Neural Architecture Search for Conditional Dependencies

Romain Égele
romain.egele@polytechnique.edu

Jérémy Gozlan
jeremy.gozlan@polytechnique.edu

Paul Jacob
paul.jacob@polytechnique.edu

***Abstract*—For a few years artificial intelligence algorithms have demonstrated outstanding capabilities for a wide range of industrial applications. In the family of AI algorithms, neural networks has recently shown particularly impressive performances on a variety of problems related to speech, images, videos or natural language. However, these methods rarely benefit from possible dependencies existing between the different predicted variables. Furthermore, the knowledge learned by neural networks is often hard to interpret as a human. Therefore, using neural architecture search to automatically find these dependencies can benefit both the performance and human-understanding of predictions.**

I. Introduction

Neural architecture search (NAS) was introduced by Zoph and Le (2016) with a reinforcement learning (RL) approach based on a distributed vanilla policy gradient (VPG). In a following work (Zoph et al.; 2018), they switched the VPG for proximal policy optimisation (PPO, Schulman et al. (2017)) which helped them to reduce the use of 800 GPUs over 28 days with VPG down to 500 GPUs over 4 days. Even with this improvement, the amount of resources used is huge. Furthermore, this field of research shows difficulties to find its flagship algorithm. The main works regroup reinforcement learning (RL), Bayesian, gradient or genetic based algorithms (Liu et al.; 2018; Chen et al.; 2019; Klein et al.; 2016).

This work focuses on:

- Showing the benefits of using conditional dependencies in the architecture of a neural network (i.e., reducing the number of parameters in a model, improving generalisation abilities).
- Showing the feasibility of an automatic search for conditional dependencies with different computational scales (i.e., local machine and clusters).
- Comparing the performance of a classic Bayesian-based search with a reinforcement learning-based search on the given problem.
- Providing a ready to go *Python* package[1].

II. Background and Related Work

Neural architecture search (NAS) is a subset of problems included in the so called hyper-parameter search (HPS) problem. The particularity of NAS is to address problems with a search space of categorical-nominal (i.e., discrete without ordering) dimensions. The NAS community has provided many open source resources for different algorithms. One of these software, *DeepHyper* [2], interested us because of its *Keras*-like API to describe a search space of neural networks as well as the different algorithms available.

A. Conditional Dependencies and Multi-Outputs Prediction

Let us define a tuple (x, y) where $x \in \mathbb{R}^{N_x}, y \in \mathbb{R}^{N_y}$ are respectively called input and output. Initially the problem is defined such as the j^{th} output is predicted only given x which make us interpret the output such as $P(Y_j = y_j|x)$. This formulation is lacking power in the sens that it does not take advantage of predicting some outputs before others and using them in addition to the input to predict remaining outputs. Indeed, some of the outputs can induce some conditional dependencies to others, and bring important information.

Formally speaking, in our case, we want the j^{th} output to be predicted using not only the input x, but also a subset Ω_j of the output set $\Omega = \{y_1, y_2..., y_{N_y}\}$. The prediction of the j^{th} output will correspond to $P(Y_j = y_j|x, \Omega_j)$, rather than $P(Y_j = y_j|x)$ for a direct feed-forward neural network.

Naturally, we want our network to be acyclic : two variables can't predict each other. Formally, a network will be cyclic if there is a list of output indexes $(j_1, j_2, ..., j_n)$ such that $j_1 = j_n$ and $j_{i+1} \in \Omega_i$ for each i. If our search algorithm selects a cyclic network, we will bring to him a penalty value r_{min}. See figure 1 for an example of a cyclic and acyclic architecture.

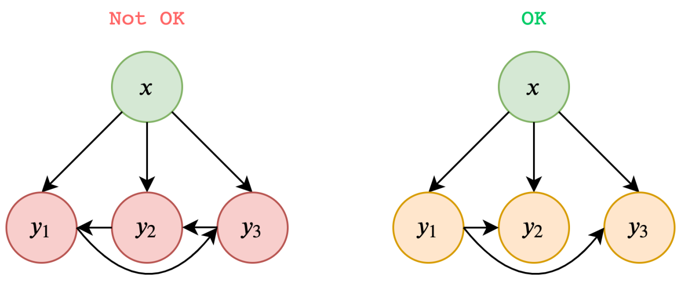


Figure 1: Possible conditional dependencies architectures

B. Recurrent Neural Network Architectures

In the neural architecture search problem we use a policy-based reinforcement learning methods where the surrogate

[1]Github: github.com/Deathn0t/NAS-CD

[2]Documentation of DeepHyper: deephyper.readthedocs.io

model for the policy is a neural network. In this framework the agent has to make a sequential set of decisions where an action is conditioning the next one (e.g., if I choose 10 neurons for the current layer how many should I choose for the next one). Hence, the use of a recurrent neural network (RNN) suits such a behaviour. In this part, two RNN architectures (see figure 2) used in our work will be introduced: gated recurrent unit networks (GRU) and long short-term memory networks (LSTM).

1) Long Short-Term Memory Network: Introduced by Hochreiter and Schmidhuber (1997), it is a kind of recurrent neural network (RNN) that has a feedback connections. It is useful for sequential data as it can learn long-term dependencies. This type of network is composed of sequential units where each of them include a cell state, a hidden state and three gates: input, output and forget.

- The gates regulate the flow of information, learn the important data in a sequence and forget the rest. This is made possible by internal operations between gates such as addition, multiplication, concatenation and activation functions.
- Cell states are used as transport gateways to pass the important information along the chain. They are the memories of the network.

2) Gated Recurrent Unit Network: Introduced by Chung et al. (2014), it is another kind of RNN that has similarities with LSTM. It differs by not having a cell state but only a hidden state and its update gate acts as both forget and input gates. Finally, the reset gate controls the importance of past information.

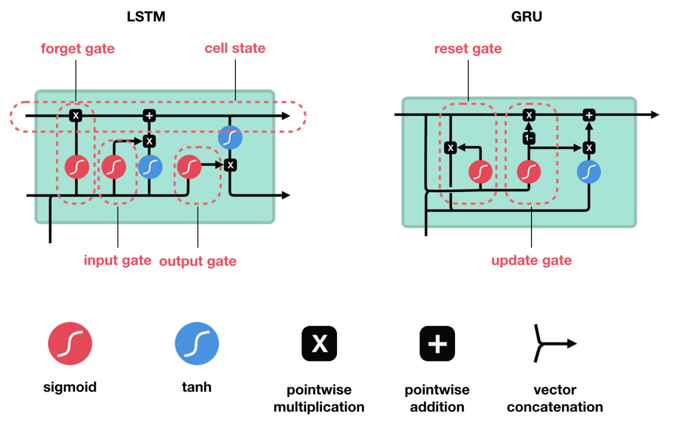


Figure 2: LSTM and GRU units

III. Reinforcement-Learning Framework

A. Environment

DeepHyper provides an *OpenAi* Gym environment called: `NeuralArchitectureVecEnv`. This environment is vectorized and asynchronous which allows the user to act with different environments in parallel at each step. Such environment can be of great importance when the reward is requiring a lot of time and computational resources. On a local laptop, neural networks are evaluated one after the other so this feature is not used, but it becomes necessary if search comes to be scaled up on a cluster.

This environment represents a search space of neural networks. Inside this object is defined a graph composed of two types of nodes: `VariableNode` and `ConstantNode`. A `VariableNode` represents a choice between multiple operations which are chosen given the action of the agenta. A `ConstantNode` represents a fixed operation in the neural network architecture which cannot change during the search. Thus, each `VariableNode` represents an action step of the agent.

The observation space is a discrete subset of $\mathbb{R}$, and it takes the value of last action done by the agent in the current episode (the first observation is bootstrapped with a value of 1, see figure 4 in the agent part on the left side to have an insight on how the agent RNN policy works). The action space is $[0 \;..\; a_{\max}]^n$ where $a_{\max} = \max_{\mathcal{N} \in \mathcal{S}}(|\mathcal{N}|)$ and $n = |\mathcal{S}|$. $\mathcal{S}$ represents the search space and $\mathcal{N}$ a variable node of this search space. Hence, $|\mathcal{S}|$ is the number of variable nodes in the search space $\mathcal{S}$ and $|\mathcal{N}|$ is the number of possible operations for the variable node $\mathcal{N}$. In our experiments we extended the observation space because we did not consider it as fullfilling the Markov property. To do so we make an observation a pair of two values: the first one being the one used by *DeepHyper* (i.e. the previous action) and the second one being the index of current action (i.e., from 1 to the number of variable nodes). The experimental part will show that this choice helped to improve consequently the performance of the search.

The reward is the learning performance of the neural network generated from the sequence of actions. Hence, the reward is different when it comes to a classification or a regression problem. In the classification case, we use the element-wise accuracy for the outputs, and we use the R^2 determination coefficient for the regression. The advantage of choosing these rewards is that in general both will be in the $[0, 1]$ range which will help us to fine tune the search algorithm for both problem in same time. As demonstrated in the figure 1 it is possible to have wrong models containing cycles. When DeepHyper is trying to instantiate these kind of models a cycle exception is raised and the model cannot be created then the environment returns a penalising reward of -1.

The figure 3 shows an example of search space. There are 3 variable nodes respectively labelled: XOR, AND, OR. Each of these nodes has four possible operations which are connections represented by the dashed lines (see appendix 1 for a code snippet):

- `XOR` node:

  Action 0 Connect to inputs.
  Action 1 Connect to inputs and `OR` node.
  Action 2 Connect to inputs and `AND` node.
  Action 3 Connect to inputs, `OR` node and `AND` node.

- `AND` node:

  Action 0 Connect to inputs.

Action 1 Connect to inputs and `XOR` node.
Action 2 Connect to inputs and `OR` node.
Action 3 Connect to inputs, `XOR` node and `OR` node.

- `OR` node:

Action 0 Connect to inputs.
Action 1 Connect to inputs and `XOR` node.
Action 2 Connect to inputs and `AND` node.
Action 3 Connect to inputs, `XOR` node and `AND` node.

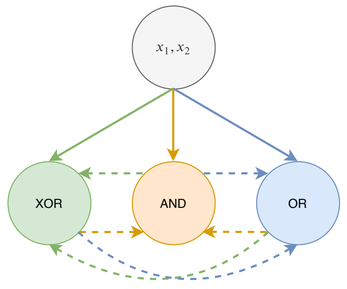


Figure 3: *XOR-AND-OR* Search Space

B. Agent

The software also provides a Deep-RL algorithm using a LSTM as actor (i.e., policy network) and critic in the agent context. The proximal policy optimization (PPO) algorithm is used for the optimisation. In our case, we also consider a stochastic, parametrized policy π_θ but we decided to try our own Deep-RL implementation based on a vanilla policy gradient (VPG) optimisation and implemented two policy networks: `PolicyLSTM` and `PolicyGRU`, respectively based on LSTM and GRU recurrent neural networks. The VPG algorithm consists in maximising the expected return $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$ where $R(\tau)$ is the finite-horizon undiscounted return. The gradient of this expectation is:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \underset{\tau \sim \pi_\theta}{\mathbb{E}}[R(\tau)] = \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]$$

where the expectation is estimated by using the empirical mean. Finally, a global diagram of the interaction between the agent and the environment is shown in figure 4.

IV. Experimental results

As it is not classic to use an RL algorithm for a black-box optimization problem we decided to use a baseline algorithm available in *DeepHyper*: asynchronous model-based search (AMBS). This method is based on Bayesian optimisation and tries to learn relationships between the inputs and the outputs through search space pruning to identify promising regions. Finding the correct neural network architecture may lead to a multitude of different networks being built and trained which may take an unfeasible time to compute for large problems. A advantage of AMBS is to prune the search space very quickly.

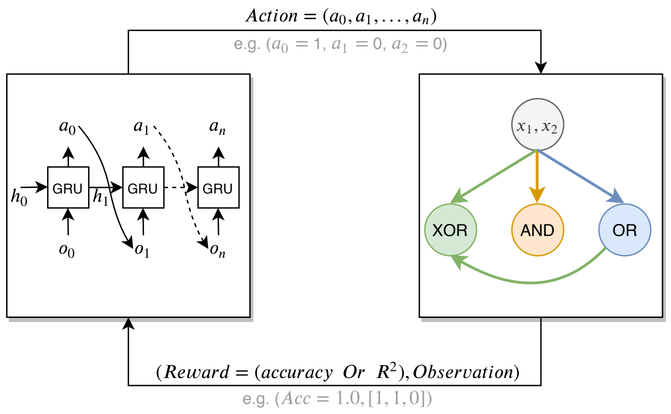
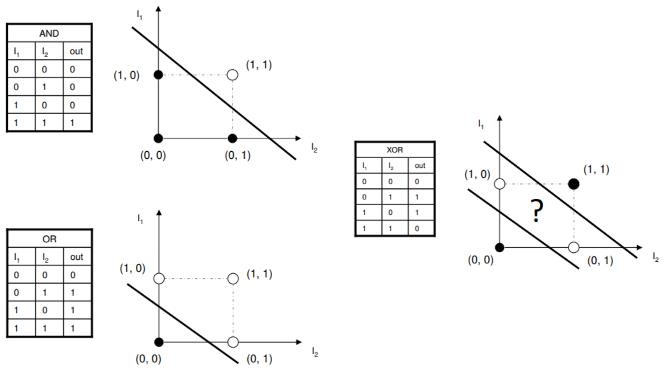

$Action = (a_0, a_1, \ldots, a_n)$
e.g. $(a_0 = 1, a_1 = 0, a_2 = 0)$
a_0
a_1
a_n
h_0
h_1
GRU
GRU
GRU
o_0
o_1
o_n
x_1, x_2
XOR
AND
OR
$(Reward = (accuracy\ Or\ R^2), Observation)$
e.g. $(Acc = 1.0, [1, 1, 0])$

Figure 4: RL Settings



AND		
I₁	I₂	out
0	0	0
0	1	0
1	0	0
1	1	1

OR		
I₁	I₂	out
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
I₁	I₂	out
0	0	0
0	1	1
1	0	1
1	1	0

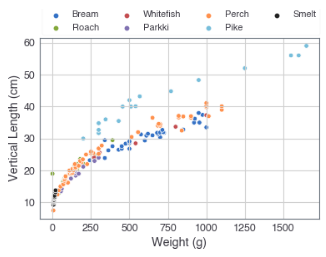
Figure 5: *XOR-AND-OR* Truth tables

A. XOR-AND-OR

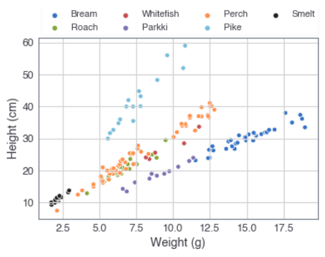
The *XOR-AND-OR* problem is a perfect to illustrate the problem of conditional label dependency. Taking a pair of logical elements, one neuron per logical output and direct links, the *AND* and *OR* can easily be predicted while the *XOR* cannot. Indeed, due to the fact that the *XOR* is not linearly separable (see figure 5), an another input must be passed to the *XOR* neuron for complete prediction. In a supervised learning task our dataset is composed of 4 elements where the inputs are pairs and outputs are triplets. Based on the `XOR-AND-OR` search space described previously the probability of finding a "good" network architecture is $3/16$ (only 48 architectures out of 64 have a dependency on the *XOR*, and only 12 of them are acyclic). Each generated NN is trained with the following hyper-parameters: batch size of 2, learning rate of 1.0, RMSProp as optimizer, 2500 epochs and an early stopping mechanism tracking the min value of the loss (i.e. mean squared error) with a patience of 5 epochs. Based on the size of our search space we ran the search for no more than 64 evaluations which is the threshold where an exhaustive search would become more efficient than the one we use. As expected, the networks with 100% accuracy had *XOR* with conditional dependencies on either the *AND*, *OR* or both. On the other hand, the other outputs did not have necessaries dependencies as they were from the start linearly separable.

B. FISH MEASUREMENTS

The *Fish* dataset comprises 160 records of 7 different species and their physical measurements. To illustrate this problem, we focused on the prediction of a fish vertical, horizontal, cross lengths as well as its height and width using its specie and weight as inputs. Because many of these features are intrinsically correlated (see figure 6a), a neural network taking advantage of conditional dependencies architecture can be designed to improve the prediction of hard characteristics (see figure 6b).



(a) Weight vs Vertical Length



(b) Weight vs Height

Figure 6: Visualisation of Fishes Data

However, finding the optimal dependencies to minimise the overall error is a difficult task when a large number of output variables are to be predicted (i.e. combinatorial optimization). One of the major interest in using *DeepHyper* for such kind of problems is the ability to only specify initial numbers of layers and possible dependencies in order to find viable networks architecture.

For the *Fish* dataset, we define a search space of conditionnal dependencies where each regressor (i.e., coloured circle on figure 9) is composed of a Concatenation layer, a Dense layer with 8 units and `ReLu` activation function, a Dense layer with 1 unit. The models will be evaluated for 200 epochs using the Adam optimizer with a learning rate of 0.01. Then, to evaluate results of searches we create a baseline feed forward neural network that is trained for 1,000 epochs and achieved a 0.969 R^2 score on the validation set. This NN consist of 3 dense layers each with 10 units and *ReLU* activation, and a dense output layer with 5 units.

As a baseline search method we used AMBS which we ran for 64 evaluations. Among these evaluated models, 32 were infeasible because of cycles in their architecture and 32 reached high validation R^2 scores from 0.87 up to 0.945. It is to note that some models had redundant or symmetric architecture which resulted in duplicated models and that the R^2 score is highly dependant on the initial network weights. Upon the comparison of the 32 feasible, similar patterns were found in the best performing networks which reflected the existence of optimal dependencies. Furthermore, analysing the dependencies for the prediction of all outputs, most of them had real life meanings.

For instance, the following dependencies were observed in many successful models (figure 9a) :

- The fish vertical length was dependant on the inputs alone.
- The fish cross section length was dependant on the data and prediction for the fish width, vertical and diagonal length.
- The fish width was dependant on the data and prediction for the fish cross section length.
- The fish height was dependant on the the data and most other predicted measurements.

Reducing the possible choices for each layer depending on the best performing trained models, another search space was defined which resulted in a higher proportion of feasible and highly performing models. Compared to the first search, 16 models were infeasible and almost all other models exceeded 0.91 R^2 scores with 4 different models reaching more than 0.95 (see figure 7). It is to note that the number of neurons per *Dense* layer during the search space was fixed to 8 and it could probably have been automatically searched to improve the performance or reduce the number of trainable parameters.

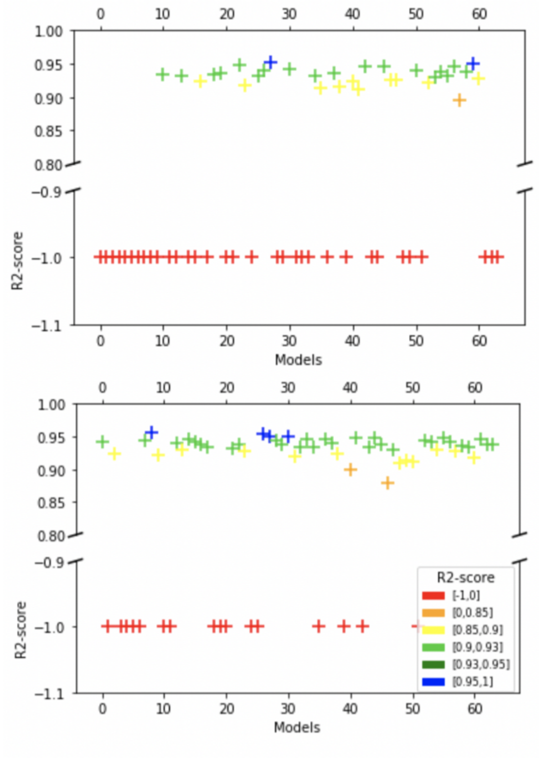
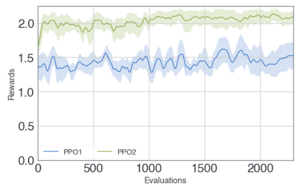


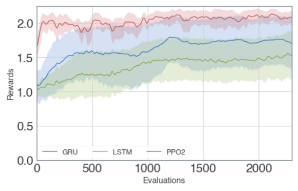
Figure 7: Initial (top) and improved (bottom) space search models and their R^2 scores

C. Deep Reinforcement Learning

1) Toy Problem: Before trying our Deep-RL implementation on the NAS problem we decided to evaluate it on a simpler problem where instead of training neural networks we replace the evaluation performed withing the environment by a function performing the sum of the actions of the current episode to compute the reward of this episode. Hence, the goal is to maximise the sum of actions over an episode.



(a) Proximal Policy Optimization with different observation spaces



(b) Vanilla Policy Gradient with RNN policies versus Proximal Policy optimization and RNN policy

Figure 8: Statistical performance on toy problem

The figure 8 shows a couple of figures where we want to compare the performance of different algorithm on our sum maximisation problem. Each of the trajectory represents the mean of 10 executions and their associated standard deviation.

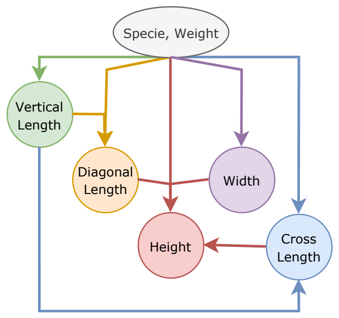
The figure 8a shows the PPO algorithm from *DeepHyper* using the original environment (PPO1) against the same algorithm using the extended environment with the new observation space considering the index of the current variable node, as explained in part III-A earlier (PPO2). In our toy problem the maximum reward achievable is 2.25 which is achieved very quickly by PPO2 whereas it was never achieved by PPO1. Based on this improvement we decided to use this new environment for neural architecture search.

The figure 8b compares the PPO2 to our VPG using either GRU or LSTM policies. As we can observe the GRU policy looks statistically better than the LSTM policy and both are performing better than PPO1. However PPO2 is much more efficient and secure because of its smaller variance. Therefore, PPO2 will be used for neural architecture search.

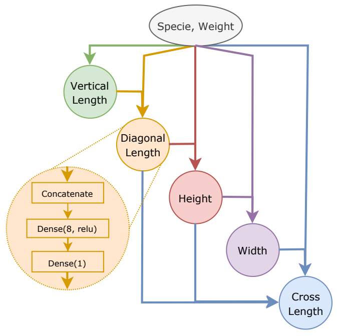
2) Neural Architecture Search: A search using PPO2 was executed on the reduced search space for the *Fish* dataset. At the 115th step the search find a model with a similar performance as the one found by AMBS (fig. 9a). The search was continued to see if it could reach better results. At around 1,000 evaluations a model with a performance of $R^2 = 0.968$ was found with 231 trainable parameters. This model is represented in the figure 9b. Experimentally we decided to add 2 more units per hidden layer now having a total of 265 trainable parameters. Then we train the model for 1,000 as we did for the baseline. The model reached a performance of 0.976 which is higher than the baseline. Also, this model had fewer trainable parameters than our baseline model (which had 335 trainable parameters).

V. Conclusion and Future Work

In this work we started by showing the role of conditional dependencies for some class of problems. Then, we demonstrated that it was feasible to use neural architecture search with a reinforcement learning scheme on a local laptop. Also, our experimentation is very encouraging given that we



(a) AMBS



(b) PPO2

Figure 9: Best models found

managed to beat the performance of a feed forward neural network by evaluating about 1,000 models in a search space of size 1,048,576. Moreover, the ability to build a neural network using conditional dependencies give the power to combine later multiple regressor inside such a model. It would be interesting to start the process only with neural networks and then try to replace some regressors by other methods automatically (i.e. such as random forest, gradient boosting...). Finally, the capacity to interpret such a model can help applicative scientists to understand more about their problem.

References

Chen, X., Xie, L., Wu, J. and Tian, Q. (2019). Progressive differentiable architecture search: Bridging the depth gap between search and evaluation, *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1294–1303.

Chung, J., Gulcehre, C., Cho, K. and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling, *arXiv preprint arXiv:1412.3555* .

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory, *Neural computation* **9**(8): 1735–1780.

Klein, A., Falkner, S., Springenberg, J. T. and Hutter, F. (2016). Learning curve prediction with bayesian neural networks.

Liu, H., Simonyan, K. and Yang, Y. (2018). Darts: Differentiable architecture search, *arXiv preprint arXiv:1806.09055* .

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017). Proximal policy optimization algorithms.

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning, *arXiv preprint arXiv:1611.01578* .

Zoph, B., Vasudevan, V., Shlens, J. and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710.

Appendix

Structure of the search space

The search space for our Neural Architecture Search is defined using both ’Constant Nodes’ and ’Variable Nodes’, which represent operations in the neural network’s structure. Constant Nodes store a unique operation, that cannot change during the search, while variable nodes store different possibilities of operations for the neural network’s architecture.

Here is an example of what a variable node looks like in the source code : it is the one stocking the XOR node operations for the XOR-AND-OR problem, which can operate on different outputs.

```
in_xor = VariableNode(name="in_XOR")
in_xor.add_op(Concatenate(ss, [x]))
in_xor.add_op(Concatenate(ss, [x, out_or]))
in_xor.add_op(Concatenate(ss, [x, out_and]))
in_xor.add_op(Concatenate(ss, [x, out_and,
    out_or]))
```