Deep equilibrium Net

v1.0 - Simon Scheidegger, 12/2021 v1.1 - Simon Scheidegger, 01/2022: Added eligibility trace as an option

Dependencies

The basic dependencies are tensorflow==2.3, hydra-core >= 1.1 and tensorboard (for monitoring). For a full set of dependencies see the environment.yaml - this includes the development environment as well (Spyder 3).

Horovod

The code can be run on multiple nodes / gpus by making use of Horovod - in this case you need to configure the Horovod environment depending on the underlying system.

The code will automatically import and use Horovod if it is being run using MPI via `horovodrun`.

Running the code

To run the code, just execute:

```
python run_deepnet.py hydra.run.dir=<WORKING_DIRECTORY_TO_USE>
```

A default `hydra.run.dir` can be specified in config/config.yaml and hydra can also use default paths + automatic timestamp additions as well. For more information see the first example: https://hydra.cc/docs/configure_hydra/workdir

The `hydra.run.dir` will contain checkpoints, tensorboard outputs and also a .hydra folder that contains the **merged** run configuration yaml file.

It is important to specify the right `hydra.run.dir` if you want to restart a computation, etc..

Configuration

Configuration is done via hydra (see. hydra.cc). For example to run with a different Neural-Net config

```
python run_deepnet.py net=deep
```

Individual values can also be overwritten inside the config yaml files. The base config file is contained under config/config.yaml and this specifies by default how other configuration files are picked up.

NOTE: It is possible to use a previously merged configuration file instead of putting it together from individual configuration files by placing the config.yaml file inside the `hydra.run.dir/.hydra` folder and setting the

`USE_CONFIG_FROM_RUN_DIR` environment variable before starting the run, see the Post-Processing section for an example.

IMPORTANT: Note that activation functions on the output layer can be set policy-by-policy. To do so, please go to /config/variables/*.yaml, and add the relevant function that transforms your (probably linear) output, e.g.:

```
- name: p_x
  bounds:
    lower: 0.1
    upper: 1.6
  activation: tf.nn.relu
```

Note, you can apply here any function - the script is not constrained to tensorflow.

NOTE: If you want to sample exogenous variables from a particular distribution, you can do so by adding in the "variables.yaml" file for instance for in case of a normal distribution:

```
- name: Ax
  init:
    distribution: normal
    kwargs:
      mean: 1.0
      stddev: 0.1
```

or e.g. for a uniform distribution

```
- name: Gx
  init:
    distribution: uniform
    kwargs:
      minval: 0.127
      maxval: 0.177
```

Running a specific model

Model files can be put into directories (e.g. analytic). Model files currently contain the following:

- Definitions (derived quantities)
- Dynamics (state transitions)
- Equations (equilibrium conditions)
- Hooks (things to be done after each episode - e.g. plotting)

For example, to run the analytic model (from the start):

```
python run_deepnet.py MODEL_NAME=analytic run=analytic net=analytic
optimizer=analytic constants=analytic variables=analytic
```

How the model is simulated can be found inside config/run/*.yaml. The latter file needs to contain the following variables:

```
N_sim_batch: 1000
N_episode_length: 30
N_epochs_per_episode: 1
N_minibatch_size: 500
N_episodes: 500000
sorted_within_batch: false
```

Important to note: N_sim_batch is the number of trajectories simulated in parallel, so e.g. 1000 episodes are simulated in parallel. Each (parallel) episode is 30 long in this case, so altogether 30000 states are drawn at each episode iteration. These are then batched into N_minibatch_size for gradient steps (so in this case 60 minibatches are drawn of size 500, for each episode).

If `sorted_within_batch` is set, then the batches will try to be contiguous trajectory elements. This should in general entail that N_episode_length is a multiple of N_minibatch_size and each minibatch then is a contiguous fragment of one trajectory.

Calibrated model example:

```
python run_deepnet.py constants=sudden_stop_calibrated
variables=sudden_stop_calibrated MODEL_NAME=sudden_stop_calibrated
```

Dropout, Initialization of Network weights, and choice of optimizers

- There is a possibility to add a 'dropout_rate' to a layer configuration. In this case a Dropout layer is added before that layer.

The Dropout layer is only active during the 'epoch' run (calculating losses & doing a gradient step based on that) and is not active during the episode generation (i.e. simulation) and when running Hooks (e.g. plotting policies). See, e.g., in config/net/analytic.yaml

```
layers:
  - hidden:
     units: 100
     type: dense
     activation: relu
     init_scale: 0.1
     dropout_rate: 0.05
  - hidden:
     units: 50
```

```
      type: dense
      activation: relu
      init_scale: 0.1
      dropout_rate: 0.05
  - output:
      type: dense
      activation: linear
      init_scale: 0.1
net_initializer_mode: fan_avg
net_initializer_distribution: truncated_normal
```

- There exists the possibility to choose the optimizer - just specify it in the optimizer name (as listed in https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) in config/optimizer/yourconfigfile.yaml.

```
optimizer: Adam
learning_rate: 0.00001
clipvalue: 1.0
```

- There is also the possibility to choose the distribution (uniform, normal, truncated_normal) and mode of the neural network initializer (eg. fan_in, fan_out or fan_avg). Together with the scale parameter, this should cover https://www.tensorflow.org/api_docs/python/tf/keras/initializers/GlorotUniform as well. See the exact details at: https://www.tensorflow.org/api_docs/python/tf/keras/initializers/VarianceScaling.

Batch-normalization can be done before a given layer, by adding a `batch_normalize` entry to the layer, like:

```
layers:
  - hidden:
      units: 100
      type: dense
      activation: relu
      init_scale: 0.1
      batch_normalize:
        momentum: 0.99
```

The parameters of this field are passed on directly to https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization

Adding a new model

To add a new model, the following needs to be done:

0. In the `config/variables` folder create a yaml file for the model, which will contain the name (and bounds) of the available:
    - states
    - policies

    - definitions

1. Create a folder for the model
2. Add a `Dynamics.py` file. This should define the following (for an example see the analytic or sudden_stop models):
    - a variable called `shock_values`: This should be a tensor where rows are possible outcomes used for quadrature approximation
    - a variable called and `shock_probs`: this should contain the associated probabilities
    - A function called `total_step_spec_shock`: this should take two inputs: previous state and previous policy value. It applies the specific shock index and returns the state assuming the given shock has happened. Used for conditional expectation.
    - A function called `total_step_random`: this should take two inputs: previous state and previous policy value. It applies random shocks and gives back future state. Used for simulation.
3. Add an `Equations.py` file. This should contain a single function called `equations` that takes current state and policy and returns deviation from the equilibrium conditions as a dictionary of deviations. Equations should import the 'top level' State, PolicyState and Definitions modules as in this case the bounds will automatically be enforced to make sure that the equations make sense.

If the equations involve conditional expectations, then you can import the `E_t_gen` function from `State.py`. This function accepts the current state and policy and returns a function that calculates conditional expectations. It is advised to reuse this function to avoid having to re-calculate the policy for the future possible states for each conditional expectation.

4. Add a `Definitions.py` - this should contain defined quantities as a functions of state and policy. These should also be registered in the variables configuration file, where it is possible to define bounds.
5. Add a `Hooks.py` - this should contain a single function called `cycle_hook` - this takes two inputs: current state and the episode count. It should contain logic to create plots, etc...

Expectation types

It is possible to use 3 types of expectations:

- product
- monomial
- pseudo_random

This can be configured by providing the `expectation_type` (e.g. in run/xyz.yaml).

To use the monomial setup, the Dynamics.py needs to generate the shock_values and shock_probs using State.monomial_rule in case Parameters.expectation_type is monomial. See an example in `sudden_stop/Dynamics.py`.

The pseudo_random approach can be applied without modification. In the case also the sample number should be given:

```
expectation_type: pseudo_random
expectation_pseudo_draws: 5
```

Bounds

If in `config/variables/xyz.yaml` a bound is given, then the loss function will automatically add a penalty term push the model to respect the bound. Bounds on derived quantities can be added by creating a new definition in the model specific Definitions.py and adding a bound in `config/variables/xyz.yaml`.

IMPORTANT: if a definition doesn't directly depend on the policy, then the bound will be enforced, but the penalty term will not push the optimizer to respect the bound. This is because the gradient step is taken on already simulated data, so the state part is not dependent on the neural-network parameters.

State bounds are also supported (enforced) - however these don't depend directly on the policy, so it doesn't make sense to add a default simple penalty term. What can be done under this setting is to set up additional equilibrium equations using E_t_gen - where evalFun is set to (State.xyz - State.xyz_RAW)**2. This will force the model to choose policy functions where the 'RAW' (unbounded) version in the next step is equal to the bounded version.

Customized penalty functions

There is a custom field in the variables where you can add custom penalties which override the default (1/bound**2). So you can set an explicit 0 bound, just make sure to specify a custom penalty for the bound as follows:

```
policies:
  - name: nu
    bounds:
      upper: 5.0
      lower: 0.0
      penalty_lower: 100000000.0
```

There is always a default squared penalty (bound_exceedence ** 2). If you would like to add a custom penalty, you can always add it as a new equation inside the model referencing the given value you would like to have (the square-root if you are on quadratic equation losses) - and you can set the corresponding penalty in the yaml file to zero to remove the quadratic penalty.

Example: if you want an exponential-type lower bound of z on policy xyz:

loss_dict['eq_N'] = exp(tf.math.relu(z - PolicyState.xyz)))

This will be zero if the Policy is above the bound and quickly increase if it's below.

Note, that you need to make sure the bounds in the yaml and the equations are consistent (or you can reference the bound dictionaries from the Parameters module explicitly to make it less error prone) .

One small addition: The bounds in the yaml will be always enforced, i.e. PolicyState.xyz will never return values outside the bound. So you will need to use PolicyState.xyz_RAW in the custom loss function which is the 'uncut' version from the neural net.

The best way to define the custom penalty is to contrast the xyz_RAW with xyz - this is actually what we do for the default bounds. If the bound is not active the difference is zero between the two, otherwise it's not

Implied bounds

If both upper and lower bounds are specified, then it's possible to use `activation: implied` to use a sigmoid type activation which will make the policy respect automatically the bounds.

Restarting from a checkpoint

```
python run_deepnet.py STARTING_POINT=LATEST
```

LATEST can be replaced by the name of a checkpoint as well. By default checkpoints are taken every episode (this can be changed via the CHECKPOINT_INTERVAL) config parameter. Make sure to use the `hydra.run.dir` option set to the directory where the previous run was in.

NOTE: if you want to restart the run with the exact same configuration as it was before, then use the `export USE_CONFIG_FROM_RUN_DIR` as in the post_processing example to pick up the merged `config.yaml` from the run.

NOTE-2: if you don't use the `USE_CONFIG_FROM_RUN_DIR=...` setup and restart with a different configuration (e.g. changed something in the config/ folder), then this will overwrite the `<hydra.run.dir>/.hydra/config.yaml` file.

Monitoring

Monitoring can be done via Tensorboard, pointing it to the hydra.run.dir directory. Diagnostic information (e.g. loaded config values, current iteration, etc...) is printed also to stdout.

Post-processing

Post-processing can be done by defining an environment variable called `USE_CONFIG_FROM_RUN_DIR` - in this case hydra will pick up the output config.yaml inside the .hydra folder.

Then just import Parameters in a script and call it with the given checkpoint name. For example:

```
export USE_CONFIG_FROM_RUN_DIR=runs/bm1972/2024-09-03/13-13-03 && python
post_process.py STARTING_POINT=LATEST hydra.run.dir=$USE_CONFIG_FROM_RUN_DIR
```

export USE_CONFIG_FROM_RUN_DIR=runs/bm1972/2024-09-03/13-13-03 && python post_process_bm1972.py STARTING_POINT=LATEST hydra.run.dir=$USE_CONFIG_FROM_RUN_DIR

Using this, the exact same settings will be used to re-initialize the model as was used in the original run and then any kind of post-processing can be done (e.g. simulate more, plot, etc..).

It's advisable to save a copy of the original outputs in case some files are modified in the USE_CONFIG_FROM_RUN_DIR.

To customize the post_process.py even further, there is a global modules called Globals, where you can set Global configuration. It contains variables such as POST_PROCESSING or DISABLE_SHOCKS which can be used inside Dynamics.py to condition behaviour. For example in post_process.py we set Globals.POST_PROCESSING=True.

Custom behaviour can be then configured in any part of the code by declaring

import Globals and using the value of Globals.POST_PROCESSING for conditional actions.

If there is e.g. two types of dynamics for the states to be used, the way to go would be to have a single Dynamics.py and inside it have something like

Dynamics.py

import Globals

def policy_step(prev_state, policy_state): if not Globals.POST_PROCESSING then: do regular dynamics else: # sample from ranges instead of doing a proper dynamics update policy_step = tf.zeros_like(prev_state) policy_step = State.update(policy_step, "Kx",State.Kx(prev_state)) policy_step = State.update(policy_step, "Cx",State.Cx(prev_state)) policy_step = State.update(policy_step, "Ix",State.Ix(prev_state))

Post-processing configuration

The default simulation length / width (which depends on `initialize_each_episode`) can be overridden by adding new flags when running post-process:

```
python post_process.py  ... +N_simulated_batch_size=100
+N_simulated_episode_length=100
```

Multi-node runs via Horovod

Running multi-node configuration is possible via Horovod. If running via `horovodrun`, the code will load Horovod and apply the following:

- Each process will initialize it's own seed depending on its rank (thus random starting state)
- Scale the learning rate by the number of processes
- On the first iteration the master process will broadcast the neural net parameters
- Subsequent gradient steps will be averaged across worker and the net parameters will be kept in sync

The master process is tasked with creating checkpoints and writing out tensorboard data.

Since the worker processes don't write out their local state values, restarting a Horovod run in a way that it would continue on the exact same trajectory as without a restart is currently not possible (when restarting from a checkpoint, all workers will load the state values that the master checkpointed). However the worker seeds will be different, so after a few episodes the different workers will again work on a 'different part of the state space'.

How to run tensorboard

You need to go to the runs folder that contains that path to the solution. You go to the folder with the time stamp with the model and press right button 'Copy path'. The path should look like this: DEQN_workshop/Day_2/DEQN_library/runs/bm1972/2024-09-03/13-13-03

Then in the terminal window you execute the following commands:

- rm tensorboard_logdir This command helps you to remove any previous links that were stored. If there was no links you get an error message and it is fine.
- ln -s DEQN_workshop/Day_2/DEQN_library/runs/bm1972/2024-09-03/13-13-03 tensorborad_logdir This command gets you started with the tensorboard

Then you go to the Launcher and press tensorboard icon (top left). It should open in the new browser tab.