A CLOSER LOOK AT REINFORCEMENT LEARNING FOR NEURAL NETWORK ARCHITECTURE SEARCH

James A. Preiss*
University of Southern California
japreiss@usc.edu

Eugen Hotaj
Google Research NYC
ehotaj@google.com

Hanna Mazzawi
Google Research NYC
mazzawi@google.com

ABSTRACT

We explore the design decisions involved in reducing neural network architecture search (NAS) to a reinforcement learning (RL) problem. We compare several reductions on the NAS-Bench-101 dataset, while holding the RL algorithm and search space constant. Based on our findings, we discuss how NAS differs from typical RL settings, and suggest guidelines for applying RL to NAS problems.

1 INTRODUCTION

Novel neural network architectures have been a major driver of machine learning progress. Neural architecture search (NAS) aims to replace the human intution and experiments typically required for network design with automatic methods. Elsken et al. (2019) taxonomize NAS methods by three factors: The *search space* defines a combinatorial subset of all networks. The *search strategy* may be anything that supports a stochastic oracle query model, such as Bayesian optimization, heuristic search, continuous relaxations, or reinforcement learning (RL). *Performance estimation* reduces computational load: methods include early stopping, weight sharing, and predictive models.

RL as a search strategy has found state-of-the-art networks (Zoph et al., 2018; Pham et al., 2018), but the NAS and RL problems are not an exact match. RL targets sequential tasks where major challenges are unknown transition dynamics, temporal credit assignment, and exploration (Sutton & Barto, 2018). In contrast, NAS is not inherently sequential. To apply RL, one constructs an RL environment where an action sequence specifies an architecture. The dynamics are known, and the architecture's performance depends on all actions without obvious temporal structure. Still, RL suggests interesting possibilities because it produces a *design policy* instead of a single good network. This policy could, for example, be a function of the dataset or of in-progress training state.

To realize these ideas, we must first understand the simple case of designing a network unconditionally. However, few studies compare RL approaches for NAS in a controlled setting. If more than one of the three factors is changed, it hard to isolate the impact of each factor. In this work, we explore only one decision: the design of the RL environment used to reduce NAS to an RL problem. We hold the search space and RL algorithm constant, use no performance estimation, and evaluate architectures on a benchmark dataset (Ying et al., 2019). We compare several sequential reductions and a one-step reduction. Our top RL approaches are competitive with an evolutionary baseline (Real et al., 2018). Interestingly, our best-performing reduction is relatively less common in the literature. We interpret some of our findings into general remarks on the design of NAS $\rightarrow$ RL reductions.

2 REDUCTIONS FROM NAS TO RL

In this section, we formalize the NAS problem and the general RL problem, and introduce several ways to reduce NAS to RL. We use $\mathcal{P}(\mathcal{X})$ to denote the set of probability distributions over $\mathcal{X}$.

NAS Problem Statement. Let $\mathcal{M}$ denote a space of network architectures, henceforth *models*. Each model $M \in \mathcal{M}$ defines its own parameter space Θ_M, a reward function $R_M : \Theta_M \mapsto \mathbb{R}$ related to the learning task, and a randomized learning algorithm that induces a parameter distribution $\mathcal{O}_M \in \mathcal{P}(\Theta_M)$. R_M need not be the same loss optimized by $\mathcal{O}_M$ – in general R_M is the "true"

*Work completed during an internship at Google Research, New York, USA.

quantity of interest, e.g. $0-1$ classification accuracy, that may be intractable to optimize. R_M should be taken over a held-out validation set to account for overfitting. The NAS problem is therefore

$$\underset{M \in \mathcal{M}}{\text{maximize}} \; R_{\mathcal{O}}(M), \quad \text{where} \quad R_{\mathcal{O}}(M) = \mathbb{E}_{\theta \sim \mathcal{O}_M}[R_M(\theta)]. \tag{1}$$

RL Problem Statement. RL occurs in a Markov Decision Process (MDP) defined by state space $\mathcal{S}$, action space $\mathcal{A}$, initial state distribution $\rho \in \mathcal{P}(\mathcal{S})$, transition dynamics $T : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{P}(\mathcal{S})$, reward $r : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{P}(\mathbb{R})$, horizon $H \in \mathbb{N}$, and policy space $\Pi \subseteq \{\pi : \mathcal{S} \mapsto \mathcal{P}(\mathcal{A})\}$. Each $\pi \in \Pi$ induces a distribution over the trajectory space $\mathbb{T} = (\mathcal{S} \times \mathcal{A} \times \mathbb{R})^H$, meaning $s_1 \sim \rho$, $a_t \sim \pi(s_t)$, $s_{t+1} \sim T(s_t, a_t)$, $r_t \sim r(s_t, a_t)$. Using the shorthand $\tau \sim \pi$ for this distribution, the RL problem is

$$\underset{\pi \in \Pi}{\text{maximize}} \; \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{H} r_t\right]. \tag{2}$$

Desiderata of reductions. For a given NAS problem $(\mathcal{M}, \{R_M, \mathcal{O}_M\}_{M \in \mathcal{M}})$, we reduce it to RL by designing an MDP and a trajectory-model map $f : \mathbb{T} \mapsto \mathcal{M}$ that preserve the following properties:

Model coverage. For any $M \in \mathcal{M}$, there exists a policy $\pi \in \Pi$ such that $\Pr_{\tau \sim \pi}\{f(\tau) = M\} > 0$.

Reward correspondence. For any $\tau \in \mathbb{T}$, $\mathbb{E}\left[\sum_{t=1}^{H} r_t\right] = R_{\mathcal{O}}(f(\tau))$. In our reductions, there is no meaningful way to assign rewards to individual time steps, so we set $r_H = R_M(\theta)$ where $\theta \sim \mathcal{O}_M$.

Execution setting. During execution of a NAS $\rightarrow$ RL reduction for $N \in \mathbb{N}$ rounds, for each step $i \in \{1, \ldots, N\}$ we generate a trajectory $\tau_i \in \mathbb{T}$, a model $M_i = f(\tau_i)$, and reward estimate $\widetilde{R}_{\mathcal{O}}(M_i)$ computed by sampling from $\mathcal{O}_{M_i}$. The chosen model at step i is $M_i^* = \operatorname{argmax}_{1 \le j \le i} \widetilde{R}_{\mathcal{O}}(M_j)$. Although the NAS algorithm itself is optimizing for $R_{\mathcal{O}}$ using a validation set, its chosen models M_i^* will be judged on a held-out test set. We denote the test set reward by $\mathfrak{R}(M_i^*)$.

Objective mismatch. The RL objective does not exactly match the NAS objective. The RL objective is the *expected* reward of the policy, but the NAS objective is the *maximum* reward in a single episode.

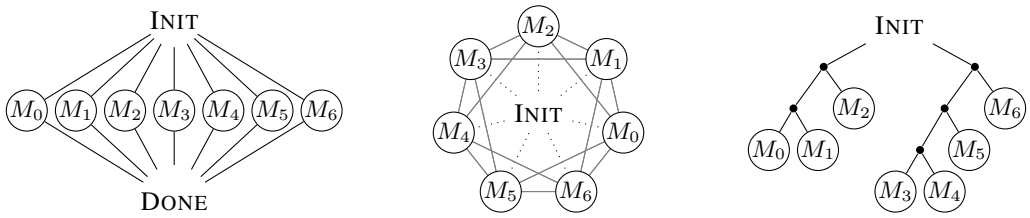
2.1 Selected reductions

We now introduce several NAS $\rightarrow$ RL reductions. Each is accompanied by a state transition graph in Figure 1 with markers for seven models $M_0, \ldots, M_6$ to illustrate the trajectory-to-model mapping f.

***Bandit* Reduction.** In the simplest possible reduction, one action specifies a model, so $\mathcal{A} = \mathcal{M}$. The name *Bandit* emphasizes that the policy is a pure distribution over actions, not a function of state. Although it is not sequential, *Bandit* is a natural reduction since the NAS problem has no inherent sequential structure. The parameterization of $\pi(a)$ can be complicated: for example, a recurrent neural network (RNN) policy can construct a network one layer at a time (Zoph & Le, 2017) or a convolutional cell one operation at a time (Zoph et al., 2018). Although sampling from the RNN is a sequential process with an internal state, it does not correspond to the MDP in an RL problem because gradients can flow through the intermediate states, whereas in RL the dynamics are black-box.

***Editor* Reduction.** In the *Editor* reduction, each state is a fully specified model, so $\mathcal{S} = \mathcal{M}$. Each action makes a small change to the current model state. For example, in the NAS-Bench-101 space where $\mathcal{M}$ is defined by a graph, the actions $\mathcal{A}$ consist of *add edge* (i, j), *delete edge* (i, j), and *set vertex i to operation k*. The state space forms a connected (not fully connected) graph. The actions are similar to mutations in evolutionary algorithms (Real et al., 2018; Suganuma et al., 2018) but appear in the RL context rarely (Zhou et al., 2018) compared to other reductions. We consider two axes of variation for *Editor*. One is the initial state distribution ρ: we compare a uniform distribution over $\mathcal{S}$ and a point distribution on a single $s^\star \in \mathcal{S}$. The other is the manner in which episodes end. If D denotes the maximal distance between two states in $\mathcal{S}$ using the edit actions in $\mathcal{A}$, we compare two options: 1) Set $H > D$ and add an action EVAL that stops the episode, or 2) Set $H < D$ with no EVAL action. The latter only preserves model coverage if ρ is uniform.

***Tree* Reduction.** In the *Tree* reduction, the policy specifies the model incrementally, for example by adding layers (Baker et al., 2017), applying function-preserving network morphisms (Cai et al., 2018), or building a convolutional cell (Zhong et al., 2018; Pham et al., 2018). The distinction from *Editor* is that some decisions are irreversible. States may correspond to partially or fully specified models, but the graph is no longer connected. To preserve model coverage, this implies that the initial state should be one from which all states are reachable. In the NAS-Bench-101 space, the initial state is a graph with no edges, and actions are either *add edge* (i, j), or *set vertex i to operation k*.



(a) *Bandit*. (b) *Editor*. (c) *Tree*.

Figure 1: State transition diagrams for our NAS $\rightarrow$ RL reductions. See Section 2.1 for descriptions.

3 EXPERIMENTS

The primary goal of our experiments is to compare the NAS $\rightarrow$ RL reductions defined in Section 2. We begin by describing the common setting shared by all experiments. Details are in Appendix A.

Benchmark dataset. The NAS-Bench-101 dataset (Ying et al., 2019) is the outcome of an exhaustive search over a space $\mathcal{M}$ of convolutional "cells". A cell is repeated in a fixed stack configuration to form a full network. Cells are parameterized by a connectivity graph and a choice among several convolution operations for each vertex. Each $M \in \mathcal{M}$ is trained on the CIFAR-10 image classification task (Krizhevsky, 2009) for three random seeds. Accuracy is logged at several checkpoints.

RL algorithm and reward. We use the REINFORCE policy gradient algorithm (Williams, 1992). Although more sophisticated RL algorithms exist, we do not believe that NAS shares the difficulties of more complex MDPs that make those algorithms necessary. We apply a logarithmic reward mapping $r_H(\tau) = -\log(1 + \epsilon - \widetilde{R}_{\mathcal{O}}(f(\tau)))$ for small ϵ. This magnifies differences between large accuracies and compresses differences between small accuracies. The accuracy estimate $\widetilde{R}_{\mathcal{O}}$ is the mean over the three training runs stored in the dataset. Hyperparameters were optimized with a random search.

Caching and stopping. RL is prone to query the same models repeatedly. Although some repetition is useful to overcome the randomness of $\mathcal{O}$, extreme repetition is not useful. We cache values of $\tilde{R}_{\mathcal{O}}$ to prevent repetitive training, freeing resources to explore more models. Since we compare algorithms by wall-clock time budget, the computation budget may never be reached if the algorithm gets stuck. Therefore, we terminate search if the algorithm queries 2000 cached models in a row.

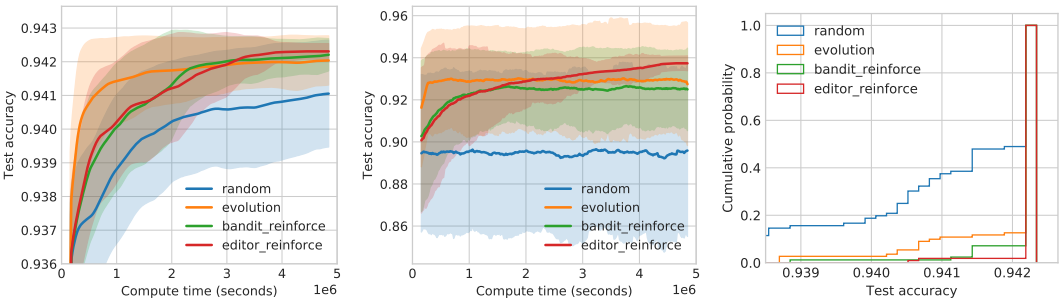
Format of results. For each experiment, we perform 1000 trials drawn from a uniform distribution over the relevant parameter. For each trial, we run the NAS algorithm until it reaches the computational time budget and compute the sequence of chosen models $\{M_i^*\}_{i=1}^N$ as described in Section 2. Note that N is different for each trial, but we leave this out of notation for simplicity. We plot the curve of total computational time used up to step i (x-axis) against the test accuracy $\mathfrak{R}(M_i^*)$ (y-axis). Results from all trials are aggregated to produce a mean $\pm$ standard deviation plot.

3.1 PER-REDUCTION EXPERIMENTS

Each reduction raises its own design decisions that must be decided experimentally to make a level playing field. For brevity, we summarize these with text only; plots and details are in Appendix B.

***Bandit* policy parameterization.** The simplest way to parameterize a distribution over $\mathcal{M}$ is a set of values $\ell \in \mathbb{R}^{d_{\mathcal{M}}}$ that define, via softmax functions, *independent* probability distributions for each categorical variable in $\mathcal{M}$. The dimensionality $d_{\mathcal{M}}$ depends on $\mathcal{M}$. To represent a distribution with *dependent* variables, we consider a latent variable model where a learned function $g : \mathbb{R}^{d_z} \mapsto \mathbb{R}^{d_{\mathcal{M}}}$ maps a latent code $z \sim \mathcal{N}(0, 1)^{d_z}$ to ℓ. Our g is a linear (not affine) map to ensure that z is not ignored. We compared the independent model against latent models with $d_z \in \{8, 16, 32\}$, but found equal performance. This suggests that expressive distribution classes are not critical for NAS.

***Editor* and *Tree* initial state distributions.** We compare the *Tree* reduction with initialization in the zero-state against the *Editor* reduction with both zero-state and random initialization. In all cases the policy $\pi : \mathcal{S} \mapsto \mathcal{P}(\mathcal{A})$ is an affine function followed by softmax. We experimented with neural network policies but found no benefit. The variants with deterministic initial state both find good models faster at first, but get stuck in repetitive distributions in the long term.



(a) Test accuracy of algorithm's chosen model M_t^* at each step.

(b) Test accuracy of the model M_t evaluated at each step.

(c) Test accuracy CDFs of algorithm's final chosen model M_N^*.

Figure 2: Test accuracy curves for NAS $\rightarrow$ RL reductions and baselines on NAS-Bench-101.

***Editor* episode stopping.** We compare the various ways to end the episode discussed in Section 2.1. More specifically, we compare option 1) with $H = 256$ and an EVAL action against option 2) with $H \in \{7, 12, 21\}$ and no EVAL action. We observe that option 1) performs significantly worse. For option 2) we observe slightly improved performance with larger H.

3.2 COMPARING SELECTED REDUCTIONS TO BASELINES

Based on the previous experiments, we select two RL methods to compare against baselines: *Bandit* with the independent parameterization, and *Editor* with $H = 12$ and ρ random. Our first baseline is random search (RS), in which M_i is sampled uniformly from $\mathcal{M}$. Despite its extreme simplicity, RS is a useful baseline because it never gets stuck. Our other baseline is regularized evolution (RE), an evolutionary algorithm that removes the oldest (instead of worst-performing) population member at each round (Real et al., 2018). We slightly modify RE to avoid getting stuck; see Appendix A.4.

In **Figure 2a**, we observe that all NAS algorithms significantly outperform random search in $\mathfrak{R}(M_i^*)$. RE begins finding good models more quickly than the RL methods, but both RL methods eventually surpass it slightly. In **Figure 2b**, we show $\mathfrak{R}(M_i)$ instead of $\mathfrak{R}(M_i^*)$. As expected, for RS the curve is constant. RE finds good models almost immediately, but its upper one-sigma bound is greater than the maximum accuracy in the dataset, indicating a skewed distribution that has some low outliers. These outliers are runs where RE got stuck. In comparison, *Bandit* samples models with slightly lower mean accuracy, but has a tighter distribution. *Editor* is notably superior to both, sampling a tight and high-accuracy distribution. In **Figure 2c**, we plot cumulative distributions of $\mathfrak{R}(M_N^*)$ across all trials for each algorithm – essentially, a more detailed visualization of the distributions at the final time step of Figure 2a. These confirm that RE has a longer tail of suboptimal models.

4 CONCLUSION

In this work we compared several NAS $\rightarrow$ RL reductions in a controlled benchmark setting. The best RL approaches we found were competitive with a strong evolutionary algorithm. While our results are limited to a single dataset and RL algorithm, some themes emerged. First, *simple parameterizations are sufficient*. For *Bandit*, latent variable models did not outperform independent distributions; for *Editor*, neural networks did not outperform linear policies. Second, *external randomness is helpful*. Among sequential reductions, the best-performing was *Editor* with random initialization and fixed episode length. The random initialization ensured that the policy constantly visits unseen states, and the short episode made it impossible for the policy to generate the same model every episode. *Editor* is less common in the literature than *Bandit* and *Tree*; it may be worthy of more attention.

Overall, we interpret our results as evidence that the desired behavior of a NAS $\rightarrow$ RL reduction is to bias a mostly-random search towards good models, and do so reliably. Complex parameterizations and deterministic environments make it easier to increase the per-episode expected reward, but they can be more prone to getting stuck, which ultimately hurts the true NAS objective: the maximum reward over all episodes. This idea can be carried into future work where the role of the RL design policy goes beyond single datasets into lifelong learning, multi-task, and other settings.

REFERENCES

Alekh Agarwal, Sham M. Kakade, Jason D. Lee, and Gaurav Mahajan. Optimality and approximation with policy gradient methods in markov decision processes. *CoRR*, abs/1908.00261, 2019.

Alexander A. Alemi, Ben Poole, Ian Fischer, Joshua V. Dillon, Rif A. Saurous, and Kevin Murphy. Fixing a broken ELBO. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 159–168. PMLR, 2018.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *ICLR*. OpenReview.net, 2017.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, pp. 2787–2794. AAAI Press, 2018.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20:55:1–55:21, 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1856–1865. PMLR, 2018.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

Alex Krizhevsky. Learning multiple layers of features from tiny images. *Master's Thesis, University of Toronto*, 05 2009.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4092–4101. PMLR, 2018.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 1889–1897. JMLR.org, 2015.

Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In *IJCAI*, pp. 5369–5373. ijcai.org, 2018.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (second edition)*. The MIT Press, second edition, 2018.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

Ronald J. Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3:241–, 09 1991. doi: 10.1080/09540099108946587.

Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *CoRR*, abs/1902.09635, 2019.

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *CVPR*, pp. 2423–2432. IEEE Computer Society, 2018.

Yanqi Zhou, Siavash Ebrahimi, Sercan Ömer Arik, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *CoRR*, abs/1806.07912, 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *ICLR*. OpenReview.net, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, pp. 8697–8710. IEEE Computer Society, 2018.

A EXPERIMENT SETUP DETAILS

In this appendix, we provide details about the common aspects shared by all experiments.

A.1 MOTIVATION AND DETAILS OF REINFORCE

To isolate the effect of NAS $\rightarrow$ RL reductions, we use the same RL algorithm for all reductions: REINFORCE with an exponential moving average reward baseline (Williams, 1992). In single-step, fixed-length, or variable-length episode reductions, our REINFORCE implementation takes exactly one gradient step per episode. We also applied L_2 regularization on the policy parameters and add a single-step entropy bonus to discourage collapse on bad local optima. To our knowledge, entropy regularization in REINFORCE was first proposed by Williams & Peng (1991) for applying REINFORCE to combinatorial optimization (!), in a setting similar to our *Bandit* reduction.

While REINFORCE is not considered a state-of-the-art algorithm, its flaws are less relevant to NAS than to other RL settings. Temporal difference methods like Q-learning (Watkins & Dayan, 1992) and Soft Actor-Critic (Haarnoja et al., 2018) primarily improve in the area of temporal credit assignment, which is not a major concern for NAS (see Section 1). We experimented with a Q-learning algorithm that exploits knowledge of the MDP dynamics to augment its replay memory with extra "imagined" experience, but we never found this method to outperform REINFORCE. Stabilized policy gradient methods like TRPO (Schulman et al., 2015) may offer some benefits (Duan et al., 2016; Zoph et al., 2018), but we found that REINFORCE did not suffer from stability issues, possibly due to the narrow range of reward values, regularization, and simple policy parameterizations we used.

Note that the REINFORCE algorithm commonly applied to sequential RL problems can also be applied to bandit problems without modification. Instead of optimizing the distribution of actions conditioned on states, we simply optimize a distribution of actions directly.

Our RL methods were implemented in TensorFlow. We did not use any third-party RL library. Our non-RL baselines were implemented in NumPy. Each experimental trial was fully sequential. Our results do not depend on any details of the execution environment such as a scheduler or the number of physical machines.

A.2 REWARD FUNCTION

In the NAS-Bench-101 dataset, models are measured by classification accuracy on the CIFAR-10 dataset. Accuracy has an undesirable failure mode when used directly as a reward and taken in expectation over a model distribution. Suppose we have two policies π_1 and π_2, each inducing the following distributions over accuracy values:

$$A_1 = \text{Uniform}(\{98\%, 99\%\}), \quad A_2 = \text{Uniform}(\{97\%, 99.5\%\}).$$

This leads to the undesirable property $\mathbb{E}[A_1] > \mathbb{E}[A_2]$, making π_1 preferable for an RL algorithm, whereas A_2 is preferable for NAS. The core problem is that, from the perspective of NAS, differences between high accuracies are more important than differences between low accuracies. We transform the reward to reflect this using concave mapping

$$R = -\log(1 + \epsilon - \text{accuracy}). \tag{3}$$

The factor of $\epsilon \geq 0$, when nonzero, ensures the argument to $\log$ remains positive. When $\epsilon = 0$, the mapping has the property that halving the error rate leads to a constant increase in the reward. This property is most important in datasets where the accuracy can get close to 100%. We also tried the mapping $R = \text{accuracy}^3$ suggested by Zoph & Le (2017) and found similar performance on NAS-Bench-101 as our logarithmic mapping. However, $R = \text{accuracy}$ degraded performance.

A.3 RL HYPERPARAMETERS

For all scalar hyperparameters, we selected values based on random search with 1000 trials. We used learning rates of 0.01 for probability distribution parameters and 0.003 for neural network weights. The reward moving average was updated with inertia of 0.95. The L_2 regularization weight was 10^{-2} for *Bandit* and 10^{-4} for *Editor*. The single-step entropy regularization weight was 10^{-2} for *Bandit* and 10^{-3} for *Editor*. In all cases, data visualization showed a wide range of approximately optimal values from which a "round" value could be manually chosen.

A.4 REGULARIZED EVOLUTION

We implement RE according to the listing of Algorithm 1 in Real et al. (2018). The mutation operators are the same as the action space of *Editor*. We use the population and tournament sizes of 100 and 10 respectively that were found to be near-optimal for the NAS-Bench-101 data set by the hyperparmeter search of Ying et al. (2019).

In our initial tests, we found that RE frequently converged to a repetitive set of models and triggered our repetitive-query stopping criterion long before consuming the full budget of compute time. To prevent this, we added another form of regularization: with probability 0.05, instead of mutating the tournament winner we replace it with a uniform sample from $\mathcal{M}$.

B EXTENDED EXPERIMENT RESULTS

In this appendix, we provide additional details and plots that did not fit within the page limit for the per-reduction experiments in Section 3.1. We also include one experiment in Appendix B.4 that was inconclusive, but may still be of interest.

B.1 *Bandit* POLICY PARAMETERIZATION

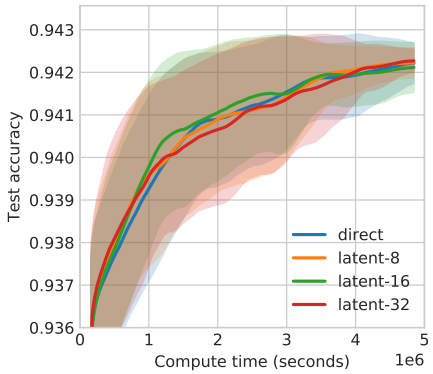


Figure 3: Comparing parameterizations of the policy distribution in *Bandit*. *Direct*: learn the parameters of an independent softmax distribution for each variable. *Latent-d_z*: learn a linear model mapping from a $\mathcal{N}(0, 1)^{d_z}$ latent variable to the softmax parameters. Although *latent-d_z* supports learning more complex distributions, it does not translate into better NAS performance.

Here we expand on our experiment comparing the independent and latent-variable models for parameterizing the *Bandit* policy, as described in Section 3.1. We chose the latent-variable model over the popular RNN (Zoph et al., 2018) because Ying et al. (2019) reported difficulty reproducing state-of-the-art RNN results on the NAS-Bench-101 dataset. The latent variable model resembles the decoder part of a variational autoencoder (Kingma & Welling, 2014). Under the latent model, the marginal distribution over $\mathcal{M}$ is computationally intractable, but since REINFORCE only requires sample access this is not an issue.

Our experiments parameterize the "decoder" g as a linear map without any affine bias term. This ensures that the output logits depend on the latent code. If we added a bias term, it would be possible to learn $\ell = 0 \cdot z + b$ for a nonzero bias b, and end up with independent distributions over each model variable despite our efforts to avoid that. This is a well-known problem with variational autoencoders (Alemi et al., 2018). In preliminary experiments where we introduced the bias term b, we found performance to be unchanged.

Results are shown in Figure 3. As mentioned in Section 3.1, the nearly-equal rewards for the independent model and for all latent variable models suggest that the ability to learn complex distributions is not a requirement for NAS $\rightarrow$ RL reductions. Given these results, we did not pursue further expressive models such as the RNN.

B.2 *Editor* AND *Tree* INITIAL STATE DISTRIBUTIONS

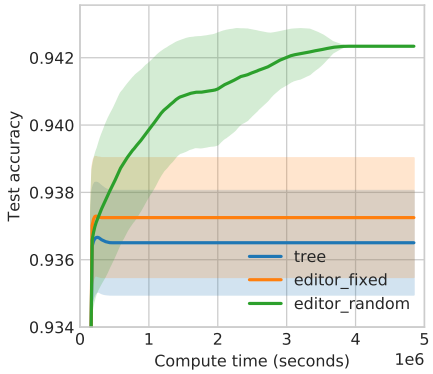


Figure 4: Accuracy curves for *Tree* with deterministic initialization, *Editor* with deterministic initialization, and *Editor* with random initialization. All but the latter get stuck on repetitive distributions.

In this experiment, we compare variations of the sequential *Editor* and *Tree* reductions discussed in Section 2. Results are shown in Figure 4. The environments with deterministic initial states begin finding good models quickly, but soon get stuck in repetitive distributions over actions that lead to repetitive distributions over models, triggering our stopping criterion.

We conjecture that the superior performance of random *Editor* is due to improved robustness against bad local optima. Agarwal et al. (2019) showed that suboptimality in policy gradient RL can be upper-bounded by a function involving the likelihood ratio of the optimal policy's state distribution and the initial state distribution. The uniform initial state distribution ensures that this ratio cannot become too large. Other possible ways to inject good exploration would be an off-policy RL algorithm with epsilon-greedy exploration, or functional constraints on the softmax inputs (such as $\tanh$ squashing) that lower-bound the entropy of the action distributions. However, if these strategies are not used, it appears essential to inject randomness via the environment itself.

B.3 *Editor* EPISODE STOPPING

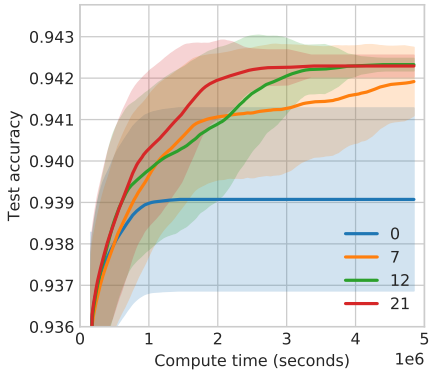
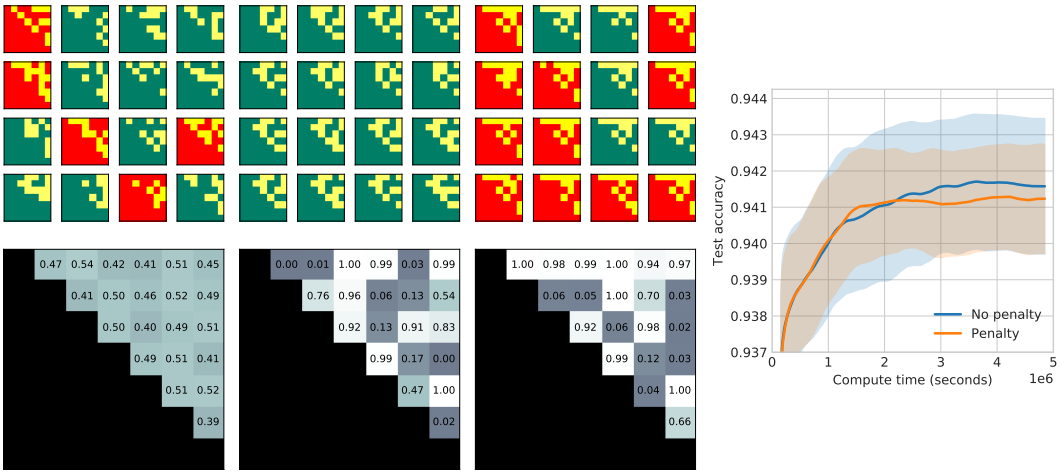


Figure 5: Accuracy curves for *Editor* with different episode ending schemes. Zero indicates (almost) unlimited episode length H with an EVAL action to end episodes. Positive values indicate fixed H.

Next, we compare the different ways to end the episode for *Editor* discussed in Section 2.1. Results are shown in Figure 5. The blue curve labeled with 0 corresponds to option 1): an EVAL action and a limit on H to ensure finite episodes. However, the limit H is 256, much larger than the maximal edit distance $D = 26$ for the NAS-Bench-101 dataset. The remaining curves correspond to option 2): ending the episode after a fixed number of steps, with no policy action to end early. The latter perform dramatically better.



(a) (b) (c) (d)

Figure 6: NAS-Bench-101 adjacency matrix samples and per-edge probabilities for: (a) Random policy with uniform distribution on each variable; (b) Trained policy with reward penalty for invalid models; (c) Trained policy with no penalty for invalid models. Invalid graphs are colored red; note that (c) samples many invalid models. Test accuracy curves for policies (b) and (c) are shown in (d).

B.4 Penalizing invalid models

When we parameterize the NAS-Bench-101 search space as a product of categorical variables, there exist assignments of these variables that specify an invalid model. For example, to keep benchmark data collection tractable, the number of edges in the connectivity DAG is limited to reduce the size of $\mathcal{M}$. This raises the question: when the policy samples an invalid model, what should happen? We compare two options: 1) Penalize the policy by giving a reward of zero, or 2) Do nothing – leave the policy parameters unchanged and move on to the next episode.

We compare the two reward schemes in Figure 6. In Figures 6a to 6c, the top image represents a sample of sixteen adjacency matrices for the NAS-Bench-101 search space drawn from a policy. Valid graphs are colored green, while invalid graphs are colored red. The bottom image represents the probabilities of individual edges estimated over a larger sample. In Figure 6a, a uniform distribution over the adjacency matrix includes some invalid models (colored red) but more valid models (colored green). In Figure 6b, the policy has been trained with a penalty. This leads to a distribution of mostly valid models. In Figure 6c, the policy has been trained with no penalty. This leads to a distribution that includes invalid models more often than the uniform (untrained) case.

These qualitative results suggest that the penalty for invalid models might be a "distraction" in the sense that a distribution over $\mathcal{M}$ containing half invalid models and half high-reward valid models may lead to a lower expected reward than a distribution containing only low-reward valid models. We test this hypothesis by examining the test accuracy curves for both schemes in Figure 6d. The curves show a possible slight advantage for the "do nothing" reward scheme, but due to the large standard deviations relative to the effect size, we consider this experiment inconclusive. However, based on this experiment, we decided to use the "do nothing" reward scheme for all other experiments discussed in this paper.