(s)$ is state distribution and the remaining expression is the probability of taking an action a from state s under policy $\pi$.

To have an optimal policy, we need to maximize $G_1$, meaning that we have to find $\theta$ which maximizes this function. As reward is non-differentiable, therefore, we try to find gradient concerning $\theta$ in function J($\theta$) and use the updated method as given below:

$$\theta \leftarrow \theta + \alpha \nabla_\theta (J(\theta)).$$

When using gradient ascent on the function, one thing to heed is that the performance depends both on the action selections and the distribution of states. But, as the environment function is still unknown, so the question is how do we calculate the effect of policy parameters on the state distribution. To counter this problem, we used a policy gradient theorem which does not consider differentiation of state distributions. Hence, by ignoring this, the expression can be written as:

$$\nabla_\theta (J(\theta)) = \nabla_\theta \sum_{t=1}^{T} (\log \pi(T;\theta)) R(T).$$

Now, for the number of different architectures, $m$, that our controller creates and for some different hyperparameters, $T$, that our controller needs to predict to design an architecture, we use the REINFORCE rule from neural architecture search paper [2], which is an extension rule from the above equation, and is given as:

$$\nabla_\theta (J(\theta)) = \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_t | a_{(t-1):1}; \theta) R_k.$$

Next, in order to handle high variance in estimates, we are using a baseline function (as mentioned in [2]) which is an exponential moving average of previous architecture accuracies which lead to our final unbiased in expectation of loss function for the policy gradient algorithm as given below:

$$\nabla_\theta (J(\theta)) = \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_t | a_{(t-1):1}; \theta)(R_k - b).$$

Lastly, an entropy term has been added as a regularization tool to increase exploration, then our Loss function has been defined as:

$$\mathcal{L} = E[R|\theta] \approx \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \log p_\theta(a_t | a_{(t-1):1})(R_k - b)$$

$$-\lambda \sum_{k=1}^{m} \sum_{t=1}^{T} p_\theta(a_t) \log p_\theta(a_t),$$

where:
$E[R|\theta]$ is the expectation of a reward, R, given a policy, $\theta$, which is approximated by taking the log of the probability of the action multiplied with their reward and then taking the difference of this and the entropy, $\lambda$, to improve the exploration space of this algorithm.

## 4. EXPERIMENTS

We know that the time it takes to implement the algorithms on datasets is totally dependent on the size and complexity of the dataset. In our case, algorithmic complexity is already high and it needs high computational power to perform a task in a limited amount of time. So, as mentioned above, we are setting up our experiments with one of the simplest datasets, "Half Moon Data set", in order to visualize the learning capacity of our model and how accurate our model can be with the tuning of selected parameters. In this race of finding an optimal architecture, a separate validation data set is used to compute the reward for the policy gradient algorithm. Further details on experimentation and results will be shown as follows.

**Dataset :** From Half moon dataset, a total of 1000 samples have been taken into consideration, out of which, a model is getting trained on 700 samples and validated on 200 samples. The remaining 100 samples are used as the test set which is computed only once for the network that pulls off the best result of the validation data set. In order to deal with a non-trivial dataset and to reduce overfitting, which is possible because of our small-sized dataset and the memorizing ability of the neural network, the noise of 0.2 has been taken into account.

**Search space :** Our search space is comprised of a limited number of values for four different hyperparameters. For every linear layer, the RNN controller will choose different combinations for hidden units in $[1, 2, 4, 8, 16, 32]$, corresponding to the number of hidden layers (cannot be more than five). Activation functions will be chosen between ReLU and TanH.

**Training analysis :** It was implemented using multilayered LSTM cells in our policy network with ADAM optimizer in which the learning rate is considered as 0.01 which is exactly similar for the child network, the one chosen by RNN architecture. However, the child network is trained and validated over 1000 episodes with the principle of early stopping, whereas the reward used for updating the controller is the maximum validation accuracy of the last 5 epochs cubed [2]. Whereas as in policy training, number of episodes used are 500 and for the baseline function the decay of 0.9 is used to actualize it.

**Results :** The result shows the performance of the policy network while creating an optimal child network on half moon dataset.

Figure 5 shows the training accuracy and loss, where the network gives a very low accuracy and high loss initially when only one hidden unit, hidden layer and an activation function are considered as hyperparameters. When

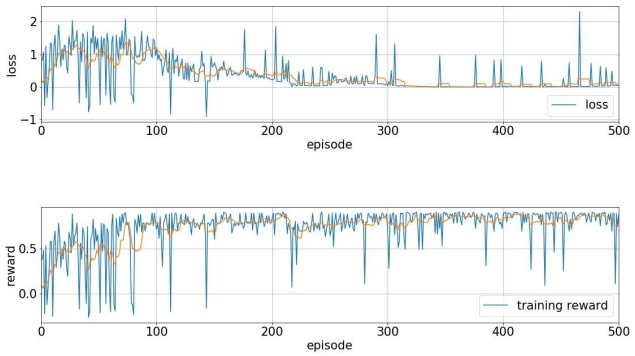


**Fig. 5**. Training accuracy (reward) and Loss of the policy network (controller). The orange curve is the moving average curve of the last 20 episodes.

the network started training with different combinations of hyperparameters the training accuracy improved and started converging after 500 epochs.

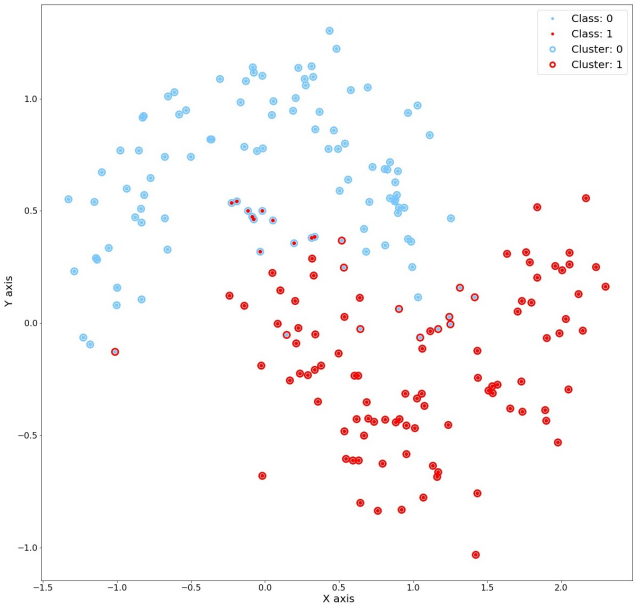


**Fig. 6**. The classification results of the whole half moon data set using a childNet architecture obtained by the controller.

Since the accuracy is not 100 %, there was some mis-classification of some data points. There are two classes, red and blue, which are categorized correctly and the mis-classified data points are categorized incorrectly as shown in Figure 6. Note that these mis-classifications are hard to ignore due to the inherent noise of the dataset.

To prove that the training policy obtains optimal childNet architectures, Figure 7 shows the distribution accuracy (histogram) of different hyperparameter combinations of a childNet architecture - using 5000 random hyperparameter combi-

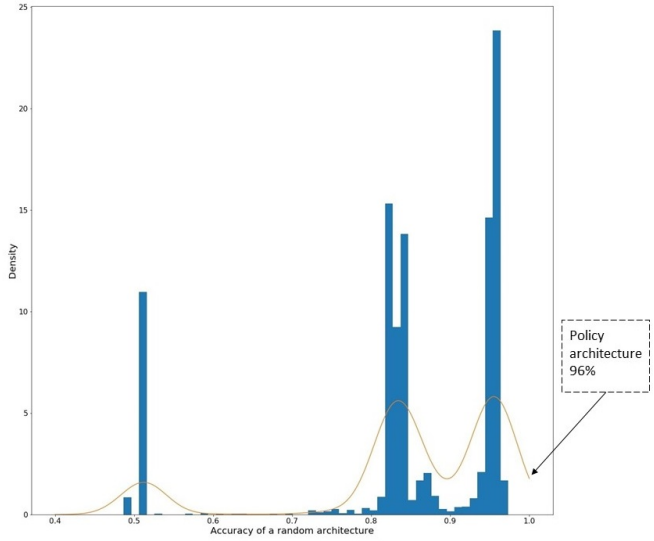


**Fig. 7**. Distribution accuracy of all the possible hyperparameter combinations of the childNet. It has been computed taking 5000 random combinations. The orange curve indicates a Gaussian kernel of the original histogram.

nations. The figure shows three clear peaks, for instance, the first one (peak at 0.5 accuracy) includes the childNet architectures for many layers with 1 as hidden unit. As the policy algorithm generates a stochastic controller, the trained controller can generate various networks, once it has been trained. The power of our AutoML algorithm is obviously commendable as the trained controller generates architectures only in the highest peak. For instance, the one which was used in Figure 6 project was at 96.04% test-accuracy and it consisted of three hidden layers of 8, 4, 4 hidden units and Tanh, ReLU as activation functions.

## 5. CONCLUSION

The primary purpose of this report is to find the best possible neural architecture for the Half-Moon dataset which can compete with the performance of the existing ones. The idea of using a recurrent neural network (RNN) as a controller to generate models has proven that it works efficiently on small datasets and can be implemented on larger datasets as well in a similar manner, by considering more hyperparameters which can give an equally good accuracy on child networks. The only issue that can be faced while considering a large number of hyperparameters on a bigger dataset is the processing and training time of the network. The main benefit of using RNN controller is to make our method as flexible as possible in order to make it search in a variable-length architecture space. The accuracy obtained from our architecture is quite promising, however the accuracy might vary for larger datasets which can be optimised as mentioned in the

future scope of this project. Automated architecture search of our AutoML model addresses the problem of finding a well-performing architecture of a deep neural network without evaluating different architectures manually.

## 6. FUTURE SCOPE

As the domain of deep reinforcement learning is so vast and expandable, there are so many things to explore and possible upbringings of its applications. AutoML is just a small part of this great subject that can be developed up to a further extent.

Right now, we are tuning some limited hyperparameters, however, we can follow up by adding the new ones in further research and also try to explore and find the efficient algorithms which can be more inexpensive and fruitful.

Also our dataset is not that enormous and complex, we would like to advance with a more complex dataset like MNIST or even CIFAR-10. However, training on these datasets require high computation power and high performance GPU for training.

Till now, we were more focused on following neural architecture research paper [2] due to time constraints but for further enhancements, we can explore the ways of graph theory for better proficiency (as mentioned in paper [1]) and embed this with our existing work.

The horizon of AutoML is not just limited to build architectures but it can be expanded into further ways. That being said, "By 2025, 50% of data scientist activities will be automated by AI, easing the acute talent shortage" by Gartner in "How Augmented Machine Learning Is Democratizing Data Science" explains thoroughly about the use of AutoML in the data science field.

## 7. ACKNOWLEDGEMENT

We would very much like to thank Professor Ole Winther for his guidance during the lectures and special thanks to the teaching assistants Nicklas Andreas Hansen, Alexander Rosenberg Johansen and Peter Ebert Christensen for their constant support throughout the course work and their phenomenal assistance during the project.

## 8. REFERENCES

[1] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean, "Efficient neural architecture search via parameter sharing," *arXiv preprint arXiv:1802.03268*, 2018.

[2] Barret Zoph and Quoc V Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[3] Mohamed Maher and Sherif Sakr, "A meta learning-based framework for automated selection and hyperparameter tuning for machine learning algorithms (2019)," *Advances in Database Technology-EDBT: 22nd International Conference on Extending Database Technology, Lisbon, Portugal, March 26-29*, 2019.

[4] Thomas Simonini, "An introduction to policy gradients with cartpole and doom," 2018.