
jax_verify

DeepMind

Nov 05, 2020

CONTENTS

1	API Reference	1
1.1	Verification methods	1
1.2	Bound objects	3
1.3	Utility methods	3
2	SDP Verification	5
2.1	API Reference	5
3	Installation	9
4	Support	11
5	License	13
	Index	15

API REFERENCE

1.1 Verification methods

`jax_verify.crown_bound_propagation` (*function*, **bounds*)

Performs CROWN as described in <https://arxiv.org/abs/1811.00866>.

Parameters

- **function** – Function performing computation to obtain bounds for. Takes as only argument the network inputs.
- ***bounds** – `jax_verify.IntervalBound`, bounds on the inputs of the function.

Returns Bounds on the output of the function obtained by FastLin

Return type `output_bound`

`jax_verify.crownibp_bound_propagation` (*function*, *bounds*)

Performs Crown-IBP as described in <https://arxiv.org/abs/1906.06316>.

We first perform IBP to obtain intermediate bounds and then propagate linear bounds backwards.

Parameters

- **function** – Function performing computation to obtain bounds for. Takes as only argument the network inputs.
- **bounds** – `jax_verify.IntervalBounds`, bounds on the inputs of the function.

Returns Bounds on the output of the function obtained by Crown-IBP

Return type `output_bound`

`jax_verify.fastlin_bound_propagation` (*function*, **bounds*)

Performs FastLin as described in <https://arxiv.org/abs/1804.09699>.

Parameters

- **function** – Function performing computation to obtain bounds for. Takes as only argument the network inputs.
- ***bounds** – `jax_verify.IntervalBound`, bounds on the inputs of the function.

Returns Bounds on the output of the function obtained by FastLin

Return type `output_bound`

`jax_verify.ibpfastlin_bound_propagation` (*function*, **bounds*)

Obtains the best of IBP and Fastlin bounds.

Parameters

- **function** – Function performing computation to obtain bounds for. Takes as only argument the network inputs.
- ***bounds** – jax_verify.IntervalBound, bounds on the inputs of the function.

Returns Bounds on the output of the function obtained by FastLin

Return type output_bound

`jax_verify.interval_bound_propagation` (*function*, **bounds*)

Performs IBP as described in <https://arxiv.org/abs/1810.12715>.

Parameters

- **function** – Function performing computation to obtain bounds for. Takes as only argument the network inputs.
- ***bounds** – jax_verify.IntervalBounds, bounds on the inputs of the function.

Returns Bounds on the output of the function obtained by IBP

Return type output_bound

`jax_verify.solve_planet_relaxation` (*logits_fn*, *initial_bounds*, *boundprop_transform*,
objective, *objective_bias*, *index*, *solver=<class*
'jax_verify.src.cvxpy_relaxation_solver.CvxpySolver'>)

Solves the “Planet” (Ehlers 17) or “triangle” relaxation.

The general approach is to use jax_verify to generate constraints, which can then be passed to generic solvers. Note that using CVXPY will incur a large overhead when defining the LP, because we define all constraints element-wise, to avoid representing convolutional layers as a single matrix multiplication, which would be inefficient. In CVXPY, defining large numbers of constraints is slow.

Parameters

- **logits_fn** – Mapping from inputs (batch_size x input_size) -> (batch_size, num_classes)
- **initial_bounds** – *IntervalBound* with initial bounds on inputs, with lower and upper bounds of dimension (batch_size x input_size).
- **boundprop_transform** – *bound_propagation.BoundsTransform* instance, such as *jax_verify.ibp_transform*. Used to pre-compute interval bounds for intermediate activations used in defining the Planet relaxation.
- **objective** – Objective to optimize, given as an array of coefficients to be applied to the output of logits_fn defining the objective to minimize
- **objective_bias** – Bias to add to objective
- **index** – Index in the batch for which to solve the relaxation
- **solver** – A *relaxation.RelaxationSolver*, which specifies the backend to solve the resulting LP.

Returns The optimal value from the relaxation status: The status of the relaxation solver

Return type val

1.2 Bound objects

```
class jax_verify.LinearBound(lower_bound:      jax_verify.src.fastlin.LinearExpression,    up-  
                             per_bound:      jax_verify.src.fastlin.LinearExpression,    reference:  
                             Optional[LinearBound])
```

Represent a pair of linear functions that encompass feasible activations.

We store the linear functions as `LinearExpressions` objects in `lower_lin` and `upper_lin`, and also maintain a reference to the initial bounds on the input to be able to concretize the bounds when needed.

```
class jax_verify.IntervalBound(lower_bound:      jax._src.numpy.lax_numpy.ndarray,    up-  
                              per_bound: jax._src.numpy.lax_numpy.ndarray)
```

Represent an interval where some activations might be valid.

1.3 Utility methods

```
jax_verify.open_file(name, *open_args, **open_kwargs)
```

Load file, downloading to `/tmp/jax_verify` first if necessary.

SDP VERIFICATION

The `sdp_verify` directory contains a largely self-contained implementation of the SDP-FO (first-order SDP verification) algorithm described in Dathathri et al 2020. We *encourage* projects building off this code to fork this directory, though contributions are also welcome!

The core solver is contained in `sdp_verify.py`. The main function is `dual_fun(verif_instance, dual_vars)`, which defines the dual upper bound from Equation (5). For any feasible `dual_vars` this provides a valid bound. It is written amenable to autodiff, such that `jax.grad` with respect to `dual_vars` yields a valid subgradient.

We also provide `solve_sdp_dual_simple(verif_instance)`, which implements the optimization loop (SDP-FO). This initializes the dual variables using our proposed scheme, and performs projected subgradient steps.

Both methods accept a `SdpDualVerifInstance` which specifies (1) the Lagrangian, (2) interval bounds on the primal variables, and (3) dual variable shapes.

As described in the paper, the solver can easily be applied to other input/output specifications or network architectures for any QCQP. This involves defining the corresponding QCQP Lagrangian and creating a `SdpDualVerifInstance`. In `examples/run_sdp_verify.py` we include an example for certifying adversarial L_∞ robustness of a ReLU convolutional network image classifier.

2.1 API Reference

`jax_verify.sdp_verify.dual_fun(verif_instance, dual_vars, key=None, n_iter=30, scl=-1, exact=False, dynamic_unroll=True, include_info=False)`

Returns the dual objective value.

Parameters

- **verif_instance** – a `utils.SdpDualVerifInstance`, the verification problem
- **dual_vars** – A list of dual variables at each layer
- **key** – `PRNGKey` passed to Lanczos
- **n_iter** – Number of Lanczos iterations to use
- **scl** – Inverse temperature in softmax over eigenvalues to smooth optimization problem (if negative treat as hardmax)
- **exact** – Whether to use exact eigendecomposition instead of Lanczos
- **dynamic_unroll** – bool. Whether to use `jax.fori_loop` for Lanczos for faster JIT compilation. Default is False.
- **include_info** – if True, also return an *info* dict of various other values computed for the objective

Returns Either a single float, the dual upper bound, or if `include_info=True`, returns a pair, the dual bound and a dict containing debugging info

```
jax_verify.sdp_verify.solve_sdp_dual(verif_instance, key=None, opt=None,
                                     num_steps=10000, verbose=False, eval_every=1000,
                                     use_exact_eig_eval=True, use_exact_eig_train=False,
                                     n_iter_lanczos=30, scl=-1.0, lr_init=0.001,
                                     steps_per_anneal=100, anneal_factor=1.0,
                                     num_anneals=3, opt_name='adam',
                                     gd_momentum=0.9, add_diagnostic_stats=False,
                                     opt_multiplier_fn=None, init_dual_vars=None,
                                     init_opt_state=None, opt_dual_vars=None,
                                     kappa_reg_weight=None, kappa_zero_after=None,
                                     device_type=None, save_best_k=1)
```

Compute verified lower bound via dual of SDP relaxation.

NOTE: This method exposes many hyperparameter options, and the method signature is subject to change. We instead suggest using `solve_sdp_dual_simple` instead if you need a stable interface.

```
jax_verify.sdp_verify.solve_sdp_dual_simple(verif_instance, key=None, opt=None,
                                             num_steps=10000, eval_every=1000,
                                             verbose=False, use_exact_eig_eval=True,
                                             use_exact_eig_train=False,
                                             n_iter_lanczos=100,
                                             kappa_reg_weight=None,
                                             kappa_zero_after=None,
                                             device_type=None)
```

Compute verified lower bound via dual of SDP relaxation.

Parameters

- **verif_instance** – a `utils.SdpDualVerifInstance`
- **key** – `jax.random.PRNGKey`, used for Lanczos
- **opt** – an `optax.GradientTransformation` instance, the optimizer. If `None`, defaults to Adam with learning rate `1e-3`.
- **num_steps** – int, the number of outer loop optimization steps
- **eval_every** – int, frequency of running evaluation step
- **verbose** – bool, enables verbose logging
- **use_exact_eig_eval** – bool, whether to use exact eigendecomposition instead of Lanczos when computing evaluation loss
- **use_exact_eig_train** – bool, whether to use exact eigendecomposition instead of Lanczos during training
- **n_iter_lanczos** – int, number of Lanczos iterations
- **kappa_reg_weight** – float, adds a penalty of `sum(abs(kappa_{1:N}))` to loss, which regularizes `kappa_{1:N}` towards zero. Default `None` is disabled.
- **kappa_zero_after** – int, clamps `kappa_{1:N}` to zero after `kappa_zero_after` steps. Default `None` is disabled.
- **device_type** – string, used to clamp to a particular hardware device. Default `None` uses JAX default device placement

Returns A pair. The first element is a float, the final dual loss, which forms a valid upper bound on the objective specified by `verif_instance`. The second element is a dict containing various debug info.

class `jax_verify.sdp_verify.SdpDualVerifInstance` (*bounds*, *make_inner_lagrangian*,
dual_shapes, *dual_types*)

A namedtuple specifying a verification instance for the dual SDP solver.

Fields:

- `bounds`: A list of bounds on post-activations at each layer
- `make_inner_lagrangian`: A function which takes `dual_vars` as input, and returns another function, the inner lagrangian, which evaluates $\text{Lagrangian}(x, \text{dual_vars})$ for any value x (the set of activations).
- `dual_types`: A pytree matching `dual_vars` specifying which `dual_vars` should be non-negative.
- `dual_shapes`: A pytree matching `dual_vars` specifying shape of each var.

`jax_verify` is a library for verification of neural network specifications.

INSTALLATION

Install `jax_verify` by running:

```
$ pip install jax_verify
```


SUPPORT

If you are having issues, please let us know by filing an issue on our [issue tracker](#).

LICENSE

jax_verify is licensed under the Apache 2.0 License.

INDEX

C

`crown_bound_propagation()` (in module `jax_verify`), 1
`crownibp_bound_propagation()` (in module `jax_verify`), 1

D

`dual_fun()` (in module `jax_verify.sdp_verify`), 5

F

`fastlin_bound_propagation()` (in module `jax_verify`), 1

I

`ibpfastlin_bound_propagation()` (in module `jax_verify`), 1
`interval_bound_propagation()` (in module `jax_verify`), 2
`IntervalBound` (class in `jax_verify`), 3

L

`LinearBound` (class in `jax_verify`), 3

O

`open_file()` (in module `jax_verify`), 3

S

`SdpDualVerifInstance` (class in `jax_verify.sdp_verify`), 7
`solve_planet_relaxation()` (in module `jax_verify`), 2
`solve_sdp_dual()` (in module `jax_verify.sdp_verify`), 6
`solve_sdp_dual_simple()` (in module `jax_verify.sdp_verify`), 6