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

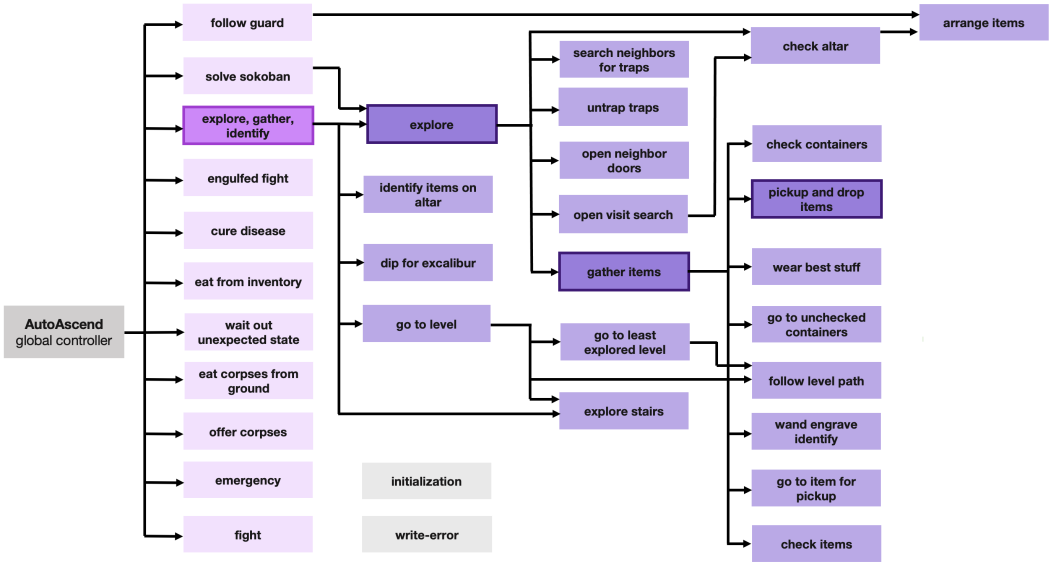

Figure 7: Full control flow structure of `AutoAscend`, separated across explicit *strategy* and *sub-strategy* routines. There are 13 possible "strategy" or hierarchical labels in `HiHack`, 11 representing the explicit *strategies* employed by the bot (in magenta) and two additional labels to handle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

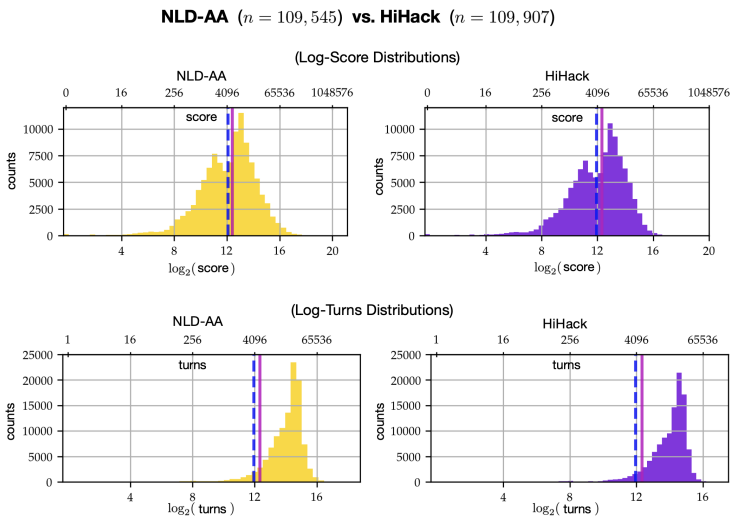

Figure 8: Distributional comparisons of basic `AutoAscend` game statistics in `NLD-AA` vs `HiHack`. Median and mean values (as reported in Table 1) are indicated respectively by vertical dashed-blue and solid-pink lines in each figure. *Top*: Log-score vs game counts in `NLD-AA` and `HiHack`. *Bottom*: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

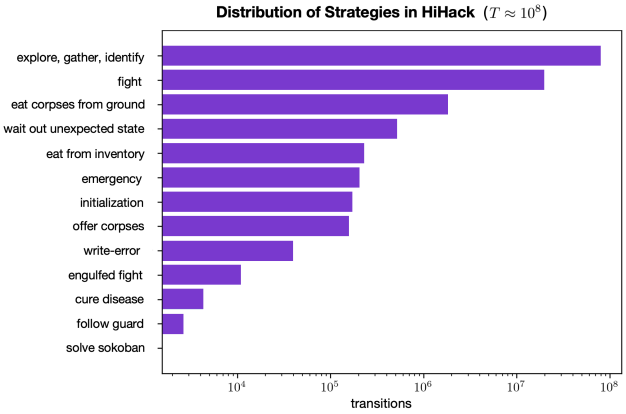

Figure 9: A visualization of the distribution of strategies across a sample of 4,300 `HiHack` games, containing a total of $\approx 10^8$ transitions.

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.

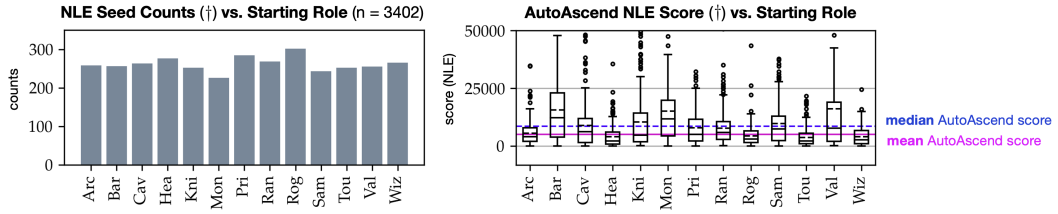

Figure 10: *Left*: The distribution of starting roles across the large-scale "in-depth evaluation." As described above, this evaluation was run for the top-performing neural policy seeds (out of 6) across model class. We observe a near-uniform distribution of possible NLE roles across random seeds. *Right*: `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.

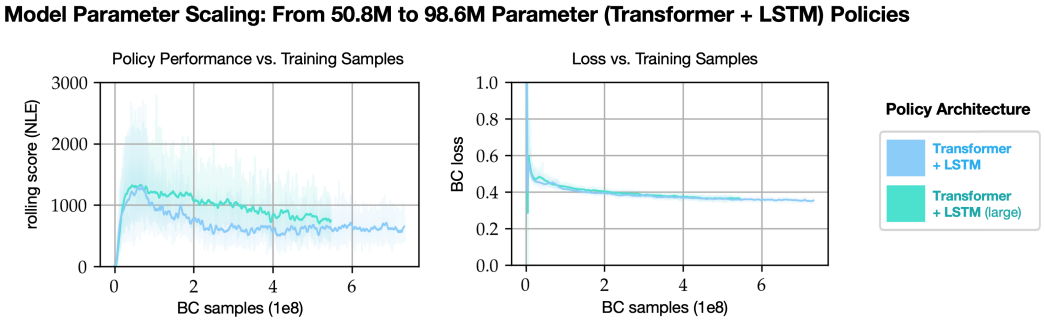

Figure 11: **Model parameter scaling experiment training curves**. In each plot, solid lines reflect point-wise averages across 6 random seeds, while shaded regions reflect point-wise min-to-max value ranges across seeds. *Left*: Rolling NLE evaluation score vs total BC training samples, for "default"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

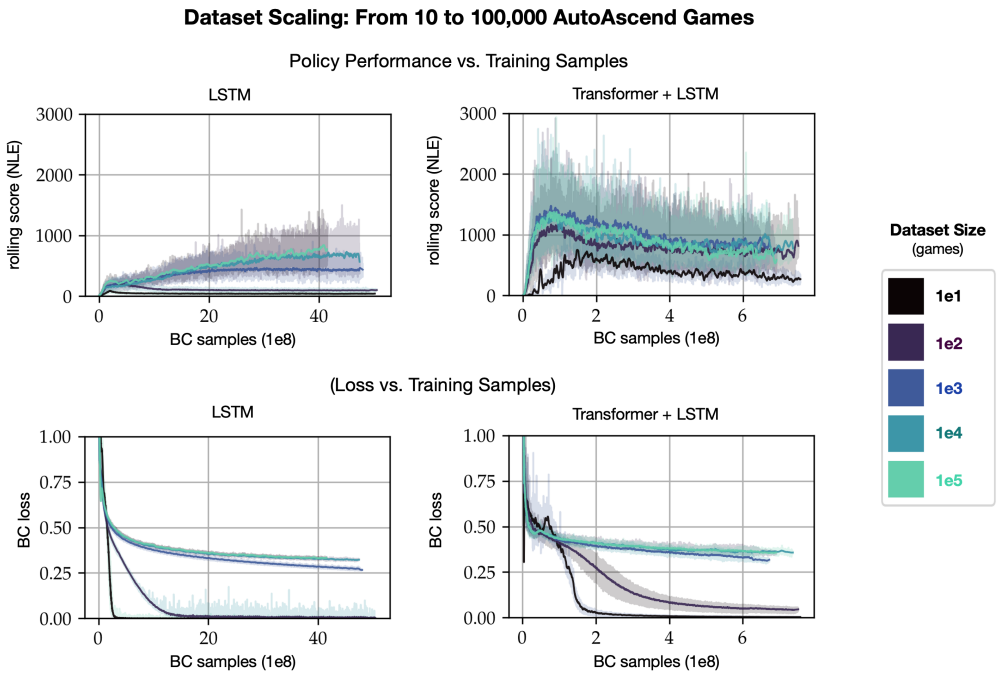

Figure 12: **Dataset scaling experiment training curves**. All experiments for a given dataset size were trained on a dataset sub-sampled without replacement from `HiHack`. The sub-sampling procedure was seeded. Training hyperparameters and model architectures were identical across all runs belonging 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.

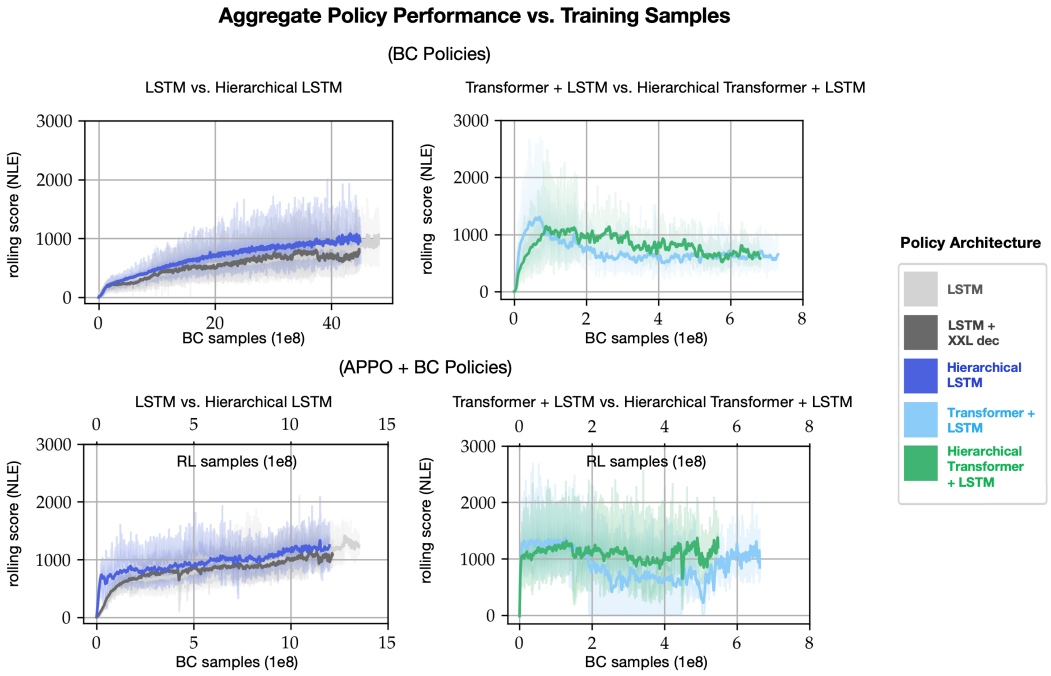

Figure 13: **Aggregate rolling NLE evaluation score curves for the central BC and APPO + BC experiments discussed in this paper**. Model parameter and dataset scaling training curves are omitted here on account of being visualized in Figures 11 and 12, respectively. For APPO + BC experiments, batch accumulation 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.

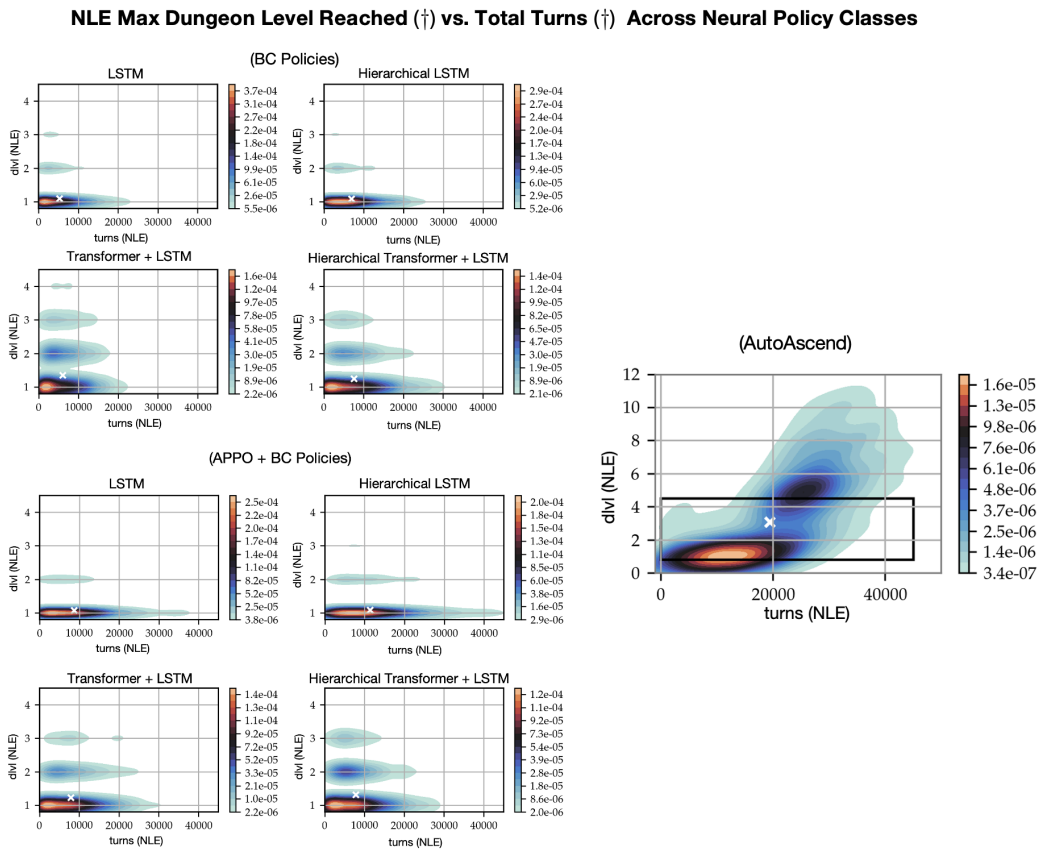

Figure 14: **Max dungeon level reached vs. total turns across "in-depth evaluation" games, visualized via 2-D contour density plots**. Contour densities are indicated by the color-bars accompanying each subplot. Mean quantity values (computed dimension-wise for all policies) across the "in-depth evaluation" 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.

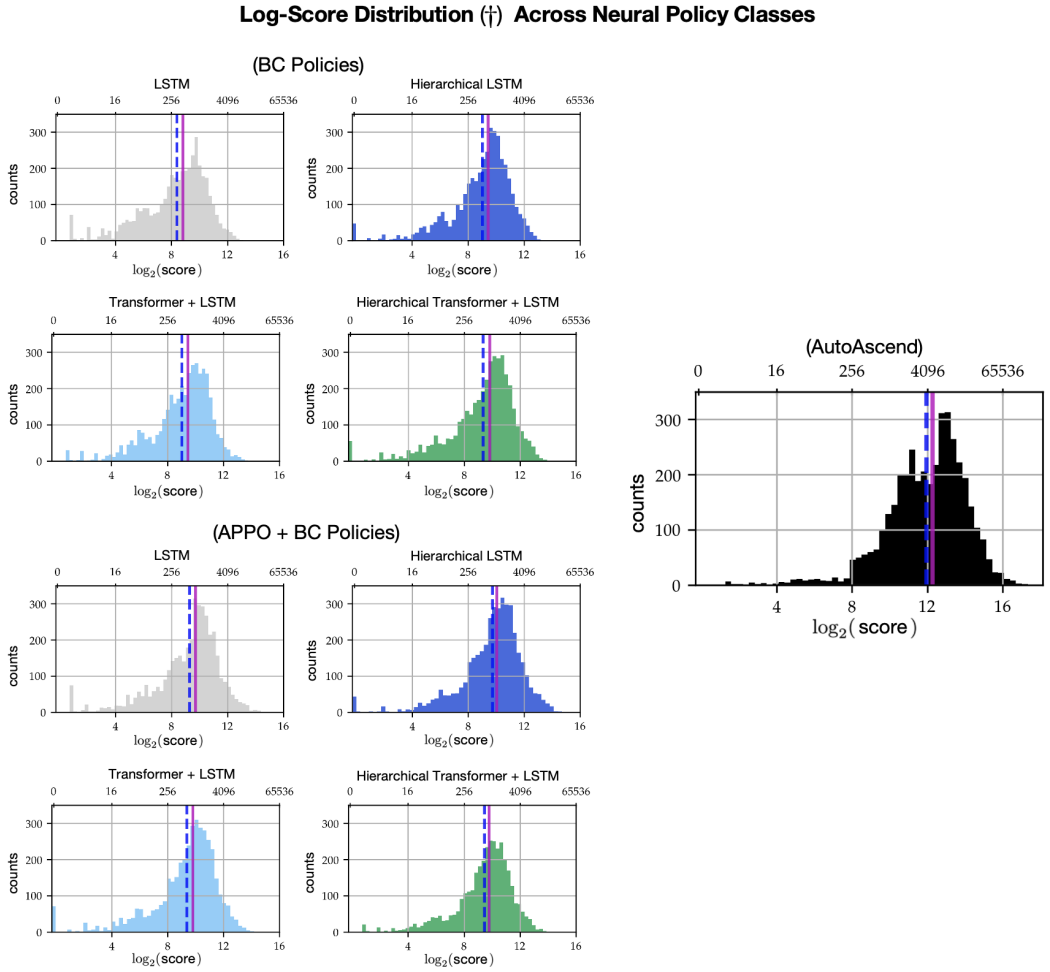

Figure 15: **Log-score distribution across "in-depth evaluation" games**. As in Figures 8 and 10, we indicate absolute median and mean values of policies' NLE scores in the "in-depth evaluation" with dashed-blue and solid-pink lines. The introduction of an RL loss reduces the "mass" associated with the 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.