NetHack is Hard to Hack

Anonymous Author(s)
Affiliation
Address
email

Abstract

Neural policy learning methods have achieved remarkable results in various control problems, ranging from Atari games to simulated locomotion. However, these methods struggle in long-horizon tasks, especially in open-ended environments with multi-modal observations, such as the popular dungeon-crawler game, NetHack. Intriguingly, the NeurIPS 2021 NetHack Challenge revealed that symbolic agents outperformed neural approaches by over four times in median game score. In this paper, we delve into the reasons behind this performance gap and present an extensive study on neural policy learning for NetHack. To conduct this study, we analyze the winning symbolic agent, extending its codebase to track internal strategy selection in order to generate one of the largest available demonstration datasets. Utilizing this dataset, we examine (i) the advantages of an action hierarchy; (ii) enhancements in neural architecture; and (iii) the integration of reinforcement learning with imitation learning. Our investigations produce a state-of-the-art neural agent that surpasses previous fully neural policies by 127% in offline settings and 25% in online settings on median game score. However, we also demonstrate that mere scaling is insufficient to bridge the performance gap with the best symbolic models or even the top human players.

1 Introduction

Reinforcement Learning (RL) combined with deep neural policies has achieved impressive results in control problems, such as short-horizon simulated locomotion tasks [54, 7]. However, these methods struggle in long-horizon problem domains, such as NetHack [33], a highly challenging grid-world game. NetHack poses difficulties due to its vast state and action space, multi-modal observation space (including vision and language), procedurally-generated randomness, diverse strategies, and deferred rewards. These challenges are evident in the recent NetHack Challenge [23], where agents based on hand-crafted symbolic rules outperform purely neural approaches (see Figure 1), despite the latter having access to high-quality human demonstration data [24] and utilizing large-scale models.

We propose three reasons for the poor performance of large-scale neural policies compared to symbolic strategies. First, symbolic strategies implement hierarchical control schemes, which are generally absent in neural policies used for NetHack. Second, symbolic models use hand-crafted parsers for multi-modal observations, suggesting that larger networks could enhance representations extracted from complex observations. Third, symbolic strategies incorporate error correction mechanisms, which could be crucial for improving neural policies if integrated with RL-based error correction.

In this work, we conduct a comprehensive study of NetHack and examine various learning mechanisms to enhance the performance of neural models. We bypass traditional RL obstacles, such as sparse rewards or exploration challenges, by focusing on imitation learning. However, we find that existing datasets lack crucial information, such as hierarchical labels and symbolic planning traces. To address this, we augment the codebase of `AutoAscend`, the top-performing symbolic agent in

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

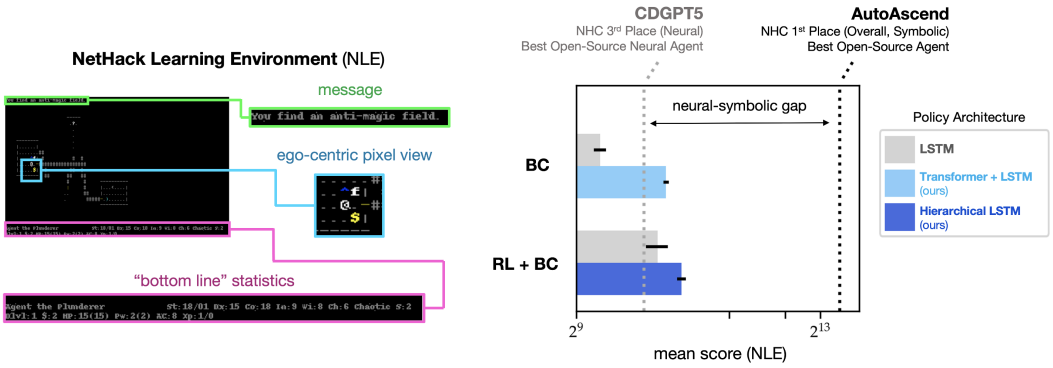

Figure 1: *Left:* The per-step observation for NetHack agents consists of an ego-centric pixel image (blue) and two text fields containing in-game messages (green) and statistics (pink). *Right:* Selected results from the NeurIPS 2021 NetHack Challenge (NHC) [23] showing game score on a log-scale. Neural baseline models (grey) trained with behavioral cloning (BC) on human data perform poorly, but are somewhat improved when fine-tuned with RL. We find that the introduction of hierarchy and changes in model architecture yield significant improvements (light/dark blue), resulting in state-of-the-art performance for a neural model. However all neural approaches are significantly worse than `AutoAscend`, a hand-crafted symbolic policy. Our paper explores this performance gap.

the 2021 NetHack Challenge, and extract hierarchical labels tracking the agent's internal strategy selection in order to construct a large-scale dataset containing 10^9 actions.

Using this dataset, we train a range of deep neural policies and investigate: (a) the advantages of hierarchy; (b) model architecture and capacity; and (c) fine-tuning with reinforcement learning. Our main findings are as follows:

- Hierarchical behavioral cloning (HBC) significantly outperforms BC and baseline methods, provided that the model has adequate capacity.
- Large Transformer models exhibit considerable improvements over baselines and other architectures, such as LSTMs. However, the power-law's shallow slope indicates that scaling alone will not suffice to solve the game.
- Online fine-tuning with RL further enhances performance, with hierarchy proving beneficial for exploration.
- The combined effects of hierarchy, scale, and RL lead to state-of-the-art performance, narrowing the gap with `AutoAscend` but not eliminating it.

Additionally, we open-source our code, models, and the `HiHack` repository, which includes (i) our 10^9 dataset of hierarchical labels obtained from `AutoAscend` and (ii) the augmented `AutoAscend` and `NLE` code employed for hierarchical data generation, encouraging further development.

2 Related Work

Our work builds upon previous studies in the NetHack environment, imitation learning, hierarchical learning, and the use of transformers as policies. In this section, we briefly discuss the most relevant works.

NetHack Following the introduction of the NetHack Learning Environment (NLE) [33], the NetHack Challenge (NHC) competition [23] enabled comparisons between a range of different agents. The best performing open-source symbolic and neural agents were `AutoAscend` and `Chaotic-Dwarven-GPT-5` (`CDGPT5`), respectively, and we base our investigations on them, as well as on the NetHack Learning Dataset [24] dataset.

Several notable works make use of the NetHack environment: Zhong et al. [60] show how a dynamics model can be learned from the Nethack text messages and leveraged to improve performance. On account of their utility, we also encode the in-game message but instead use a model-free policy to

pick actions. Bruce et al. [9] show how a monotonic progress function in NetHack can be learned from human play data and then combined with RL reward to solve long-range tasks in the game. This represents a complementary way of employing NetHack demonstration data without direct action imitation.

Imitation Learning Pomerleau [46] demonstrated the potential of driving an autonomous vehicle using offline data and a neural network, which has since become an ongoing research topic for scalable behavior learning [3, 6, 49]. These approaches can be categorized into two main classes: Offline RL [21, 31, 32, 57, 35, 20], which focuses on learning from mixed-quality datasets with reward labels; and Imitation Learning [41, 43, 44, 26], which emphasizes learning behavior from expert datasets without reward labels. Our work primarily belongs to the latter category as it employs a behavior cloning model. Behavior cloning, a form of imitation learning, aims to model the expert's actions given the observation and is frequently used in real-world applications [59, 61, 59, 47, 18, 58]. Since behavior cloning algorithms typically address a fully supervised learning problem, they are often faster and simpler than reinforcement learning or offline RL algorithms while still yielding competitive results [20, 22]. A novel aspect of our work is the use of hierarchy in conjunction with behavioral cloning, i.e. supervision at multiple levels of abstraction, a topic which has received relatively little attention. Recent efforts to combine large language models with embodied agents use the former to issue a high-level text "action" to the low-level motor policy. Approaches such as Abramson et al. [1] have shown the effectiveness of hierarchical BC for complex tasks in a simulated playroom settings.

Hierarchical Policy Learning Hierarchical Reinforcement Learning (HRL) based techniques [5, 4] have sought to address complex and long-horizon tasks via temporal abstraction across hierarchies, as demonstrated by Levy et al. [36] and Nachum et al. [39][40]. Similarly, numerous studies have concentrated on showing that primitives [17, 53, 52] can be beneficial for control. These concepts have been combined in works such as Stochastic Neural Networks by Florensa et al. [19], where skills are acquired during pretraining to tackle diverse complex tasks. Likewise, Andreas et al. [2] learn modular sub-policies for solving temporally extended tasks. However, most prior work focus on learning both levels of the hierarchy which makes training complex and correspondingly the resulting approaches have had limited success on more challenging tasks and environments. Le et al. [34] explores the interaction between hierarchical learning and imitation and find benefits albeit in goal-conditioned settings. In contrast, our work uses a fixed hierarchy, chosen by the domain-expert designer of `AutoAscend`, which simplifies our study of overall learning mechanisms for NetHack.

Transformers for RL The remarkable success of transformer models [56] in natural language processing [15, 8] and computer vision [16] has spurred significant interest in employing them for learning behavior and control. In this context, [11, 28] apply transformers to Reinforcement Learning and Offline Reinforcement Learning, respectively, while [12, 14, 37] utilize them for imitation learning. Both [14, 37] primarily use transformers to summarize historical visual context, whereas [12] focuses on their long-term extrapolation capabilities. More recent work have explored the use of multi-modal transformers [27] to fit large amounts of demonstration data [48, 51, 55]. To enable transformers to take in larger token sizes, recurrent transformer models have been proposed [13, 10]. Our work draws inspiration from these use cases, employing a transformer to consolidate historical context and harness its generative abilities.

3 Data Generation: Creating the `HiHack` Dataset

3.1 Extending the NetHack Learning Environment

The NetHack Learning Environment (NLE) is a *gym* environment wrapping the NetHack game. Like the game itself, the action and state spaces of NLE are complex, consisting of 121 distinct actions and ten distinct observation components. The full observation space of NLE is far richer and more informed than the view afforded to human players of NetHack, who observe only the more ambiguous "text-based" components of NLE observations, denoted as *tty_chars*, *tty_colors*, and *tty_cursor*. This text-based view corresponds also to the default format in which both NetHack and NLE gameplay is recorded, loaded, and streamed via the C-based `ttyrec` library native to the NetHack game.

The popular NetHack Learning Dataset (NLD) offers two large-scale corpuses of NetHack game-play data, `NLD-AA`, consisting of action-labeled demonstrations from `AutoAscend`, and `NLD-NAO`, consisting of unlabeled human player data [24]. NLD adopts the convention of recording only *tty∗* components of NLE observations as a basis for learning, hence benefiting from the significant speedups in data operations offered via integration with the `ttyrec` library. We adhere to this convention with the hierarchical `HiHack` dataset introduced in this paper. Thus, in order to generate our dataset, we extend the `ttyrec` library to store hierarchical *strategy* or *goal* labels alongside action labels. We further integrate this extension of `ttyrec` with NLE, modifying the gym environment to accept an additional hierarchical label at each step of interaction. This input hierarchical label does not affect the underlying state of the environment, and is instead employed strictly to enable the recording of hierarchically-informed NetHack game-play to the `ttyrec` data format.

3.2 AutoAscend: A Hierarchical Symbolic Agent

An inspection of the fully open-source code base underlying the `AutoAscend`, publicly released in the proceedings of the NeurIPS 2021 NetHack Challenge competitions, reveals the internal structure of the bot to be composed of a directed acyclic graph of explicitly defined *strategies*, which are switched between by the bot's underlying *global controller* in an imperative manner via sets of strategy-specific predicates, as visualised in Figure 2. Among these strategies are hand-engineered routines for accomplishing a broad range of goals crucial to effective survival in the game and successful descent through the NetHack dungeons. These include routines for fighting off arbitrary monsters, selecting food that is safe to eat from an agent's inventory, and efficiently exploring the dungeon while gathering and identifying valuable items, among many others. The various strategies are supported in turn by shared *sub-strategies* for accomplishing simpler "sub-goals" (see supplemental material for the full graph).

We exploit this explicit hierarchical structure in the generation of `HiHack`, extending the `AutoAscend` codebase to enable per-step logging of the strategy responsible for yielding each action executed by the bot throughout environment interaction, as supported by our modifications to the C-based `ttyrec` writer library and NLE.

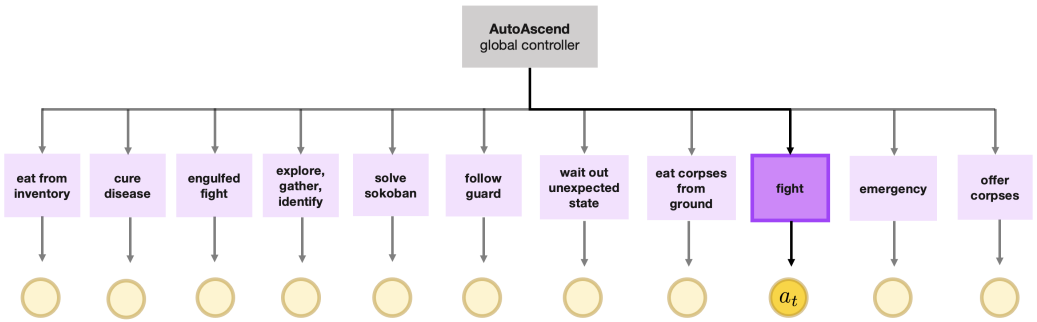

Figure 2: A diagrammatic visualization of the internal structure of `AutoAscend`. The bot is composed of eleven goal-directed, high-level *strategies*. The "global controller" underlying `AutoAscend` employs a complex predicate-based control flow scheme to determine which strategy to query for an action on a per-timestep basis [23].

3.3 The HiHack Dataset

Our goal in generating the HiHack Dataset (`HiHack`) is to create a hierarchically-informed analogue of the large-scale `AutoAscend` demonstration corpus of NLD, `NLD-AA`. Thus, as previously alluded to, `HiHack` is composed of demonstrations recorded in an extended version of the `ttyrec` format, consisting of sequences of *tty∗* observations of the game state accompanied by `AutoAscend` action and strategy labels. `HiHack` contains a total of 3 billion recorded game transitions, reflecting more than a hundred thousand `AutoAscend` games. Each game corresponds to a unique, procedurally-generated "seed" of the NetHack environment, with `AutoAscend` playing as one of thirteen possible character "starting roles" across a unique layout of dungeons.

We verify that the high-level game statistics of `HiHack` match those of `NLD-AA` in Table 1. Indeed, we find a high degree of correspondence across mean and median episode score, total number of transitions, and total number of game turns. We attribute the very slightly diminished mean scores, game transitions, and turns associated with `HiHack` to a difference in the underlying versions of `NLE` employed in the generation of `HiHack` and `NLD-AA`, with the former generated via the `NLE v0.9.0`.

Table 1: A comparison of dataset statistics between `NLD-AA` [23] and our generated `HiHack`, produced by running `AutoAscend` in `NLE v0.9.0`.

	`NLD-AA`	`HiHack`
Total Episodes	109,545	109,907
Total Transitions	3,481,605,009	3,244,729,367
Mean Episode Score	10,105	8,166
Median Episode Score	5,422	5,147
Median Episode Game Transitions	28,181	27,496
Median Episode Game Turns	20,414	19,991
Hierarchical Labels	✗	✓

4 Hierarchical Behavioral Cloning

Our first set of experiments leverage the hierarchical strategy labels recorded in `HiHack` for offline learning with neural policies, via Hierarchical Behavior Cloning (HBC).

Method Mimicking the imperative hierarchical structure of `AutoAscend`, we introduce a bilevel hierarchical decoding module over a popular NetHack neural policy architecture, namely the `ChaoticDwarvenGPT5` (`CDGPT5`) model. This model achieved 3rd place in the neural competition of the NeurIPS 2021 NetHack Challenge when trained from scratch with RL, making it the top-performing open-source neural model for NetHack [23].

The `CDGPT5` model consists of three separate encoders: a 2-D convolutional encoder for pixel-rendered visual observations of the dungeon o_t, a multilayer perceptron (MLP) encoder for the environment message m_t, and a 1-D convolutional encoder for the bottom-line agent statistics b_t. These three observation portions are extracted from the *tty∗* NLE observations of `HiHack`. The core module of the network is an LSTM, which is employed to produce a recurrent encoding of an agent's full in-game trajectory across what may be hundreds of thousands of keypresses, both in training and at test-time. The core module also receives a one-hot encoding of the action a_{t-1} executed at the previous time-step as input.

Our hierarchically-extended version of this LSTM-based policy is shown in Figure 3(left). We replace the linear decoder used to decode the LSTM hidden state into a corresponding action label in the `CDGPT5` model with a hierarchical decoder consisting of (i) a single "high level" MLP, responsible for predicting the strategy label g_t, given the environment observation tuple $\{m_t, o_t, b_t\}$, and (ii) a set of "low level " MLPs, one for each of the discrete strategies in the `AutoAscend` hierarchy (see Figure 2), with a SoftMax output over discrete actions. The strategy prediction g_t selects which of these low-level MLPs to use.

We employ a simple cross-entropy loss to train both the baseline non-hierarchical LSTM `CDGPT5` policy, as well as our Hierarchical LSTM policy, aggregating gradients across the bilevel decoders of the latter via the Gumbel-Softmax reparameterization trick.

Training and evaluation details We train all policies on a single GPU for 48 hours with the full $3.2B$ `HiHack` dataset. As with all offline experiments in the paper, a total of 6 seeds are used to randomize dataloading and the neural policy parameter initialization. We employ mean and median NLE score on withheld instances of NLE ($n = 1024$) as our central metrics for evaluating and comparing model performance at the conclusion of training, following the convention introduced in the NetHack Challenge competition [23]. Reported performance is aggregated over random seeds. Further details of architectures as well as training and evaluation procedures can be found in the supplemental material.

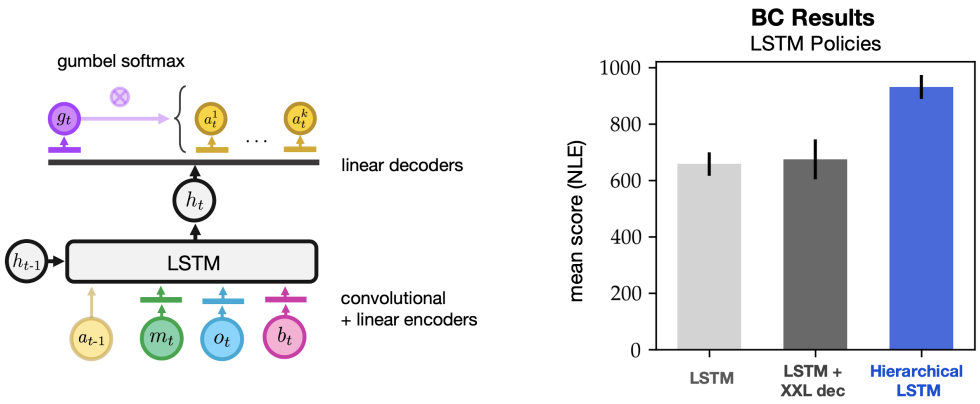

Figure 3: *Left:* Hierarchical LSTM-based policy model for behavioral cloning, where g_t is the high-level strategy prediction (purple) that is used to select over the k low-level policies (yellow). Figure 1 shows the input observations. *Right:* Mean score for baseline LSTM model [23] (grey), our hierarchical model (blue) at the conclusion of training. The addition of hierarchy labels provides a significant performance gain, not matched by a model capacity-matched version of the baseline (dark grey). All mean NLE scores are computed over large-scale evaluations run over 6 model seeds.

Results We find that the introduction of hierarchy results in a significant improvement to the test-time performance of LSTM policies trained with behavioral cloning, yielding a 40% gain over the baseline in mean NLE score as shown in Figure 3(right), and 50% improvement in median score across seeds as shown in Table 2. Additionally, to verify that this improvement in performance is indeed due to hierarchy and not simply a result of the increased parameter count of the hierarchical LSTM policy, we run ablation experiments with a modified, large-decoder version of the baseline (non-hierarchical) policy architecture. The results, shown in Figure 3(right), show that increasing the size of the LSTM decoder, without the introduction of a hierarchy, does not result in any performance improvements over the baseline.

5 Architecture and Data Scaling

Despite the benefits of introducing hierarchical labels, the performance remains significantly behind the symbolic policy used to generate the `HiHack` demonstrations in the first place, `AutoAscend`.

This finding prompts us to explore scaling – perhaps increasing the data and/or model capacity data may close this performance gap.

Method To test this new hypothesis, we conduct a two-pronged investigation: (i) to explore model capacity, we develop a novel base policy architecture for NetHack that introduces a Transfomer module into the previous `CDGPT5`-based architecture; and (ii) for data scaling, we run a second set of "scaling-law" [29] experiments that use subsets of the `HiHack` dataset to quantify the relationship between dataset size and the test-time performance of BC policies.

Our novel base policy architecture is visualized in Figure 4 (left). This features two copies of the observation encoders employed in `CDGPT5`. One set is kept frozen and employed strictly to yield a recurrent encoding of the complete NetHack trajectory up to the current observation step t via a pre-trained frozen LSTM, while the second is kept "unlocked" and is employed to provide embeddings of NetHack observations directly to a causal Transformer, which receives a shorter, fixed context length during training.

Training and evaluation details The training and evaluation procedures employed in here echo those of section 4. In our data-scaling experiments, the subset of sampled `HiHack` games employed in offline training is also randomized over model seeds. The causal Transformer component of our Transformer-LSTM models is trained with the same fixed, context-length window, $c = 64$. The size

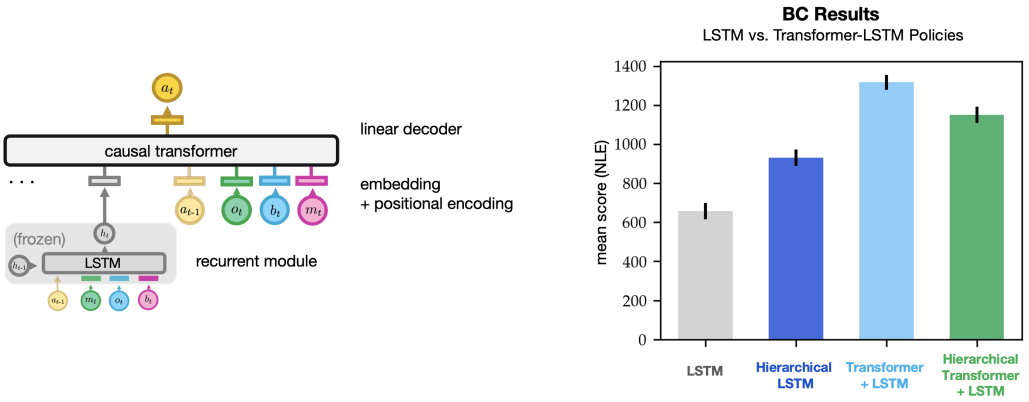

Figure 4: *Left:* Transformer-based architecture (non-hierarchical version). The LSTM encoder (grey) is used to provide a long temporal context h_t to the Transformer. *Right:* The Transformer model outperforms LSTM-based models with & without hierarchy (see Section 4 and [23] respectively).

of this content length window was selected via a set of hyperparameter tuning experiments, delineated in Appendix D.

Results The architecture experiments in Figure 4(right) show that both the non-hierarchical and hierarchical variants of our combined transformer-LSTM policy architecture yield gains eclipsing those granted solely by the introduction of hierarchy in the offline learning setting. Probing further, in Figure 5(left), we compare the performance of two variants of our Transformer-LSTM model, one with 3 layers (50.8M parameters) and another with 6 layers (98.9M parameters). The larger model can be seen to perform worse than the smaller one due to over-fitting. This suggests that scaling of model capacity alone will not be sufficient to close the neural-symbolic gap.

In Figure 5(right), we now explore the effect of training set size on mean NLE test score. We perform BC training for the LSTM baseline [23] and our largest 6 layer Transformer-LSTM model (98.9M params) for 10^1 up to 10^5 games, subsampled from the `HiHack` dataset. For both architectures, we observe a **sub log-linear** dependence on training set size, asymptoting at a mean score of approximately 1000. Thus, brute force scaling of the dataset alone cannot viably close the gap to symbolic methods (score of 8500).

Though our architecture and data scaling experiments are compute-time constrained, we find the test-time performance of all tested models to saturate on the given computational budget. Full training curves are included in the supplementary materials.

6 Combining Imitation with Reinforcement Learning

Given that hierarchy and scaling are insufficient to bridge the performance gap with `AutoAscend`, we now explore the impact of an online learning using reinforcement learning.

Method In this set of experiments, we build on results from the "Dungeons and Data: A Large-Scale NetHack Dataset" paper [24], which showed that behavioral cloning coupled with reinforcement learning is superior to RL training from scratch. As in Hambro et al. [24], we employ the asynchronous `moolib` distributed-RL library to train our models with a combination of **BC and asynchronous proximal policy optimization (APPO)** [38, 45, 50]. At each time-step of training, the overall loss is a weighted combination of BC and RL losses, i.e. the cross-entropy loss of a batch of demonstrations from `HiHack` plus an RL loss over a batch of rollouts of the current policy in NLE. For hierarchical models, we only use RL to update the low-level strategies; that is the strategy selection policy is trained with BC alone and not updated with RL.

Training and evaluation details Our high-level training procedure here mirrors that of our hierarchical behavioral cloning experiments: we evaluate model performance under the constraint of

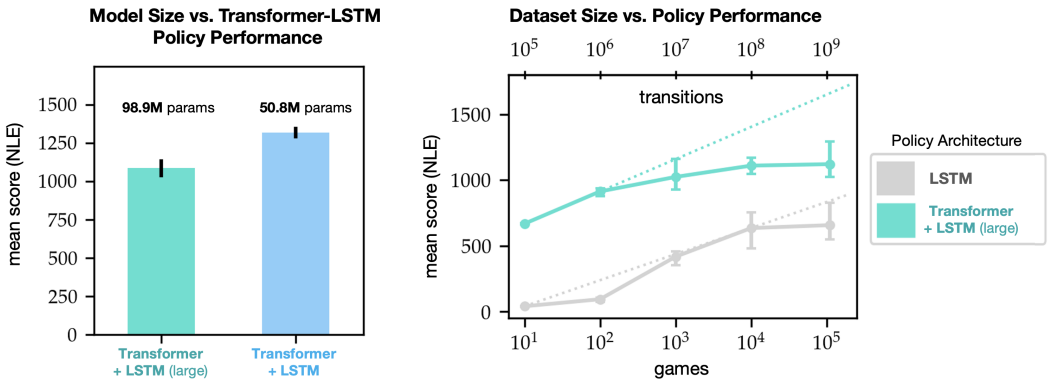

Figure 5: *Left:* Model capacity versus mean score for our Transformer-LSTM model. The larger model performs worse. *Right:* Dataset scaling experiments showing diminishing returns as the number of training games reaches 10^5. Collectively these two plots show that scaling of data and model size is not sufficient, in of themselves, to close the performance gap to symbolic models.

Table 2: [V4] Evaluating the impact of **hierarchical labels** and **architectural improvement** on the performance of policies trained both with behavioral cloning, as well as with combined behavioral cloning and asynchronchronous proximal policy optimization. All policies were trained for 48 hours on a single GPU. Metrics annotated with (†) were computed only for the top-performing neural policy seed (out of 6) across each model class.

			Score		Dlvl (†)	Turns (†)	
		Hierarchy	Mean	Median	Mean	Mean	Median
BC	LSTM [23]	✗	658 ± 41	403	1.11 ± 0.01	5351 ± 76	4111
BC	LSTM	✓	931 ± 42	614	1.09 ± 0.01	6983 ± 84	5981
BC	Transformer-LSTM	✗	**1318 ± 38**	**914**	**1.36 ± 0.01**	6088 ± 75	5121
BC	Transformer-LSTM	✓	1151 ± 43	731	1.26 ± 0.01	**7568 ± 99**	**6242**
APPO + BC	LSTM [23]	✗	1204 ± 138	779	1.07 ± 0.01	8712 ± 112	7376
APPO + BC	LSTM	✓	**1551 ± 73**	**972**	1.09 ± 0.01	**11435 ± 134**	**9849**
APPO + BC	Transformer-LSTM	✗	1326 ± 28	887	1.25 ± 0.01	7924 ± 99	6788
APPO + BC	Transformer-LSTM	✓	1346 ± 16	894	**1.32 ± 0.01**	7874 ± 101	6769
Symbolic	`AutoAscend`	✓	8556 ± 187	4918	3.10 ± 0.04	19586 ± 171	19710

computation time, training all policies for exactly 48 hours on a single A100 GPU, using 6 different seeds to randomize data loading and environment seeding only. All policies belonging to the same model class are initialized from a single checkpoint pre-trained with BC alone via the procedure delineated in section 4. The pre-trained checkpoints used for initialization are selected on the basis of test-time performance, with the median checkpoint employed in each set of experiments.

Results Table 2 summarizes the performance of all our models and a number of observations can be made: (a) RL fine-tuning offers clear and significant performance boost to all models, with gains in the test-time mean NLE score associated with all model classes; (b) the best performing approach is APPO + BC using the hierarchical LSTM model. The mean score of 1551 represents a new state-of-the-art for neural policies on NLE, beating the previous best result by 48% in mean NLE score and 25% in median NLE score; (c) the Transformer-LSTM models, being slower to train than LSTM models, perform worse due to the fixed training time budget imposed; (d) other metrics, such as dungeon level reached and the lifetime of the agent (in game turns) show a broadly similar pattern to the mean score metric (used in NHC [23]) and (e) for BC, hierarchy seems to hurt performance for the larger Transformer-RL models but this gap is closed once APPO fine-tuning is used.

In NetHack, the player can choose from 13 distinct roles (barbarian, monk, wizard, etc.), each of which require distinctive play-styles. In NLE, starting roles are randomized, by default. Figure 6 shows a score distribution breakdown across role for different neural policy classes, trained with

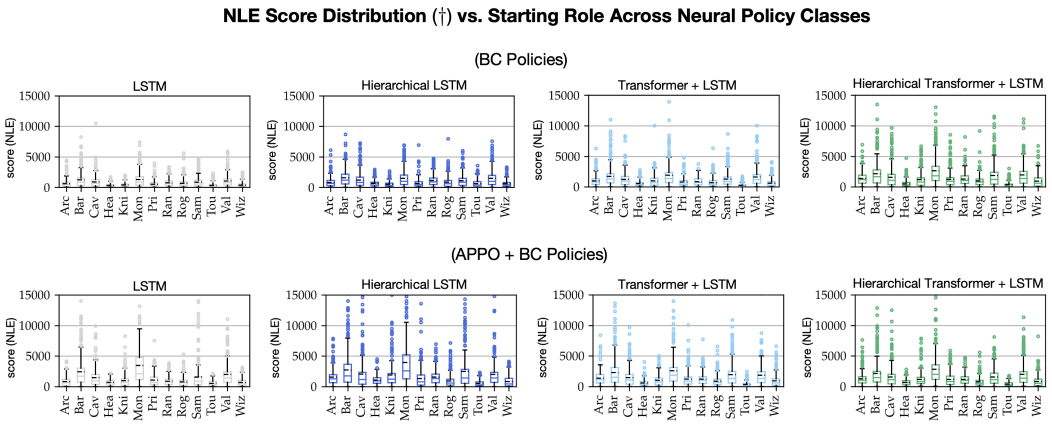

Figure 6: Aggregate NLE score breakdown versus player role. Our model refinements (hierarchy, Transformer, RL fine-tuning) show gains over the LSTM-based `CDGPT5` baseline [23] across all roles. As in Table 2, we employ (†) to confer that these score distributions were computed only for the top-performing neural policy seed (out of 6) across each model class.

BC and APPO+BC. In general, we observe that fine-tuning with RL improves the error-correction capability of models of all classes over their purely offline counterparts.

7 Conclusion and Discussion

In this work, we have developed a new technique for training NetHack agents that improves upon prior state-of-the-art neural models by 25%. We achieve this by first creating a new dataset called HiHack by accessing the best symbolic agent for NetHack. This dataset, combined with new architectures, allows us to build the strongest neural agent for NetHack currently available, to the best of our knowledge. More importantly, we analyze several directions to improve performance, including the importance of hierarchy, the role of large transformer models, and the boosts that RL could provide. Our findings are multifaceted and provide valuable insights for future progress in training neural agents in open-ended environments and potentially bridging the gap to symbolic methods.

- Hierarchy improves underfitting models. Prior LSTM based models severely underfit on NetHack. Adding hierarchical goal-directed strategy labels improves such models.
- Hierarchy hurts overfitting models. Transformer based models are able to overfit, even on our large HiHack dataset. Consequently, hierarchy hurts this class of models at test-time, with any gains resultant from the separation of demonstration data across separate goal-directed modes of behavior eclipsed by bilevel error accumulation.
- Reinforcement learning provides larger improvements on underfitting models. We obtain only minor improvements with using RL on our overfit Transformer models. However, the underfit LSTM models enjoy significant gains with RL, ultimately outperforming Transformer-based models.
- Scale alone is not enough. Our studies on increasing both model and dataset size (Figure 5) show sub-log-linear scaling laws. The shallow slope of the data "scaling laws" we observe indicates that successful imitation learning for NetHack will require more than just scaling up demonstrations.

Possible avenues for future exploration include: (a) methods for increasing the Transformer context length to give the agent a longer memory to aid exploration; (b) addressing the multi-modal nature of the demonstration data (i.e. quite different trajectories can lead to the same reward), which is a potential confounder for BC methods. Some forms of distributional BC (e.g. GAIL [26], BeT [51]) could help alleviate this issue.

References

[1] J. Abramson, A. Ahuja, A. Brussee, F. Carnevale, M. Cassin, S. Clark, A. Dudzik, P. Georgiev, A. Guy, T. Harley, F. Hill, A. Hung, Z. Kenton, J. Landon, T. P. Lillicrap, K. W. Mathewson, A. Muldal, A. Santoro, N. Savinov, V. Varma, G. Wayne, N. Wong, C. Yan, and R. Zhu. Imitating interactive intelligence. *arXiv*, abs/2012.05672, 2020. 3

[2] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches, 2016. 3

[3] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009. 3

[4] P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. 3

[5] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003. 3

[6] A. Billard, S. Calinon, R. Dillmann, and S. Schaal. Survey: Robot programming by demonstration. *Handbook of robotics*, 59(BOOK_CHAP), 2008. 3

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, S. Sidor, I. Sutskever, and R. S. Zemel. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 1

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 3

[9] D. Bruce, E. Hambro, M. Azar, and M. G. Bellemare. Learning about progress from experts. In *International Conference on Learning Representations*, 2023. 3

[10] A. Bulatov, Y. Kuratov, and M. Burtsev. Recurrent memory transformer. *Advances in Neural Information Processing Systems*, 35:11079–11091, 2022. 3

[11] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *CoRR*, abs/2106.01345, 2021. URL https://arxiv.org/abs/2106.01345. 3

[12] H. M. Clever, A. Handa, H. Mazhar, K. Parker, O. Shapira, Q. Wan, Y. Narang, I. Akinola, M. Cakmak, and D. Fox. Assistive tele-op: Leveraging transformers to collect robotic task demonstrations. *arXiv preprint arXiv:2112.05129*, 2021. 3

[13] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019. 3

[14] S. Dasari and A. Gupta. Transformers for one-shot visual imitation. *arXiv preprint arXiv:2011.05970*, 2020. 3

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, pages 4171–4186, 2018. doi: 10.18653/v1/N19-1423. 3

[16] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 3

[17] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. *CoRR*, abs/1802.06070, 2018. URL http://arxiv.org/abs/1802.06070. 3

[18] P. Florence, L. Manuelli, and R. Tedrake. Self-supervised correspondence in visuomotor policy learning. *IEEE Robotics and Automation Letters*, 5(2):492–499, 2019. 3

[19] C. Florensa, Y. Duan, and P. Abbeel. Stochastic neural networks for hierarchical reinforcement learning, 2017. 3

[20] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020. 3

[21] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1587–1596. PMLR, PMLR, 2018. 3

[22] C. Gulcehre, Z. Wang, A. Novikov, T. Le Paine, S. Gomez Colmenarejo, K. Zolna, R. Agarwal, J. Merel, D. Mankowitz, C. Paduraru, et al. Rl unplugged: Benchmarks for offline reinforcement learning. *arXiv e-prints*, pages arXiv–2006, 2020. 3

[23] E. Hambro, S. Mohanty, D. Babaev, M. Byeon, D. Chakraborty, E. Grefenstette, M. Jiang, D. Jo, A. Kanervisto, J. Kim, et al. Insights from the neurips 2021 nethack challenge. In *Advances in Neural Information Processing Systems*, 2021. 1, 2, 4, 5, 6, 7, 8, 9, 17, 19

[24] E. Hambro, S. Mohanty, D. Babaev, M. Byeon, D. Chakraborty, E. Grefenstette, M. Jiang, D. Jo, A. Kanervisto, J. Kim, et al. Dungeons and data: A large-scale nethack dataset. In *Advances in Neural Information Processing Systems*, 2022. 1, 2, 4, 7, 16, 17

[25] D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016. 17

[26] J. Ho and S. Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, volume 29, pages 4565–4573, 2016. 3, 9

[27] A. Jaegle, F. Gimeno, A. Brock, O. Vinyals, A. Zisserman, and J. Carreira. Perceiver: General perception with iterative attention. In *International conference on machine learning*, pages 4651–4664. PMLR, 2021. 3

[28] M. Janner, Q. Li, and S. Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34, 2021. 3

[29] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv*, abs/2001.08361, 2020. 6

[30] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 17

[31] A. Kumar, J. Fu, M. Soh, G. Tucker, and S. Levine. Stabilizing off-policy q-learning via bootstrapping error reduction. *Advances in Neural Information Processing Systems*, 32, 2019. 3

[32] A. Kumar, A. Zhou, G. Tucker, and S. Levine. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020. 3

[33] H. Kuttler, N. Nardelli, T. Lavril, M. Selvatici, V. Sivakumar, M. G. Bellemare, R. Munos, A. Graves, and M. G. Bellemare. The nethack learning environment. In *Advances in Neural Information Processing Systems*, 2020. 1, 2, 14, 18

[34] H. Le, N. Jiang, A. Agarwal, M. Dudík, Y. Yue, and H. Daumé III. Hierarchical imitation and reinforcement learning. In *International conference on machine learning*, pages 2917–2926. PMLR, 2018. 3

[35] S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020. 3

[36] A. Levy, G. Konidaris, R. Platt, and K. Saenko. Learning multi-level hierarchies with hindsight, 2017. 3

[37] Z. Mandi, F. Liu, K. Lee, and P. Abbeel. Towards more generalizable one-shot visual imitation learning. *arXiv preprint arXiv:2110.13423*, 2021. 3

[38] V. Mella, E. Hambro, D. Rothermel, and H. Küttler. moolib: A Platform for Distributed RL. 2022. URL https://github.com/facebookresearch/moolib. 7, 17

[39] O. Nachum, S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. *CoRR*, abs/1805.08296, 2018. URL http://arxiv.org/abs/1805.08296. 3

[40] O. Nachum, S. Gu, H. Lee, and S. Levine. Near-optimal representation learning for hierarchical reinforcement learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. 3

[41] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters. An algorithmic perspective on imitation learning. *arXiv preprint arXiv:1811.06711*, 2018. 3

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019. 17

[43] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018. 3

[44] X. B. Peng, Z. Ma, P. Abbeel, S. Levine, and A. Kanazawa. Amp: Adversarial motion priors for stylized physics-based character control. *ACM Transactions on Graphics (TOG)*, 40(4):1–20, 2021. 3

[45] A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, pages 7652–7662. PMLR, 2020. 7

[46] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988. 3

[47] R. Rahmatizadeh, P. Abolghasemi, L. Bölöni, and S. Levine. Vision-based multi-task manipulation for inexpensive robots using end-to-end learning from demonstration. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3758–3765. IEEE, 2018. 3

[48] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022. 3

[49] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3 (6):233–242, 1999. 3

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 7

[51] N. M. Shafiullah, Z. Cui, A. A. Altanzaya, and L. Pinto. Behavior transformers: Cloning k modes with one stone. *Advances in neural information processing systems*, 35:22955–22968, 2022. 3, 9

[52] T. Shankar, S. Tulsiani, L. Pinto, and A. Gupta. Discovering motor programs by recomposing demonstrations. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgHY0NYwr. 3

[53] A. Sharma, S. Gu, S. Levine, V. Kumar, and K. Hausman. Dynamics-aware unsupervised discovery of skills. *CoRR*, abs/1907.01657, 2019. URL http://arxiv.org/abs/1907.01657. 3

[54] Y. Tassa, D. Silver, J. Schrittwieser, A. Guez, L. Sifre, G. v. d. Driessche, S. Dieleman, K. Greff, T. Erez, and S. Petersen. Deepmind control suite. *arXiv preprint arXiv:1801.01294*, 2018. 1

[55] A. A. Team, J. Bauer, K. Baumli, S. Baveja, F. Behbahani, A. Bhoopchand, N. Bradley-Schmieg, M. Chang, N. Clay, A. Collister, et al. Human-timescale adaptation in an open-ended task space. *arXiv preprint arXiv:2301.07608*, 2023. 3

[56] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. 3, 17

[57] Y. Wu, G. Tucker, and O. Nachum. Behavior regularized offline reinforcement learning. *arXiv preprint arXiv:1911.11361*, 2019. 3

[58] A. Zeng, P. Florence, J. Tompson, S. Welker, J. Chien, M. Attarian, T. Armstrong, I. Krasin, D. Duong, V. Sindhwani, et al. Transporter networks: Rearranging the visual world for robotic manipulation. In *Conference on Robot Learning*, 2020. 3

[59] T. Zhang, Z. McCarthy, O. Jow, D. Lee, X. Chen, K. Goldberg, and P. Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. In *ICRA*, pages 5628–5635. IEEE, 2018. 3

[60] Y. Zhong, E. Hambro, E. Grefenstette, T. Park, M. Azar, and M. G. Bellemare. Improving policy learning via language dynamics distillation. In *Advances in Neural Information Processing Systems*, 2022. 2

[61] Y. Zhu, Z. Wang, J. Merel, A. Rusu, T. Erez, S. Cabi, S. Tunyasuvunakool, J. Kramár, R. Hadsell, N. de Freitas, et al. Reinforcement and imitation learning for diverse visuomotor skills. *arXiv preprint arXiv:1802.09564*, 2018. 3

A NetHack Learning Environment

Both NetHack and the NetHack Learning Environment (NLE) feature a complex and rich observation space. The full observation space of NLE consists of many distinct (but redundant) components: *glyphs*, *chars*, *colors*, *specials*, *blstats*, *message*, *inv_glyphs*, *inv_strs*, *inv_oclasses*, *screen_descriptions*, *tty_chars*, *tty_colors*, and *tty_cursor*.

The `HiHack` dataset, as well as all RL experiments in NLE conducted in this paper, consist of and rely solely upon the *tty∗* view of the game.

B Details on AutoAscend

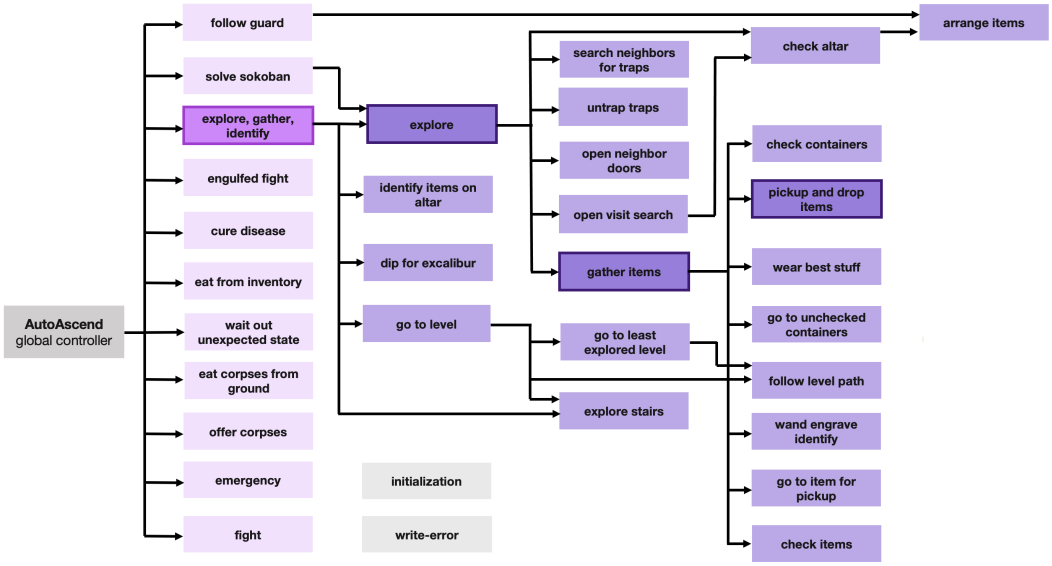

Figure 7: Full control flow structure of `AutoAscend`, separated across explicit *strategy* and *sub-strategy* routines. There are 13 possible "strategy" or hierarchical labels in `HiHack`, 11 representing the explicit *strategies* employed by the bot (in magenta) and two additional labels to handle extra-hierarchical behavior (in light-grey).

We include a comprehensive visualization of the full internal structure of `AutoAscend` in Figure 7. As indicated in the summary visualization of high-level `AutoAscend` strategies provided in Figure 2 of the main paper, the bot features 11 explicit, hard-coded *strategy* routines. These interface with other low-level *sub-strategy* routines, some which are re-used by multiple strategies or even multiple sub-strategies. One example of such a subroutine is the "arrange items" sub-strategy, which is called both by the "follow guard" strategy as well as by the "check altar" sub-strategy, which is itself a subroutine of the "explore" sub-strategy. When factorized across strategies and sub-strategies, the full structure of `AutoAscend` is a directed acyclic graph (DAG) with a maximal depth of 5 from the "root," i.e. the `AutoAscend` *global controller* "node" indicated in dark gray above, which is re-directs global behavioral flow across strategies via a predicate-matching scheme.

The `HiHack` dataset includes a hierarchical strategy *label* for each timestep of `AutoAscend` interaction. As a result, alongside the 11 explicit strategies of the bot, there are two additional labels present in the dataset, which account for extra-hierarchical behavior in `ttyrec` game records yielded by the augmented `ttyrec` writer employed for `HiHack` generation and loading. These are visualized in light gray in Figure 7. The first of these corresponds to the hard-coded initialization routine employed by `AutoAscend`, effectively serving as a twelfth (albeit implicit) strategy, while the second covers `ttyrec` timestep records with missing strategy values. Missing strategy values may reflect `ttyrec` writer errors, or advancement of the underlying NetHack state by NLE rather than by agent, which occurs e.g., during NLE's timeout-based termination of games [33]. Empirically, "write-error" strategy labels occur with very low-frequency in `HiHack`, representing less than $\approx 0.05\%$ of all data.

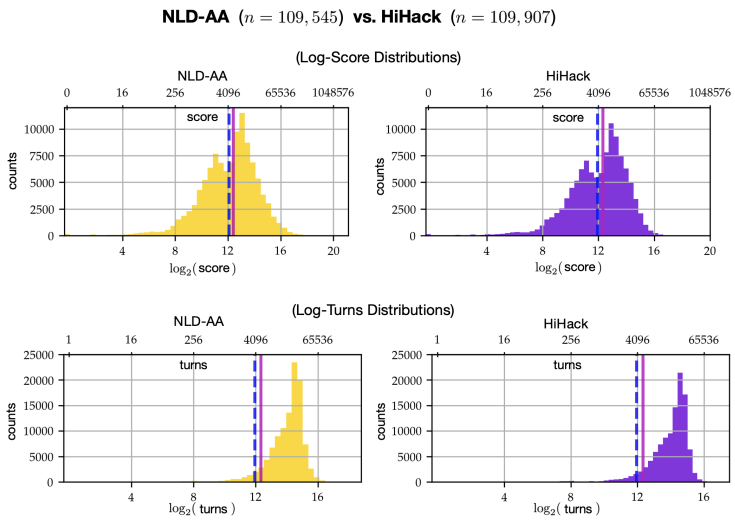

Figure 8: Distributional comparisons of basic `AutoAscend` game statistics in `NLD-AA` vs `HiHack`. Median and mean values (as reported in Table 1) are indicated respectively by vertical dashed-blue and solid-pink lines in each figure. *Top*: Log-score vs game counts in `NLD-AA` and `HiHack`. *Bottom*: Log-turns vs game counts in `NLD-AA` and `HiHack`.

C Details on HiHack

Generation All games in the HiHack dataset were recorded by running an augmented version of `AutoAscend` in NLE `v 0.9.0`. This augmented version of the `AutoAscend` source features the introduction of only a dozen extra lines of code that enable step-wise logging of the strategy trace behind each action executed by the bot. This strategy trace is recorded directly to game `ttyrecs` at each timestep via the addition of an extra channel to the C-based `ttyrec`-writer in the NetHack source code. Each game was generated via a unique NLE environment seed.

Game Statistics The comparison of the full log-score and log-turns distributions across `NLD-AA` and `HiHack` made in Figure 8 further supports the claim of high correspondence between the datasets made in Section 3.3. Figure 9 shows the distribution of strategies across a sample consisting of $\approx 10^8$ unique game transitions from `HiHack`. We observe coverage of all but the least frequent explicit strategy executed by `AutoAscend`: the "solve sokoban" routine, employed a means to gain yet more experience exclusively in highly advanced game states.

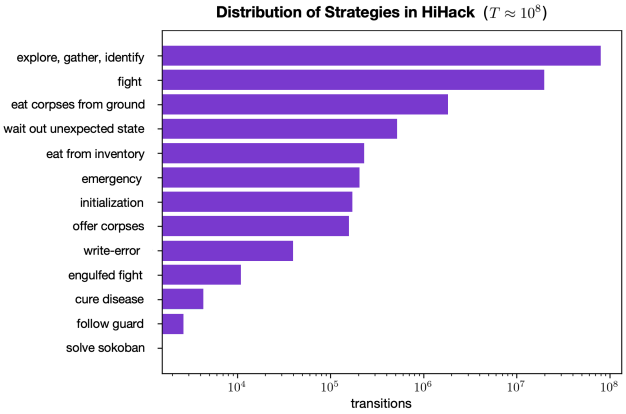

Figure 9: A visualization of the distribution of strategies across a sample of 4,300 `HiHack` games, containing a total of $\approx 10^8$ transitions.

Table 3: **Training hyperparameter configurations** across all BC and APPO + BC experiments. Hyperparameters are listed in alphabetical order. We employ (‡) to indicate hyperparameters only relevant for the corresponding hierarchical policy variants. The presence of the symbol '-' in lieu of a parameter value reflects the parameter's irrelevance for offline, BC experiments. All bolded hyperparameters were tuned. After tuning was complete, precisely the same sets of hyperparameters were employed to train models across individual policy classes belonging to the (LSTM)- and (Transformer + LSTM)-based model families explored in this paper, across all BC and APPO + BC experiments. Note that the abbreviation '`CE`' denotes the cross-entropy loss function.

	BC		APPO + BC	
Hyperparameter	LSTM	Transformer + LSTM	LSTM	Transformer + LSTM
actor batch size	-	-	512	256
adam beta1	0.9	0.9	0.9	0.9
adam beta2	0.999	0.999	0.999	0.999
adam eps	1.00E-07	1.00E-07	1.00E-07	1.00E-07
adam learning rate	0.0001	0.0002	0.0001	0.0001
appo clip baseline	1	1	1	1
appo clip policy	0.1	0.1	0.1	0.1
baseline cost	1	1	1	1
crop dim	18	18	18	18
discount factor	-	-	0.999	0.999
entropy cost	-	-	0.001	0.001
env max episode steps	-	-	100000	100000
env name	-	-	`challenge`	`challenge`
fn penalty step	-	-	constant	constant
grad norm clipping	4	1	4	1
inference unroll length	-	-	1	1
loss function	`CE`	`CE`	`CE`	`CE`
normalize advantages	-	-	✓	✓
normalize reward	-	-	✗	✗
num actor batches	-	-	2	2
num actor cpus	-	-	10	10
penalty step	-	-	0	0
penalty time	-	-	0	0
pixel size	6	6	6	6
reward clip	-	-	10	10
reward scale	-	-	1	1
RL loss coeff	-	-	1	0.001
strategy loss coeff (‡)	1	1	1	1
supervised loss coeff	1	1	0.001	1
`ttyrec` batch size	512	512	256	256
`ttyrec` cores	12	12	12	12
`ttyrec` **envpool size**	4	6	4	3
`ttyrec` **unroll length**	32	64	32	64
use prev action	✓	✓	✓	✓
virtual batch size	512	1024	512	512

D Training Details

Hyperparameters All relevant training hyperparameter values, across model families as well as BC vs APPO + BC experiment variants, are displayed in Table 3.

To kick-start all experiments, we employed the training hyperparameter values reported in Hambro et al. [24]. For several hyperparameters, however, additional tuning was conducted. These hyperparameters are indicated in bold in Table 3. Tuning across these hyperparameters was performed once for the "default" representative policy class from each of the LSTM and Transformer + LSTM model families for all but the Model Scaling experiments[1]. After tuning was complete, hyperparameter

[1]Prior to the start of these experiments, additional tuning of the "adam learning rate" and "`ttyrec` batch size" hyperparameters was conducted for the Transformer + LSTM (large) policy class. However, across the set of values tested, the same values were found to be optimal for this model configuration as for the default

configurations were fixed across all succeeding offline and combined offline + online experiments. Specifications of hyperparameter values swept over during tuning are provided in Table 4.

All models were trained with the `Adam` optimizer [30] and a fixed learning rate. We experimented with the introduction of a learning rate schedule for Transformer + LSTM models, but we found no additional improvements in policy prediction error or evaluation performance at the conclusion of training to be yielded by such a schedule.

Random Seeds For each of the hierarchical behavioral cloning, model parameter scaling, data scaling, and combined imitation and reinforcement learning experiments described in Sections 4, 5, and 6 of this paper, a total of 6 random seeds were run across all relevant policy classes. Randomized quantities included: policy parameter values at initialization, data loading and batching order, `HiHack` dataset subsampling (in data scaling experiments only), and initial environment seeding (in APPO + BC experiments only).

Training Infrastructure and Compute The RPC-based `moolib` library for distributed, asynchronous machine learning was employed across all experiments [38]. All data loading and batching was parallelized. Our model training code builds heavily upon the code open-sourced by Hambro et al. [24].

Experiments were run on compute nodes on a private high-performance computing (HPC) cluster equipped either with a NVIDIA RTX-8000 or NVIDIA A100 GPU, as well as 16 CPU cores. All policies were trained for a total of 48 hours. We detected no substantial differences in training frames-per-second (FPS) rates for both offline and online experiments across compute nodes in "speed-run" tests, provided nodes were under no external load. When running experiments, we did detect some variance in total optimization steps completed under the 48-hour constrained computational budget across seeds belonging to single policy classes, which we attribute to variance in external HPC cluster load during these runs.

Table 4: **Training hyperparameter tuning sweeps**. We specify the hyperparameter values tested during tuning sweeps conducted for the "default" representatives of each model class. Full specifications of final hyperparameter values employed in experiments are included in Table 3.

	Sweep Range
adam learning rate	$\{0.0001, 0.0002, 0.0005, 0.01\}$
discount factor	$\{0.9, 0.99, 0.999, 0.9999\}$
grad norm clipping	$\{0.1, 1, 4\}$
RL loss coeff	$\{0.001, 0.01, 1\}$
strategy loss coeff	$\{0.001, 0.01, 1, 10\}$
supervised loss coeff	$\{0.001, 0.01, 1\}$
ttyrec envpool size	$\{3, 4, 6\}$
ttyrec unroll length	$\{16, 32, 64, 128\}$
virtual batch size	$\{128, 256, 512, 1024\}$

E Model Architectures

A description of all model components and policy architectures is given in Tables 5 and 6, separated across (LSTM)- and (Transformer + LSTM)- model families. The `PyTorch` library was used for to specify all models, loss functions, and optimizers [42].

Additional Transformer Specifications All Transformer modules tested in this paper consist purely of "Transformer-Encoder" layers. Each layer is configured with 16 attention heads per attention mechanism, and layer normalization applied prior to all attention and feed-forward operations. A dropout of 0.1 is used during training [56]. Unlike the rest of the modules we employ, which use Exponential Linear Unit (`ELU`) activation functions as per the original CDGPT model architecture [23], our Transformer modules employ Gaussian Error Linear Unit (`GeLU`) activations [25].

Transformer + LSTM policy; hence, only a single set of hyperparameter values for the model family is reported here.

Table 5: **(LSTM)**-based policy architectural details. The final three columns indicate the presence (or absence) of each component across the relevant policy classes, whether trained with BC or APPO + BC.

							Policy Class		
Class	Type	Module(s)	Hidden Dim	Layers	Activ.	Copies	LSTM	LSTM + XXL dec	Hier LSTM
Enc	Message	`MLP`	128	2	`ELU`	-	✓	✓	✓
Enc	Blstats	`Conv-1D`, `MLP`	128	4	`ELU`	-	✓	✓	✓
Enc	Pixel Obs	`Conv-2D`, `MLP`	512	5	`ELU`	-	✓	✓	✓
Enc	Action Hist	`one-hot`	128	-		–	✓	✓	✓
Core	-	`LSTM`	512	1	-	-	✓	✓	✓
Dec	Default	`MLP`	512	1	-	-	✓	✗	✗
Dec	XXL	`MLP`	1024	2	`ELU`	-	✗	✓	✗
Dec	Hier Strat	`MLP`	128	1	-	-	✗	✗	✓
Dec	Hier Action	`MLP`	256	2	`ELU`	13	✗	✗	✓

Table 6: **(Transformer + LSTM)**-based policy architectural details. As in Table 5, the final three columns indicate the presence (or absence) of each component across the relevant policy classes, whether trained with BC or APPO + BC.

							Policy Class		
Class	Type	Module(s)	Hidden Dim	Layers	Activ.	Copies	Trnsfrmr + LSTM	Trnsfrmr + LSTM (large)	Hier Trnsfrmr + LSTM
Enc	Message	`MLP`	128	2	`ELU`	-	✓	✓	✓
Enc	Blstats	`Conv-1D`, `MLP`	128	4	`ELU`	-	✓	✓	✓
Enc	Pixel Obs	`Conv-2D`, `MLP`	512	5	`ELU`	-	✓	✓	✓
Enc	Action Hist	`one-hot`	128	-	-	-	✓	✓	✓
Enc	Recurrent	`LSTM` (frozen)	512	1	-	-	✓	✓	✓
Core	Default	`Trnsfrmr`	1408	3	`GeLU`	-	✓	✗	✓
Core	Large	`Trnsfrmr`	1408	6	`GeLU`	-	✗	✓	✗
Dec	Default	`MLP`	512	1	-	-	✓	✓	✗
Dec	Hier Strat	`MLP`	512	1	-	-	✗	✗	✓
Dec	Hier Action	`MLP`	512	2	`ELU`	13	✗	✗	✓

Context Length Models belonging to all policy classes from the (LSTM)-family are trained by sequentially "unrolling" batched-predictions. The length of this "unrolled" sequence is held fixed throughout training, and is specified by the value of the "`ttyrec` unroll length" hyperparameter in Table 3.

A fixed context length is also used to train the core Transformer modules of models belonging to classes from the (Transformer + LSTM)-family, similarly specified via the "`ttyrec` unroll length" hyperparameter. We found causally masking context in Transformer attention mechanisms to be greatly beneficial towards improving the generalization capability of models. The pre-trained frozen LSTM "recurrent encoder" module of these networks provides a very simple means of dramatically extending the effective context length of these models to cover full NetHack games, which may span hundreds of thousands of keypresses, without substantially slowing model training.

Hierarchical Policy Variants As alluded to in Tables 5 and 6, as well as in Section 4 of the main paper, all hierarchical policy variants are equipped with two sets of decoders: one high-level *strategy decoder* trained to predict the thirteen possible strategy labels in `HiHack`; as well as thirteen low-level *action decoders*, trained to predict actions corresponding to a single `HiHack` strategy across the 121-dimensional NLE action space [33]. Our hierarchical policies thus mimic the hierarchical structure of `Autoascend`, with an action decoder corresponding to each of the eleven, *explicit* strategies executed by the symbolic bot as well as two additional action decoders corresponding to the bot's "initialization" routine (an *implicit* twelfth strategy) and "write-errors," representing missing strategy

labels[2], respectively, as introduced in Figure 7. A full, diagrammatic illustration of the Hierarchical LSTM policy architecture is provided in Figure 3.

In all Hierarchical Behavioral Cloning (HBC) experiments, a BC loss was computed for the strategy decoder via ground-truth, batched `HiHack` strategy labels, while a separate BC loss was computed over a single action decoder over batched `HiHack` action labels, with this action decoder "selected" in an end-to-end fashion by the strategy decoder. The action decoder "selection" procedure was executed across batches by sampling predicted strategy indices from the strategy decoder with Gumbel-Softmax re-parameterization, thus preserving gradient flow across the bi-level hierarchical structure of policies during training. An illustration of the full Hierarchical LSTM policy architecture is provided in Figure 3.

The strategy-specific BC loss component was re-weighted (via the "strategy loss coefficent" hyperparameter, introduced in Table 3) and recombined with low-level action decoder losses to produce a single, overall HBC loss.

We resolved the presence of `ttyrec` transitions with missing strategy values, represented via the "write-error" hierarchical label in `HiHack`, by extending the action-space of the hierarchical strategy decoder to 13 and adding an additional hierarchical action decoder copy corresponding to this class of labels. It is possible that the performance of hierarchical policy variants can be further improved by instead filtering out all transitions with this property. We leave this evaluation for future work.

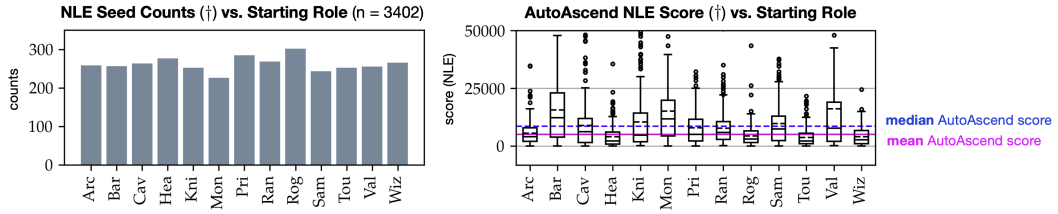

Figure 10: *Left*: The distribution of starting roles across the large-scale "in-depth evaluation." As described above, this evaluation was run for the top-performing neural policy seeds (out of 6) across model class. We observe a near-uniform distribution of possible NLE roles across random seeds. *Right*: `Autoascend` NLE Score distribution vs. starting role in the "in-depth evaluation." This figure is a companion to the visualizations of neural policy NLE scores across role in Figure 6. As in Figure 8, we indicate absolute median and mean values of `AutoAscend` NLE score in the "in-depth evaluation" with dashed-blue and solid-pink lines.

F Evaluation Details

For all policy classes belonging to the (LSTM)-model family, we observe monotonic improvements in the performance of models on withheld instances of NLE as a function of training samples; as a result, we employ the *final* training checkpoint of these policies when running evaluations across policy seeds. In contrast, due to overfitting, the generalization capabilities of (Transformer + LSTM)-family policies do not monotonically improve as a function of training samples. Thus, evaluations are conducted only for the "best" checkpoints corresponding to each policy seed, as evaluated on the basis of the rolling NLE score proxy metric. An in-depth description of this metric, as well as experiment training curves supporting the claims of over- and underfitting across model classes, can be found in Appendix G.

Two classes of evaluations are conducted in this paper for such checkpoints: a "standard evaluation" of policy NLE score across randomly sampled and withheld instances of environment, and an "in-depth evaluation," recording all metrics of game-play and employing precisely the same set of seeded environment instances to evaluate all policies.

The former policy evaluation procedure mirrors the one conducted during the NeurIPS 2021 NetHack Challenge Competition [23]. This is the procedure we employ to compute the mean and median NLE scores associated with policy seeds for all experiments in this paper as well as to compute the estimates of `AutoAscend` mean and median NLE score in Table 2, producing our core results.

[2]Please refer to our discussion in Appendix B for more details.

The latter policy evaluation procedure yields a suite of more fine-grained metrics for informed and "human-like" game-play in NetHack, such as maximal dungeon level reached and the total life-time of the agent. We run this evaluation procedure for each of the best-performing[3] seeds from each neural policy class, as well as for `AutoAscend`. Metrics computed with this procedure are denoted via the (†) symbol throughout the paper.

Standard Evaluation Policies are evaluated on a randomly seeded batch of 1024 (withheld) NLE games. Only the final NLE scores at the end of game-play are recorded.

In-Depth Evaluation Policies are evaluated across precisely the same seeded batch of 3402 (withheld) NLE games, i.e. all agents play precisely the same set of starting roles across the same NLE dungeon configurations, none of which are covered in `HiHack`. All games are recorded to the `ttyrec` data format, and can be streamed *post-facto*.

A visualization of the distribution of starting roles covered in this evaluation, as well as the corresponding `AutoAscend` score distributions (factorized by role across game instances), are shown in Figure 10.

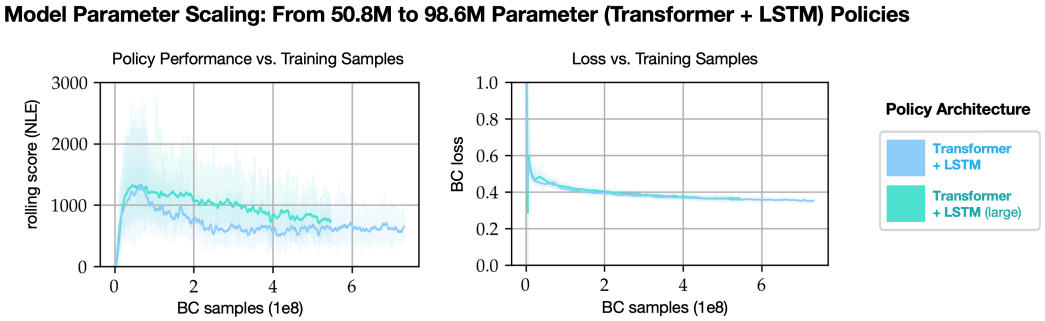

Figure 11: **Model parameter scaling experiment training curves**. In each plot, solid lines reflect point-wise averages across 6 random seeds, while shaded regions reflect point-wise min-to-max value ranges across seeds. *Left*: Rolling NLE evaluation score vs total BC training samples, for "default" and 2x deeper Transformer + LSTM policies. *Right*: BC loss vs total BC training samples.

G Training Curves

We provide training curves reflecting all conducted experiments. In Figures 11 and 12, we display both *rolling NLE scores* as well as BC loss curves as a function of training samples, across all model and data scaling experiments presented in Section 5. In Figure 13, we display aggregate rolling NLE scores as a function of training samples for all remaining BC and APPO + BC experiments, separated according to model family.

Rolling NLE Score The rolling NLE score metric introduced and displayed in the figures discussed here reflects an evaluation of policy performance on withheld NLE instances conducted continually during model training in a "rolling" fashion via a fixed number of workers. As such, this metric is biased towards shorter-length games, with the value of smoothed rolling score as a function of training sample confounded by the policy-specific relationship between NLE score and total game turns. Rolling NLE score is thus *not* interchangeable with the large-batch "standard evaluations" employed elsewhere in this paper and presented in-depth in Appendix F. However, unlike BC loss, it serves as an efficient and useful (if noisy) proxy measure of policy generalization over the course of training.

Model Parameter Scaling Training Curves As shown in Figure 11, the generalization capability of Transformer + LSTM policies (approximated via rolling NLE score) peaks soon after the start of training, decaying and flattening out as training proceeds despite continued improvement in BC

[3]As indicated by overall mean NLE score in the "standard evaluation" procedure.

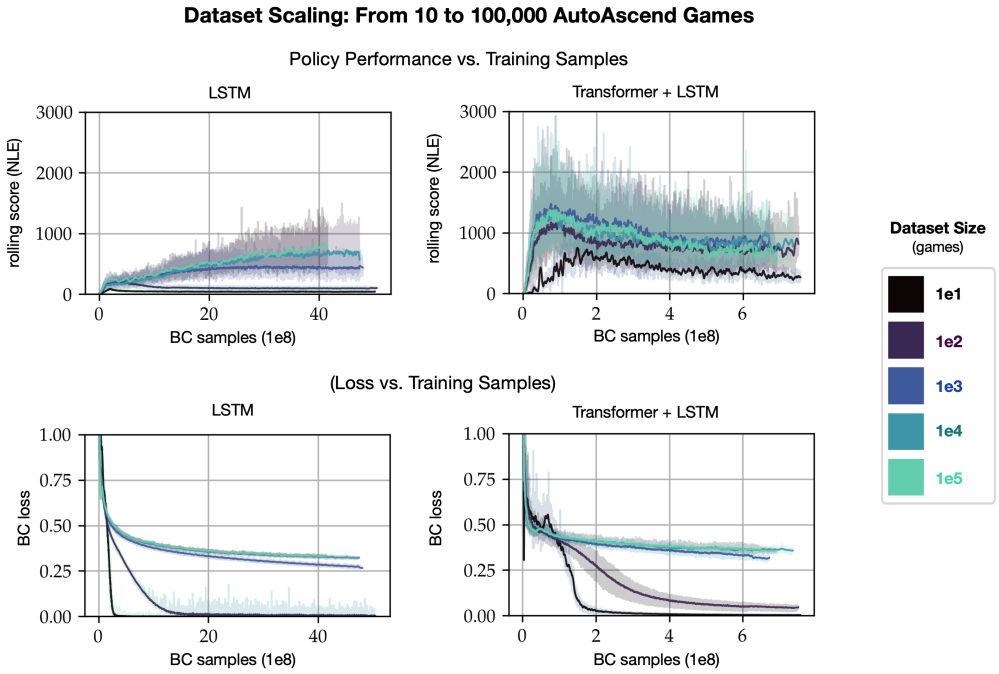

Figure 12: **Dataset scaling experiment training curves**. All experiments for a given dataset size were trained on a dataset sub-sampled without replacement from `HiHack`. The sub-sampling procedure was seeded. Training hyperparameters and model architectures were identical across all runs belonging to a single policy class. As in Figure 12, solid lines reflect point-wise averages across 6 random seeds, while shaded regions reflect point-wise min-to-max value ranges across seeds in all plots. *Top*: Rolling NLE evaluation score vs total BC training samples, across dataset sizes for the "LSTM" and "Transformer + LSTM" policy classes. *Bottom*: BC loss vs total BC training samples, across dataset sizes for the "LSTM" and "Transformer + LSTM" policy classes.

loss. This observation supports the claim made in Section 7 that Transformer-based models overfit to `HiHack`.

Despite the aforementioned noisy nature of rolling NLE score, the "Policy Performance vs. Training Samples" curves on the left of Figure 11 allude to our large-scale "standard evaluation" finding from Section 5; namely, that policy performance does not increase when model parameter count is scaled up, even after training hyperparameters are tuned. Similarly, the "Loss vs. Training Samples" curves on the right of this figure indicate nearly identical training errors across both models as a function of BC training samples.

Dataset Scaling Training Curves In Figure 12, we note that for datasets consisting of, or exceeding, 1,000 `AutoAscend` games, LSTM policies do not appear to overfit over the course of our BC experiments, with rolling NLE score consistently monotonically increasing as a function of BC training samples for all such policies. However, a positive relationship between dataset size and maximal rolling NLE score over training persists, indicating that the addition of more data does lead to measurable (if sub log-linear) improvements in policy generalization. An inspection of LSTM policy loss curves reveals a similar story. Losses across policy seeds trained on 10 and 100 `AutoAscend` games drop swiftly to values near zero, supporting this overfitting hypothesis.

The Transformer + LSTM policy loss curves in Figure 12 similarly reveal harsh overfitting for policies trained with 10 and 100 games. Interestingly, the generalization capability of these policies, again indicated by rolling NLE score over training, is vastly superior to that of their LSTM counterparts. Indeed, we observe that a Transformer + LSTM policy trained on just 100 games vastly outperforms a pure LSTM policy trained on 10x as many games. We attribute this gap to the frozen nature of the pre-trained LSTM component of the Transformer + LSTM policies employed as a "recurrent" encoder in these policies, and hypothesize that it is the static quality of the recurrent representation output

by this encoder which bolsters the generalization capability of the resultant models in exceptionally low-data regimes.

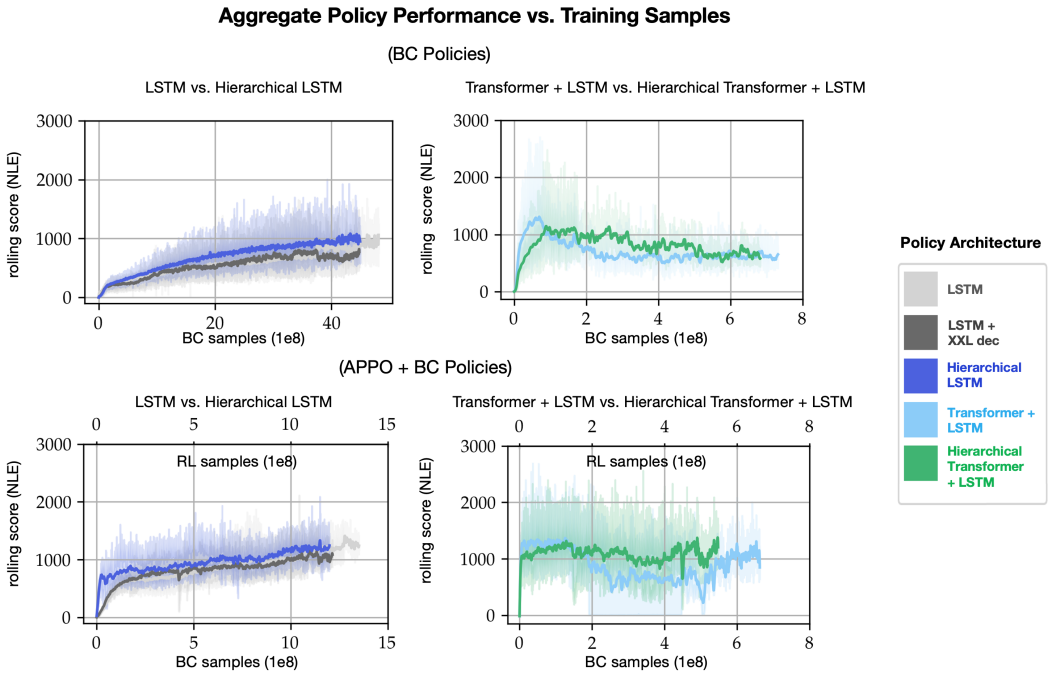

Figure 13: **Aggregate rolling NLE evaluation score curves for the central BC and APPO + BC experiments discussed in this paper**. Model parameter and dataset scaling training curves are omitted here on account of being visualized in Figures 11 and 12, respectively. For APPO + BC experiments, batch accumulation was employed to ensure that the ratio of RL to BC samples "seen" during training was 1:1. This property is indicated via the secondary x-axes of APPO + BC figures here, which show RL sample quantities. *Top*: Rolling NLE evaluation score vs total BC training samples in pure BC experiments, across non-hierarchical and hierarchical (LSTM) and (Transformer + LSTM)-based policy classes. *Bottom*: Rolling NLE evaluation score vs total BC training samples in APPO + BC experiments, across non-hierarchical and hierarchical (LSTM) and (Transformer + LSTM)-based policy classes.

Aggregate BC and APPO + BC Training Curves The "Aggregate Policy Performance vs Training Samples" curves of Figure 13 align with the general model family training trends previously observed in the scaling experiments. Notably, we find once again that (LSTM)-based policies' rolling NLE scores improve monotonically with training samples whether training is conducted with BC or APPO + BC, while this is not the case for (Transformer + LSTM)-based models. Indeed, the generalization properties' of policies belonging to this model family improve at the start of training before worsening as training proceeds across BC experiments. We interpret continued demonstration of these trends as further support for the claims of (LSTM)-underfitting and (Transformer + LSTM)-overfitting made in Section 7 of the main body of the paper as well as in Appendix F.

Moreover, we note that the introduction of an RL loss induces a particularly large amount of volatility in the rolling NLE score associated with (Transformer + LSTM)-based models, disrupting the monotonically decreasing relationship between rolling NLE score and training samples previously observed for these policies following $2 \cdot 10^8$ training samples in the pure BC experiments. This observation suggests that the performance of these policies is likely bottlenecked by an insufficient throughput of "on-policy" or interactive data, as compared to their LSTM counterparts, which train on 2x as many samples in our compute-time constrained experimental setting.

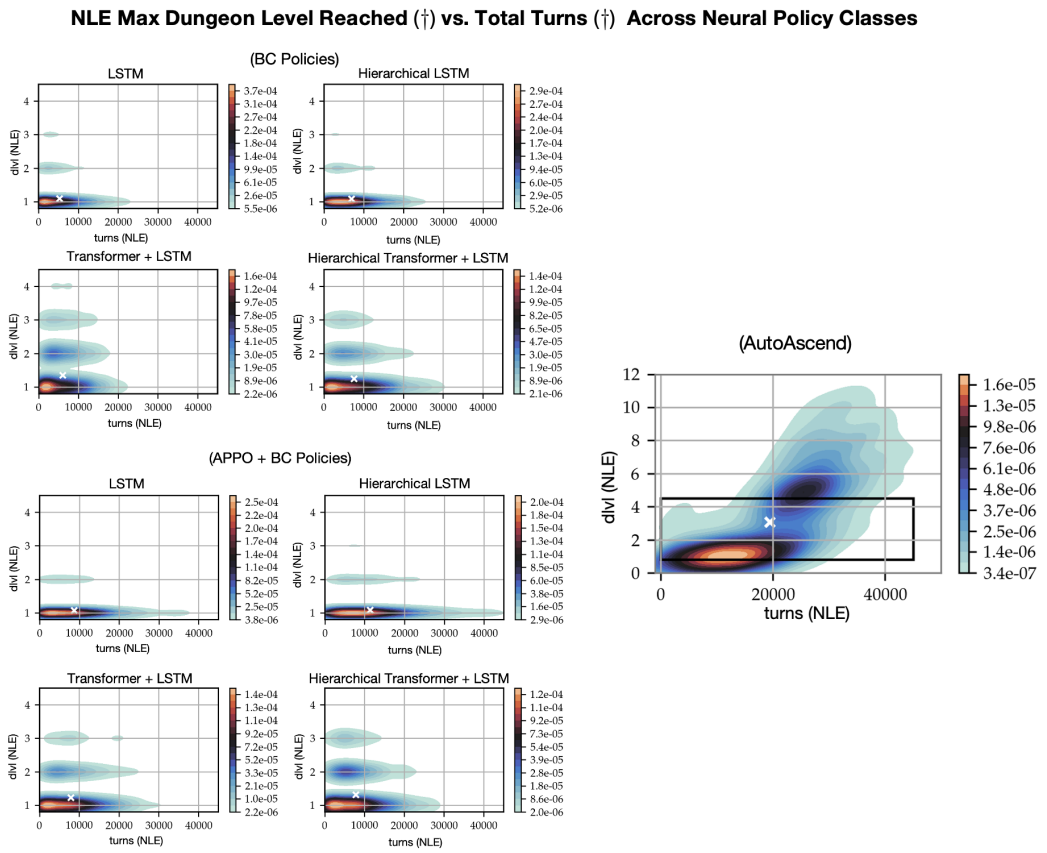

Figure 14: **Max dungeon level reached vs. total turns across "in-depth evaluation" games, visualized via 2-D contour density plots**. Contour densities are indicated by the color-bars accompanying each subplot. Mean quantity values (computed dimension-wise for all policies) across the "in-depth evaluation" batch are indicated by white '✗' symbols in each subplot. The symbolic bot's vastly superior ability to play longer and descend much further into the dungeon creates a separation in scale between it and its neural counterparts; as a result, for clarity, we indicate the max dungeon level vs turns subspace displayed in the neural policy contours with a bolded black rectangle in our visualization of `AutoAscend`'s behavior. *Top Left*: Best-performing BC neural policy seeds. *Bottom Left*: Best-performing APPO + BC neural policy seeds. *Right*: `AutoAscend`.

H Distributional Visualizations of Evaluation Results

Max Dungeon Level Reached vs. Total Turns In Figure 14, we supplement the absolute mean and median max-dungeon level and agent life-time (in game turns) statistics introduced in Table 2 with 2-D distributional visualizations of both the raw values of these metrics as well as their mutual inter-relationship, evaluated via $n = 3402$ seeded NLE games run individually for all policy class representatives in our "in-depth evaluation."

The `AutoAscend` 2-D contour plot on the right of this figure demonstrates an interesting emergent property of the bot's behavior: the high-level "descent" behavior of `AutoAscend` appears to fall along one of two modes across games. In the first of these modes, the bot spends a very large amount of turns on dungeon level 1, and avoids descending further into the dungeon before the end of the game. In the second mode, the bot begins to rapidly descend fairly deep into the dungeon, reaching as far as level 11. The overall relationship between `AutoAscend`'s total in-game life-time and max-dungeon level reached appears to be roughly quadratic.

In contrast, none of the neural policy class representatives tested here comes close to achieving `AutoAscend`'s secondary behavior mode. A large majority of games for all neural policy classes appear to end on the first level of the dungeon, with policies very rarely surviving as long on this

level as `AutoAscend` in games belonging to its corresponding behavior mode. This suggests that neural policies may be failing to master the very low-level behaviors of the bot, even when these behaviors are factorized across `AutoAscend` strategies, as in the case of hierarchical policy variants.

Nevertheless, the qualitative performance of hierarchical policies clearly improves upon that of non-hierarchical models, with the Hierarchical Transformer + LSTM policy trained with BC both surviving longer of dungeon level 1 and descending with higher frequency than other BC-trained policy representatives. Furthermore, this qualitative behavior appears to be strengthened when interaction is added into the mix, with the mode centered on "dungeon level 2" increasing in density for the APPO + BC variant of the Hierarchical Transformer + LSTM representative policy.

Taken together, these sets of observations lead us to hypothesize that the quality of neural policies trained with imitation learning on extremely complex, long-horizon tasks like NetHack may be further improved with increases in the scale of hierarchically-informed behavioral factorization, beyond the scale of factorization explored in this paper. We consider this to be a very exciting direction for future work.

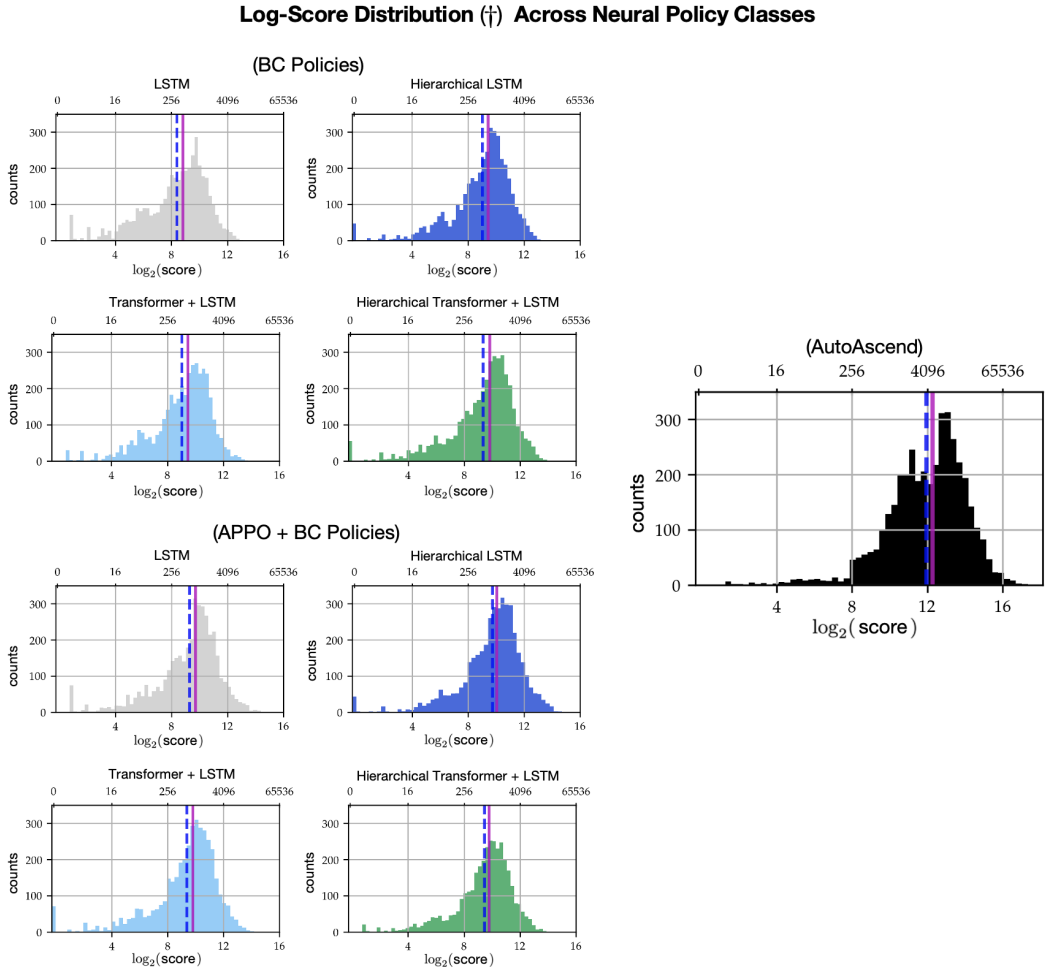

Figure 15: **Log-score distribution across "in-depth evaluation" games**. As in Figures 8 and 10, we indicate absolute median and mean values of policies' NLE scores in the "in-depth evaluation" with dashed-blue and solid-pink lines. The introduction of an RL loss reduces the "mass" associated with the left-tail of log-score and increases the "mass" associated with the right-tail across all neural policy classes, inducing right-ward shifts in median and mean scores, though difficult to perceive in this figure on account of the log-scale of the x-axis. We refer the reader to Table 2 in the main paper for aggregate absolute numerical values of NLE score. *Top Left*: Best-performing BC neural policy seeds. *Bottom Left*: Best-performing APPO + BC neural policy seeds. *Right*: `AutoAscend`.

Log-Score Distributions We conclude this supplementary analysis with a visualization of log-score distributions across neural policy classes in Figure 15, computed over the "in-depth evaluation" seeded games. The trends displayed in this figure align with those described and demonstrated previously. Improvements in model architecture as well as the introduction of hierarchy and interactive learning lead to a re-distribution of "mass" between left and right tails of distribution, with the overall counts of "low score" games decreasing and "high score" games increasing when these improvements are applied. The gap to `AutoAscend` is significantly reduced, but not bridged.