A Accessing the Dataset

For the duration of the review the information on how to download, install and use NLD will be available at https://github.com/dungeonsdatasubmission/dungeonsdata-neurips2022. This link contains a README.md file with instructions on how to download zip files for NLD-AA and NLD-NAO, as well as instructions on how to install and use the TtyrecDataset, and includes code used in the running of experiment.

After review, this information will migrate to the README.md for the NLE repo, hosted at https://github.com/facebookresearch/nle.

Hosting & Maintenance Plan The dataset will eventually be hosted on a separate, dedicated site, with infrastructure akin to that provided on https://ai.facebook.com/datasets/. This site will be permanently available for the foreseeable future, with links to the paper and NLE code. The site is currently planned for launch in advance of the NeurIPS 2022 main conference, links will be available on the NLE repo's README.md.

B License

Data is provided under the NetHack General Public License - A GPL style license that is used to covered the NetHack Game since 1989. The license can be found at https://github.com/facebookresearch/nle/blob/main/LICENSE, and is used by both the official NetHack repository[13] and NLE.

C NLE v0.9.0

NLE v0.9.0 will be the latest version of NLE released, and will be released with NLD. The release branch is will be cut from main branch on the repo, hosted at (https://github.com/facebookresearch/nle/), and deployed to PyPI. The branch currently holds all the changes needed for compatibility with NLD, including the TtyrecDataset. A full changelog will be made public after release but the changes include:

- New ability to record ttyrec3.bz2 files directly from NLE. Previous versions recorded a modified ttyrec.bz2.
- The additional logging to an xlogfile for each episodes that finish naturally (*i.e.* when done is True).
- A C++ Converter class to load ttyrec.bz2/ttyrec3.bz2 files directly in to NumPy arrays, based on a V100 Terminal emulator (from libtmt[14]).
- The TtyrecDataset class, to marshall the Converter objects into an Torch.IterableDataset interface, and handle metadata.

D The ttyrec format

The ttyrec is a file format that has historically been used to store recordings of terminal-based NetHack games.[15] The format consists of a stack of *frames* where each *frame* consists of a 12-byte header immediately followed by a variable-size buffer of terminal instructions. The header contains 8 bytes of time information (storing *when* these updates were recorded) and 4 bytes indicating the size of the subsequent buffer (storing *what* these updates were). The buffer's contents are then the instructions to be fed to a terminal that will be rendering ttyrec. As such, this buffer will then generally consist of terminal-specific Escape Sequences[16] and text to be printed.

NLE's ttyrec3 has a 13-byte header, adding 1-byte channel to indicate *what kind* of content is in the buffer: 0 for terminal output; 1 for terminal input (i.e the keypresses corresponding to actions in the

[13] https://github.com/NetHack/NetHack
[14] https://github.com/deadpixi/libtmt
[15] https://nethackwiki.com/wiki/Ttyrec
[16] https://en.wikipedia.org/wiki/Escape_sequence

Figure 3: A screenshot of a tool to read raw `ttyrec.bz2`/`ttyrec3.bz2` files, included in `NLE v0.9.0`. The numbers in green represent the frame count for each channel [0, 1, 2], followed by the timestamp, and the buffer contents in braces: channel `0` is displayed in yellow braces, `1` in blue braces, and `2` in pink braces. Terminal Escape Sequences are printed in dark grey.

game); and, `2` for in-game score. These latter two channels are written just after and just before an action is taken (respectively).

It is worth noting that *frames* with channel `0` are written whenever NetHack intends to write to the screen, and therefore several may written before an action is required. An example of such an event occurs when zapping wand which animates a beam from the player. Such animations may result in a stack of frames with channels that resemble something like `[0 2 1 0 0 0 2 1 ...]`, where several different states might correspond to one action. In this case `ttyrec3.bz2` rendering loads the final state of the of the terminal NumPy arrays, alongside the score and action, while `ttyrec.bz2` rendering loads every state/frame, having no clear delineation of where actions take place.

To further inspect `ttyrec.bz2` and `ttyrec3.bz2`, `NLE v0.9.0` comes with an adapted `read_tty.py` script that displays the raw, decompressed `ttyrec`/`ttyrec3` contents, shown in Figure 3.

E `NLD` Gameplay Metadata Reference

The gameplay metadata in `NLD` is identical to that stored NetHacks's `xlogfile` at the end of a game, with the addition of a *gameid* which is used by `NLD` to identify games. The metadata consists of:

1. *gameid* `(int)` - A unique id for the game, created by the local database.
2. *version* `(str)` - The version of NetHack played.
3. *points* `(int)` - The final in-game score of the episode.
4. *deathdnum* `(int)` - The dungeon number where you died. These correspond to the dungeons found in `dungeon.def` file:
   - `0` - The Dungeons of Doom
   - `1` - Gehennom
   - `2` - The Gnomish Mines
   - `3` - The Quest
   - `4` - Sokoban
   - `5` - Fort Ludios
   - `6` - Elemental Planes
5. *deathlev* `(int)` - The dungeon level where you died.[17]
6. *maxlvl* `(int)` - The deepest dungeon level you reached.
7. *hp* `(int)` - The number of hit points you ended the game with.
8. *maxhp* `(int)` - The number of hit points you'd have had at "full health" at the game end.

[17] For more, see `https://nethackwiki.com/wiki/Mazes_of_Menace`

9. *deaths* (int) - The number of times you died.[18]
10. *deathdate* (int) - The day the game ended (format YYYYMMDD).
11. *birthdate* (int) - The day the game started (format YYYYMMDD).
12. *uid* (int) - An id used (in conjunction with player name) to identify players (for save games etc).
13. *role* (str) - The Role of the player. The 13 roles are:
    - Arc - Archaelogist
    - Bar - Barbarian
    - Cav - Cave(wo)man
    - Hea - Healer
    - Kni - Knight
    - Mon - Monk
    - Pri - Priest(ess)
    - Ran - Ranger
    - Rog - Rogue
    - Sam - Samurai
    - Tou - Tourist
    - Val - Valkyrie
    - Wiz - Wizard
14. *race* (str) - The Race of the player. The 5 races are:
    - Dwa - Dwarf
    - Elf - Elf
    - Gno - Gnome
    - Hum - Human
    - Orc - Orc
15. *gender* (str) - The Gender of the player. The 2 genders are:
    - Fem - Female
    - Mal - Male
16. *align* (str) - The Alignment of the player. The 3 alignments are:
    - Cha - Chaotic
    - Law - Lawful
    - Neu - Neutral
17. *name* (str) - The name of the player. These are pseudonymised in the database for NLD-NAO.
18. *death* (str) - A description of the manner in which the player died or ended the game.
19. *conduct* (str) - a bitfield encoding the conducts[19] completed in the game. The bitfield encodes:
    - 0x001: Foodless
    - 0x002: Vegan
    - 0x004: Vegetarian
    - 0x008: Atheist
    - 0x010: Weaponless
    - 0x020: Pacifist
    - 0x040: Illiterate
    - 0x080: Polypileless
    - 0x100: Polyselfless

[18] In rare cases, you can die more than once. See https://nethackwiki.com/wiki/Amulet_of_life_saving

[19] https://nethackwiki.com/wiki/Conduct

- `0x200`: Wishless
- `0x400`: Artifact wishless
- `0x800`: Genocideless

20. *turns* `(int)` - The number of in-game turns played by the player. This may not correspond to transitions, as several moves do not advance the in-game clock (eg checking the inventory or moving into a wall).
21. *achieve* `(str)` - a bitfield encoding the achievements[20] attained in the game. The bitfield encodes:
    - `0x0001`: Got the Bell of Opening
    - `0x0002`: Entered Gehennom
    - `0x0004`: Got the Candelabrum of Invocation
    - `0x0008`: Got the Book of the Dead
    - `0x0010`: Performed the Invocation
    - `0x0020`: Got the Amulet of Yendor
    - `0x0040`: Was in the End Game
    - `0x0080`: Was on the Astral Plane
    - `0x0100`: Ascended
    - `0x0200`: Got the Luckstone at Mines' End
    - `0x0400`: Finished Sokoban
    - `0x0800`: Killed Medusa
    - `0x1000`: Zen conduct intact
    - `0x2000`: Nudist conduct intact
22. *realtime* `(int)` - the duration of the game in seconds
23. *starttime* `(int)` - the start time of the game as an (epoch) unix timestamp.
24. *endtime* `(int)` - the end time of the game as an (epoch) unix timestamp.
25. *gender0* `(str)` - the starting gender of the player. Same options as *gender*.
26. *align0* `(str)` - the starting alignment of the player. Same options as *align*.
27. *flags* `(str)` - a bitfield encoding some additional state of the game. The bitfield encodes:
    - `0x1`: Wizard mode
    - `0x2`: Discover mode
    - `0x4`: Never loaded a Bones file

F `NLD` Observations Reference

The `TtyrecDataset` loads minibatches of observations, with batch size B, and sequence/unroll length T. The following observations are present in each minibatch dictionary:

Name	Type	Shape	Description
`tty_chars`	`np.uint8`	[B, T, H, W]	The on-screen characters (default screen size 80 x 24).
`tty_colors`	`np.int8`	[B, T, H, W]	The on-screen colors for each character.
`tty_cursor`	`np.int16`	[B, T, 2]	The coordinates of the on-screen cursor.
`timestamps`	`np.int64`	[B, T]	The time each frame was recorded.
`gameids`	`np.int32`	[B, T]	The *gameid* for the episode being rendered.
`done`	`np.uint8`	[B, T]	An indicator that this frame is from a new *gameid*.
`scores*`	`np.int32`	[B, T]	The in-game score at this time (*i.e.* before keypress).
`keypresses*`	`np.uint8`	[B, T]	The keypress the player made in response to this state (*i.e.* after seeing `tty_*`, `scores`, etc).

The observations marked with an asterisk (*) are only available in `ttyrec3.bz2` datasets.

[20] `https://nethackwiki.com/wiki/Xlogfile`

G `TtyrecDataset` Reference

G.1 Design Overview

To load minibatches, the `TtyrecDataset` performs two main actions: first, at construction time, it obtains the list of gameids and `ttyrec.bz2`/`ttyrec3.bz2` filepaths that constitute dataset, and second, at usage time, it constructs an Python generator[21] that will sequentially load these files into fixed NumPy arrays.

The first of these actions is performed by querying a local *sqlite3* database file, which acts as the metadata information hub for all datasets. This database file need only be populated once, and from thenceforth can be copied, shared or swapped out, without issue. For more on how to add datasets see the Tutorial Notebook[22], or Appendix G.4. The metadata in this database includes the location of files for each game, the association of which games are in which dataset, and the gameplay metadata for each game (see Appendix E). For more on the database schema see Appendix G.2.

The second of these actions is performed by the `TtyrecDataset` itself, by creating a lightweight C++ `Converter` object for each batch index in a generator, and loading from these objects when `next` is called. This process can be modified in a variety of different ways, including looping forever, shuffling, or using a threadpool. For more on the way these can be modified see Appendix G.3.

G.2 Database Layout

Table 3 shows the layout of the database, where fields of the same name map directly to each other. Each dataset is associated with a string (dataset_name), and once populated the dataset can be retrieved with this string.

G.3 `TtyrecDataset` API

The `TtyrecDataset` is documented primarily through Python docstrings, which provide a helpful way automatically documenting code objects, that can be queried by using Python's `help` function. An example usage can be clearly seen in the tutorial notebook[23] supplied with the submission, that will also migrate to `NLE`.

The `TtyrecDataset` accepts the following arguments, with all but first taking reasonable default values:

1. `dataset_name` - `(str)` - The name of the dataset to load. Each datasets is associated with a unique name.
2. `batch_size` - `(int)` - The number of simulataneous episodes to load (*i.e.* size B).
3. `seq_length` - `(int)` - The number of steps to unroll for each episode in a minibatch (*i.e.* size T).
4. `rows` - `(int)` - The row size of the terminal to emulate (*i.e.* size H).
5. `cols` - `(int)` - The column size of the terminal to emulate (*i.e.* size W).
6. `dbfilename` - `(str)` - The path to the database file used.
7. `threadpool` - `(Object)` - An executor class that has a `map` function (*i.e.* `concurrent.futures.ThreadPoolExecutor`.
8. `gameids` - `(List[str])` - A predetermined list of *gameids* to iterate through.
9. `shuffle` - `(bool)` - Whether to shuffle the order of *gameids* before iterating.
10. `loop_forever` - `(bool)` - Whether to start again at the beginning of *gameids* when the iterator runs out. If `False`, empty batch indexes are padded with `0`.
11. `subselect_sql` - `(str)` - A SQL string to dynamically generate subdataset, by select *gameids*.

[21] https://peps.python.org/pep-0255/

[22] https://github.com/dungeonsdatasubmission/dungeonsdata-neurips2022/blob/main/usage_tutorial_notebook.ipynb

[23] https://github.com/dungeonsdatasubmission/dungeonsdata-neurips2022/blob/main/usage_tutorial_notebook.ipynb

Table 3: The layout of the *sqlite3* database used by the `TtyrecDataset`. The `roots` table stores information about the root directory of the files in a dataset, as well as `ttyrec` version, while `ttyrecs` table stores the data about the files and what games they belong to. The `datasets` table then associates the gameids to their dataset, and `games` contains the gameplay metadata for each gameid.

`roots`	
dataset_name	TEXT
root	TEXT
ttyrec_version	INTEGER

`datasets`	
gameids	INTEGER
dataset_name	TEXT

`ttyrecs`	
path	TEXT
part	INTEGER
size	INTEGER
mtime	REAL
gameid	INTEGER

`meta`	
ctime	REAL
mtime	REAL

`games`	
gameid	INTEGER
version	TEXT
points	INTEGER
deathdnum	INTEGER
deathlev	INTEGER
maxlvl	INTEGER
hp	INTEGER
maxhp	INTEGER
deaths	INTEGER
deathdate	INTEGER
birthdate	INTEGER
uid	INTEGER
role	TEXT
race	TEXT
gender	TEXT
align	TEXT
name	TEXT
death	TEXT
conduct	TEXT
turns	INTEGER
achieve	TEXT
realtime	INTEGER
starttime	INTEGER
endtime	INTEGER
gender0	TEXT
align0	TEXT
flags	TEXT

12. `subselect_args - (Objects)` - Arguments to any SQL query provided in `subselect_sql`.

G.4 Adding Datasets

Adding datasets to a the database is a very easy "one-off" operation. Examples are shown in the Tutorial Notebook and also in Figure 4.

Since the process for adding `ttyrec3.bz2` and `ttyrec.bz2` files are a little different from each other, two different methods should be used: `nle.dataset.add_nledata_directory` for `ttyrec3.bz2`, and `nle.dataset.add_altorg_drectory` for `ttyrec.bz2` data from NAO. This is documented in the code and in the Tutorial Notebook.

G.5 ThreadPool Usage

All files are read independently, loaded into their own C++ `Converter` object that provides lightweight terminal emulation. This independence provides the potential for very effective parallelization along the batch index, B. In the `Converter` object, we release the GIL, allowing for very true parallelization of file processing with Python's own threads. This can provide a many-fold boost to performance and throughput of frames per second, as can be seen in Table 4. Note, however, that the benefits of parallelization are mainly found with long sequence lengths. Example usage is shown in Figure 5.

Figure 4: Example of how to simply create a database file and populate it with datasets. Note that each directory must be associated with a dataset name chosen by the user (*i.e.* ‘‘`nld-aa`’’). This is then used to access the data (as in Figure 5).

```
import nle.dataset as nld
path_to_nld_aa = "/path/to/nld-aa"
path_to_nld_nao = "/path/to/nld-nao"
path_to_custom = "/path/to/a/custom/nledata/directory"

# Chose a database name/path. By default, most methods with use nld.db.DB (='ttyrecs.db')
dbfilename = "ttyrecs.db"
if not nld.db.exists(dbfilename):
    nld.db.create(dbfilename)

    # To add the NLE-AA data, or any data generated from nle, use `add_nledata_directory`.
    nld.add_nledata_directory(path_to_nld_aa, "nld-aa", dbfilename)

    # to add the NLE-NAO data, use the `add_altorg_directory`.
    nld.add_altorg_directory(path_to_nld_nao, "nld-nao", dbfilename)
```

Figure 5: Example of how to use a `ThreadPoolExecutor` using Python 3’s built-in `concurrent.futures` module. This parallelism allows control of how many CPU workers are use to load data, without having to resort to Python’s `multiprocessing`.

```
from concurrent.futures import ThreadPoolExecutor
import nle.dataset as nld

with ThreadPoolExecutor(max_workers=10) as tp:
    dataset = nld.TtyrecDataset("nld-aa", threadpool=tp)

    for i, mb in enumerate(dataset):
        # Etc ...
```

G.6 SQL Usage

With such a large and diverse dataset, it is highly likely that many will wish to train on subselections of the data according to certain criteria, contained in the metadata. The `TtyrecDataset` enables this very easily, by exposing the ability to subselect *gameids* on the basis of a SQL command. An example of this is shown in Figure 6, and was used to generate the plots in Figure 6.

Figure 6: Example of how to use a SQL queries to subselect a part of a dataset dynamically. This method was used to generate the `NLD-AA-Monk` runs only runs in Table 2 and Figure 2.

```
# Generate subdataset of "nld-aa" with only Human Monk games
import nle.dataset as nld
import nle.nethack

subselect_sql = "SELECT gameid FROM games WHERE role=? AND race=?"
subselect_sql_args = ("Mon", "Hum")

monk_dataset = nld.TtyrecDataset(
    "nld-aa",
    subselect_sql=subselect_sql,
    subselect_sql_args=subselect_sql_args
)

for mb in monk_dataset:
    # Etc ...
```

G.7 Creating Custom Datasets

It is easy to create and add your own custom dataset using `NLE` and `NLD`. `NLE` includes the ability to record files to a given save directory. The directory can then be directly added to NLD using the `add_nledata_directory` methods referred to in Appendix G.4. An example is shown in Figure 7.

Table 4: The frames per second, mean (standard deviation), across 5 runs of 500k frames when served by the NLD dataset with different parameters for batch size, sequence length, and number of CPUs used (which corresponds to the number of threads in the threadpool).

			Sequence Length		
			32	128	512
1 CPU	Batch Size	32	20.6k (325)	19.4k (250)	20.7k (499)
		128	16.5k (159)	18.9k (235)	21.2k (254)
		512	15.1k (325)	17.1k (218)	17.9k (94)
10 CPUs	Batch Size	32	73.8k (3.2k)	110.8k (3.0k)	142.7k (6.7k)
		128	113.1k (1.5k)	144.2k (1.9k)	167.0k (3.3k)
		512	114.2k (2.2k)	129.8k (2.7k)	139.1k (4.1k)
80 CPUs	Batch Size	32	86.2k (2.3k)	134.6k (3.9k)	231.2k (7.2k)
		128	177.8k (2.9k)	372.9k (16.5k)	492.4k (24.4k)
		512	222.4k (7.9k)	446.7k (22.5k)	482.5k (10.4k)

Figure 7: Example of how to create a custom dataset using NLE and NLD. A variant of this method was used to create NLD-AA.

```python
import gym
import nle
import nle.dataset as nld


def generate_rollouts(env):
    obs = env.reset()
    games = 0
    while games < 10:
        obs, reward, done, info = env.step(env.action_space.sample())
        if done:
            games += 1
            env.reset()


# 1. Create some envs, chose savedir, save every X games
envA = gym.make("NetHackChallenge-v0", savedir="path/to/save/A", save_ttyrec_every=2)
envB = gym.make("NetHackScore-v0", savedir="path/to/save/B", save_ttyrec_every=1)

# 2. Generate rollouts, resetting when `done=True`
generate_rollouts(envA)
generate_rollouts(envB)

# 3. Add to database (default dbfilename="ttyrecs.db")
nld.add_nledata_directory("path/to/save", "custom")

# 4. Use dataset
dataset = nld.TtyrecDataset("custom")
for mb in dataset:
    # Etc ...
```

H NAO Details

H.1 Conducts and Achievements

By analysing the Conducts and Achievement flags in the *conduct* and *achieve* metadata fields, one can see the variety of behaviours in the NLD-NAO dataset. The aggregated % of episodes with each conduct/achievement is shown for each dataset in Table 5.

H.2 Metadata Matching Algorithm

On NAO, a game of NetHack can be saved and resumed, resulting in games split across several `ttyrec.bz2` files and time periods. In these situations, only one entry is made to the communal `xlogfile`, and this is made at the very end of the game with no link to which files are part of the game.

However, it is possible to assign files to games making use of three pieces of information: first, that alt.org enforces that a user complete one game before starting another; second, that the `xlogfile` records the *starttime* and *endtime* of the game; and, third, that each file is saved with the same filename template indicating the user and file creation time (`<username>/<file_creation_timestamp>.ttyrec.bz2`).

Table 5: A table of the conducts and achievements accomplished as a percentage of total episodes for each dataset. Most notably we can see only 1.46% of episodes ascend in NLD-NAO, and 0.04% do this without any armour (Nudist). By contrast NLD-AA achieves only the occasional solving of the Sokoban, indicating NLD-NAO has a much better coverage of the late game.

Conducts	NLD-AA (%)	NLD-NAO (%)
Foodless	0.89	21.16
Vegan	1.35	29.50
Vegetarian	1.41	32.18
Atheist	2.36	52.13
Weaponless	7.71	17.77
Pacifist	0.26	8.53
Illiterate	33.36	34.61
Polypileless	100.00	96.88
Polyselfless	87.65	93.59
Wishless	98.91	95.88
Artifact wishless	100.00	98.74
Genocideless	100.00	96.90

Achievements	NLD-AA (%)	NLD-NAO (%)
Got the Bell of Opening	0.00	2.15
Entered Gehennom	0.00	2.06
Got the Candelabrum of Invocation	0.00	1.73
Got the Book of the Dead	0.00	1.65
Performed the Invocation	0.00	1.62
Got the Amulet of Yendor	0.00	1.60
Was in the End Game	0.00	1.56
Was on the Astral Plane	0.00	1.53
Ascended	0.00	1.46
Got the Luckstone at Mines' End	0.00	3.95
Finished Sokoban	0.17	5.93
Killed Medusa	0.00	2.31
Zen conduct intact	0.00	0.05
Nudist conduct intact	0.00	0.04

Making use of these three pieces of information, we can assign the files to the games, by lining up all episodes per user in an `xlogfile` by start and endtime, and then assigning files created between that start and endtime to that game. The full logic for this is carefully documented in the file `nle/datasets/populate_db.py`.

In this way it is possible to remove games that have started but not finished from the alt.org S3 bucket, whilst also filter out games that may have started well before the first *starttime* in an `xlogfile` (indicating for whatever reason the game was not logged). All this logic is vital in ensuring that NLD episodes can be seamlessly stitched together correctly from many file parts.

H.3 Filtering 'Bad' Episodes for NLD-NAO

We filtered 'bad' episodes for NLD-NAO in three ways. First, we removed episodes from the dataset where the player seems to have participated in 'start-scumming'. This practice involves starting a game of NetHack and ending it early if the randomly-generated starting attributes/inventory is not favourable enough. We filtered these games by filtering episodes from the dataset with fewer than 10 turns, where the death was 'quit' or 'escaped'.

We also filtered an episode with negative in-game *turns* - something which should be impossible! This single player had managed to survive many orders of magnitude longer than any other game, eventually successfully overflowing the *turns* counter (a 32-bit integer). While surviving this long is impressive, it is unlikely this episode demonstrate a drive to 'ascend' and may pose a large bias to our dataset, and so is filtered out.

Finally we also decided to remove the files of players with highly offensive names from our dataset. These accounted for very few episodes and frames (less than 0.1%) and largely from poor players. To add an extra layer of privacy, we pseudonymise all player names in the database, even though player names are generally already pseudonyms.

H.4 Noise Investigation

The `NLD-NAO` dataset is a real-world dataset collected over a decade, and as such contains some real world noise that can pose research problems.

The first problem is terminal rendering noise. Different terminals have different escape sequences and different characters sets, and some of these may not be interpretable by `NLD`'s lightweight VT100 emulator. While `NLD` has made adjustments to support DEC graphics[24] and IBM graphics[25], it is still possible for a user to define their own exotic graphics which could render noisily in `NLD`, perhaps with extra, unusual symbols. This problem is also faced by alt.org's own webplayer, but thankfully, this issues is only encountered very rarely.

Another source of rendering noise, aside from graphics, is terminal size. Players can play NetHack with different terminal sizes, and NetHack's clever paging system will adapt the screen to the right size. In our case we chose by default to render on the screen size used by `NLE`, which is also the smallest "full size" you can play NetHack on (80 x 24). For episodes where the screen size was larger than this, we make the decision to crop the screen rather than wrap around the lines (as one typically would when emulating large commands on a small screen). It is possible to control the emulator size scren by using the `rows` and `cols` arguments to the constructor, if the user so wishes.

The second source of noise comes from errors introduced into the metadata assignment process, outlined in Appendix H.2. In some cases, such as under heavy traffic, NetHack can fail to write a line to an `xlogfile`, and drop the line entirely. This introduces noise into the our assignment algorithm and could result in a user's games potentially including some `ttyrec` files from a different one of their episodes. We check for discrepancies between metadata and in-game play, by rendering the last 10 frames of all episodes in `NLD-NAO` and checking whether the *point* and number of *turns* parsed from the screen match the gameplay metadata in `NLD-NAO`. Since these two values are themselves not always displayed or might change after game end, this itself is a noisy process, yet we managed to obtain a upper bound on the errors. In the episodes we found at most ~5% occurence of a discrepancy between the final frames of an episode and the metadata for *score* and *turns*, and were able to verify correct metadata for ~90% of episodes (~5% we were unable to classify).

I Experiment Hyperparameters and Details

I.1 Methods

All methods are implemented using the open-source RL library `moolib` [31]. The online and online + offline RL methods are built on top of an APPO implementation based on [37]. The offline RL methods are built on top of the DQN algorithm. For CQL, we followed the implementation from [53], which is an established library for offline RL algorithms. For IQL, we followed the original implementation open-sourced by the authors [20]. We use the same base architecture and processing of the observations and actions for all algorithms. Please check out our open-sourced code for full details of these implementations. All experiments were run with 20 CPUs in approximately 1 day on single Volta 32GB GPUs.

I.2 Preprocessing

As a preprocessing step, a wrapper is used to render the observations `tty_chars` and `tty_colors` as pixels, which are then downsampled and cropped centered on `tty_cursor`. The resulting image is added as an observation known as `screen_image`. This preprocessing step is standard for the Chaotic Dwarf baseline model that we use. [15]

[24] `https://en.wikipedia.org/wiki/DEC_Special_Graphics`
[25] `https://nethackwiki.com/wiki/IBMgraphics`

I.3 Model

We adapt the model from the standard Chaotic Dwarf Sample Factory baseline, that was open sourced[26] during the NetHack Challenge. [15] This model relies on three small encoders for observations, that combine their representations before passing through an LSTM and taking policy and baseline heads from the output.

The original model relied on NLE's `blstats`, `message`, and the preprocessed `screen_image` (outlined in Appendix I.2) observations. Since the former two do not exist in the raw terminal output, instead they are replaced with their closest proxies - the top and bottom two lines of `tty_chars` respectively. This involves slight changing to the normalization of the encoders. The full model can be found in our open sourced code.

I.4 Hyperparameter Sweeps

For DQN, we performed a hyperparameter search over $\epsilon_{start} \in [0.1, 0.2, 0.25, 0.5, 0.75, 0.8, 0.9]$, $\epsilon_{decay} \in [2500, 5000, 10000, 20000, 25000, 50000, 75000, 250000]$, $target\ update \in [40, 200, 400, 800, 2000, 4000]$ and tried both a MSE and a Huber loss. We found the following to work best: $\epsilon_{start} = 0.25$, $\epsilon_{decay} = 25000$, $target\ update = 400$, and MSE loss. For the common hyperparameters between CQL and IQL, we use the same values as the best ones found for DQN without additional tuning. For CQL, we also performed a hyperparameter search over $\tau \in [0.005, 0.05, 0.0005, 0.1, 0.01, 0.001, 0.0001]$ and for $cql\ loss\ coef \in [1.0, 0.5, 0.1, 2.0, 10.0]$. We found $\tau = 0.005$ and $cql\ loss\ coef = 2.0$ to work best. For IQL, we performed a hyperparameter search over $expectile \in [0.8, 0.7, 0.9]$ and $temperature \in [1.0, 0.1, 10.0, 0.5, 2.0]$. We found $expectile = 0.8$ and $temperature = 1.0$ to work best. Table 6 contains a list with all relevant hyperparameters used in our experiments (*i.e.* the best ones found following our HP search).

I.5 Evaluation

We evaluate performance using the `eval.py` script in the experiment code. This script creates a `moolib.EnvPool` and assigns each batch index a number of episodes to run, to collect data. We collect 1024 episodes in this way, per run.

It is important to note that this method of evaluating performance is slightly different to the one generating curves in Figure 2. For training curves, we add any episodes to a running mean as soon as they finish rolling out (*i.e.* just before training). The end consequence of this is that short episodes with low scores are sampled disproportionately more often that very long episodes with high scores - in other words these curves are biased *down*. Accurately sampling these very high scoring and very long rollouts account for the discrepancy between the curves in Figure 2 and the values in Table 2.

J On the Scale and Performance of `NLD`

We compare the scale and performance of `NLE` and `NLD` to two other large scale datasets of human demonstrations: MineRL [14] and StarCraft II [59]. In particular, we present the size of the datasets, in terms of trajectories, and benchmark the performance of the datasets and environments, measured in frames per seconds (FPS). The overview can be found in Table 7.

J.1 Overview

These results support the claim of `NLD` to be a large-scale dataset. `NLD-NAO` has more trajectories than dataset used in AlphaStar[58], and possibly also more transitions, since the average episode of `NLE` is longer than that of StarCraft II. Even `NLD-AA` has an order of magnitude more trajectories than MineRL.

These results also support the claims of `NLD` and `NLE` to be very performant in terms of frames per second. The `NLE` environment is at least an order of magnitude faster than either alternative environment, and `NLD` is potentially several orders of magnitude faster at loading data. Further details in how the batch size and unroll length affect the performance of `NLD` can be found in Table 4.

[26] https://github.com/Miffyli/nle-sample-factory-baseline

Table 6: List of hyperparameters used to obtain the results in this paper.

Hyperparameter	Value
activation function	relu
actor batch size	512
adam betea1	0.9
adam beta2	0.999
adam eps	0.0000001
adam learning rate	0.0001
appo clip policy	0.1
appo clip baseline	1.0
baseline cost	1
batch size	256
crop dim	18
grad norm clipping	4
normalize advantages	True
normalize reward	False
num actor batches	2
num actor cpus	20
pixel size	6
rms alpha	0.99
rms epsilon	0.000001
rms momentum	0
reward clip	10
reward scale	1
unroll length	32
use batchnorm	false
use lstm	true
virtual batch size	128
rms reward norm	true
initialisation	orthogonal
use global advantage norm	norm
msg embedding dim	32
msg hidden dim	512
kickstarting loss	1.0
ttyrec envpool size	4
ttyrec batch size	256
ttyrec unroll length	32
ttyrec envpool size	4
ϵ_{start}	0.25
ϵ_{end}	0.05
ϵ_{decay}	25000
target update	400
dqn loss	mse
τ	0.005
cql loss coef	2.0
expectile	0.8
temperature	1.0

J.2 FPS Benchmarking Method

To compute the throughput of `NLD` we used the threadpool with 80 threads on a machine with 80 CPUs, a batch size of 128 and a sequence length of 32, taking the mean and standard deviation from

Table 7: The mean frames per second (± standard deviation) across 5 runs for each environment or dataset. Although batch sizes and sequence length can affect performance, it is clear that NLD is considerably faster than rival datasets.

Dataset	NLD	StarCraft II	MineRL
Trajectories	**100,000 - 1,500,000**	971,000	<5000
Environment FPS	**26400 ± 1200**	200-700	68 ± 6.2
Dataset FPS	**113100 ± 1500**	1800 ± 2.9	5000 ± 89

5 runs to 500k frames. All assessments are done with NLD-AA, the slower of the two datasets. For NLE we ran the environment for 500k frames, with an agent sampling random actions and a max episode length of 1000 frames.

For StarCraft II we include the throughput of the environment as reported in the original paper, and measure the throughput of the dataset by running the pysc2.bin.replay_actions function, provided in the github repo for batch processing replays, for 5 minutes with 80 threads on 80 cpus. This too was repeated 5 times.

For MineRL we used the MineRLTreechop-v0 environment and dataset. We measured the environment speed by running headless on a machine with 80 cpus for 10k steps, sampling random actions and assuming a max episode length of 30k steps (note, the MineRL documentation suggests that running headless slows down the environment by a factor of 2-3x because the rendering is performed on the CPU instead of the GPU). We measured the MineRL dataset throughput using the provided BufferedBatchIter with a batch size of 4096 for a full epoch. Once again, we repeated this 5 times.

Table 7 presents the mean and standard deviation of the frames per second.

K Limitations of our Work

One potential limitation of our work is our human dataset NLD-NAO doesn't contain any actions or rewards, so it cannot directly be used for imitation learning of offline RL. However, this is the case for many other domains of interests where you may have access to sequences of observations (*e.g.* when learning skills from human or agent videos in domains like robotics or autonomous driving). Thus, we believe our human dataset can enable research on this more realistic and important setting in a safe environment.

While NLD-AA contains both actions and rewards thus allowing research on imitation learning and offline RL, it is less diverse than NLD-NAO as it contains demonstrations from only one symbolic bot, the winner of the NetHack Challenge at NeurIPS 2022 [15]. In the future, we hope the community will contribute many more datasets to NLD collected using a variety of bots (including learning-based ones). We believe that the current dataset is a good starting point to develop even better agents at playing NetHack. Then, those agents could be used to collect additional trajectories which can be added to NLD, resulting in a feedback loop that constantly grows the size and diversity of the dataset and develops better and better agents.

Another potential limitation of our work is that we use the in-game score to train our RL agents, which is the standard practice from [23] and [15]. While the in-game score roughly correlates with good performance on NetHack, it is not exactly the same as ascending *i.e.* winning the game, which is ultimately how we want to evaluate agents. For example, humans sometimes attempt to ascend with as low a score as possible, as an additional challenge. This misalignment is common in other real-world domains where it can be difficult to design a good reward function that illicits the desired behavior (*e.g.* autonomous driving). Thus, this is an open research challenge which could be enabled by our dataset, but lies outside the scope of the current work.

L Broader Impact Statement

Our work introduces a new large-scale dataset based on a computer game to enable research in multiple domains including learning from demonstrations, imitation learning, and reinforcement learning. Our dataset is cheap to run in order to democratize research on large-scale datasets of demonstrations, which have historically been restricted to well-resourced (industry) labs. This dataset contains human demonstrations, but these are already available online and the demonstrations are de-identified such that only a user's chosen usernames can be retrieved. Thus, we don't envision any direct negative social impact of this work. Of course, people could use our dataset to develop new methods for learning from demonstrations. Since these techniques can be quite general, they could also be used for other real-world applications such as autonomous driving, robotics, healthcare, financial services, or recommendation systems (where large-scale datasets may be available). Of course, these domains are more sensitive than computer games, so much more care is needed to mitigate the risks and develop safe, robust systems. While we believe this lies outside the scope of current work, researchers and practitioners using our dataset should always keep an eye on potential negative societal impacts.