renn

Release 0.1.0

Jan 15, 2021

Overview

Hello! You are at the main documentation for the *renn* python package.

CHAPTER 1

What is RENN?

renn is a collection of python utilities for reverse engineering neural networks. The goal of the package is to be a shared repository of code, notebooks, and ideas for how to crack open the black box of neural networks to understand what they are doing and how they work. Our focus is on research applications.

Currently, the package focuses on understanding recurrent neural networks (RNNs). We provide code to build and train common RNN architectures, as well as code for understanding the dynamics of trained RNNs through dynamical systems analyses. The core tools for this involve finding and analyzing approximate fixed points of the dynamics of a trained RNN.

All of *renn* uses the JAX machine learning library for building neural networks and for automatic differentiation. We assume some basic familiarity with JAX in the documentation.

CHAPTER 2

What can I use this for?

Currently, the best use of *renn* is to train RNNs and then analyze the dynamics of those RNNs by studying numerical fixed points.

The best examples of this are in the following research papers:

- Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks, Sussillo and Barak, Neural Computation, 2013.
- Reverse engineering recurrent networks for sentiment classification reveals line attractor dynamics, Maheswaranathan*, Williams* et al, NeurIPS 2019.
- Universality and individuality in neural dynamics across large populations of recurrent networks, Maheswaranathan*, Williams* et al, NeurIPS 2019.
- How recurrent networks implement contextual processing in sentiment analysis, Maheswaranathan* and Sussillo*, ICML 2020.
- The geometry of integration in text classification RNNs, Aitken*, Ramasesh* et al, arXiv 2020.
- Reverse engineering learned optimizers reveals known and novel mechanisms, Maheswaranathan et al, arXiv 2020.

CHAPTER 3

Quickstart

This notebook walks through some of the basic functionality provided by the `renn` package.

[1]:
```
# Imports
from functools import partial

import jax
import jax.numpy as jnp

import renn

base_key = jax.random.PRNGKey(0)
```

```
/Users/nirum/anaconda3/lib/python3.8/site-packages/jax/lib/xla_bridge.py:130:␣
↪UserWarning: No GPU/TPU found, falling back to CPU.
  warnings.warn('No GPU/TPU found, falling back to CPU.')
```

3.1 Build and train RNNs

First, we will use the provided RNN cell classes to build different RNN architectures.

[2]:
```
# Here, we build an RNN composed of a single GRU cell.
cell = renn.GRU(32)
print(f'Made a GRU cell with {cell.num_units} units.')
```

```
Made a GRU cell with 32 units.
```

We can initialize the hidden state for this cell as follows:

[3]:
```
key, base_key = jax.random.split(base_key)
current_state = cell.init_initial_state(key)
print(f'Initialized state with shape: {current_state.shape}')
```

```
Initialized state with shape: (32,)
```

We can initialize the cell's trainable parameters using cell.init:

[4]:
```
num_timesteps = 100
input_dim = 2
input_shape = (num_timesteps, input_dim)

key, base_key = jax.random.split(base_key)
output_shape, params = cell.init(key, input_shape)

print(f'Outputs have shape: {output_shape}')
```

```
Outputs have shape: (100, 32)
```

The GRU cell is a subclass of RNNCell. All RNNCells have an apply method that computes a single RNN step.

[5]:
```
key, base_key = jax.random.split(base_key)
inputs = jax.random.normal(key, (input_dim,))

next_state = cell.apply(params, inputs, current_state)
print(f'Next state has shape: {next_state.shape}')
```

```
Next state has shape: (32,)
```

To apply the RNN across an entire batch of sequences, we use the renn.unroll_rnn function:

[6]:
```
batch_size = 8
key, base_key = jax.random.split(base_key)
batched_inputs = jax.random.normal(key, (batch_size,) + input_shape)
batch_initial_states = cell.get_initial_state(params, batch_size=batch_size)

states = renn.unroll_rnn(batch_initial_states, batched_inputs, partial(cell.batch_
↪apply, params))

print(f'Applied RNN to a batch of sequences, got back states with shape: {states.
↪shape}')
```

```
Applied RNN to a batch of sequences, got back states with shape: (8, 100, 32)
```

We can use these to train RNNs on different kinds of sequential data.

3.2 Analyzing RNNs

The RNN cells we have in renn are easily amenable for analysis. One useful tool is to *linearize* the RNN, meaning we compute a first-order (linear) Taylor approximation of the *nonlinear* RNN update.

Mathematically, we can approximate the RNN at a particular expansion point (h, x) as follows:

$$F(h + \Delta h, x + \Delta x) \approx h + \frac{\partial F}{\partial h}(\Delta h) + \frac{\partial F}{\partial x}(\Delta x)$$

In the above equation, the term $\frac{\partial F}{\partial h}$ is the *recurrent Jacobian* of the RNN, and the term $\frac{\partial F}{\partial x}$ is the *input Jacobian*.

We can easily compute Jacobians of our GRU cell at a particular point. We can do this using the `rec_jac` and `inp_jac` methods on the cell class:

[7]:
```
Jacobian = cell.rec_jac(params, inputs, current_state)
print(f'Recurrent Jacobian has shape: {Jacobian.shape}')
```

(continues on next page)

(continued from previous page)

```
Jacobian = cell.inp_jac(params, inputs, current_state)
print(f'Input Jacobian has shape: {Jacobian.shape}')
```

```
Recurrent Jacobian has shape: (32, 32)
Input Jacobian has shape: (32, 2)
```

`renn` also contains helper functions for numerically finding fixed points of the RNN, for building and training different RNN architectures, and for training and analyzing RNN optimizers.

In future tutorials, we will explore some of these additional use cases!

CHAPTER 4

Tests and code format

Tests are run using pytest. From the project root directory, simply run: `pytest` to run the tests. You will need to have pytest installed (try *pip install pytest* to install it).

Formatting checks are doing via yapf, enabled automatically by pre-commit. To get this setup, first make sure pre-commit is installed (*pip install pre-commit*) and then run `pre-commit install` from the project root directory.

CHAPTER 5

Building the docs

To rebuild the documentation, first install the dependencies: `pip install -r docs/requirements.txt`

First, generate the source API documentation by running `sphinx-apidoc -f -o docs/source renn` from the root directory. Then, the commands to build the docs are contained in the *docs/Makefile* file.

CHAPTER 6

How to contribute

6.1 Contributor License Agreement

Contributions to this project must be accompanied by a Contributor License Agreement. You (or your employer) retain the copyright to your contribution; this simply gives us permission to use and redistribute your contributions as part of the project. Head over to <https://cla.developers.google.com/> to see your current agreements on file or to sign a new one.

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

6.2 Code reviews

All submissions, including submissions by project members, require review. We use GitHub pull requests for this purpose. Consult GitHub Help for more information on using pull requests.

6.3 Community Guidelines

This project follows Google's Open Source Community Guidelines.

CHAPTER 7

renn package

7.1 Subpackages

7.1.1 renn.data package

Submodules

renn.data.data_utils module

Data utils

renn.data.data_utils.**column_parser**(*text_column*)
Returns a parser which parses a row of a csv file containing labeled data, extracting the label and the text

This parser assumes the label is the zeroth element of the row, and the text is the 'text_column' element

renn.data.data_utils.**readfile**(*filename*, *parse_row*)
Reads a csv file containing labeled data, where the function parse_row() extracts a score and text from the labeled data

renn.data.data_utils.**sentiment_relabel**(*num_classes*)
Returns a function which relabels (initially five-class) sentiment labels for subclassing the Yelp and Amazon datasets.

renn.data.datasets module

Datasets.

renn.data.datasets.**ag_news**(*split*, *vocab_file*, *sequence_length=100*, *batch_size=64*, *transform_fn=<function identity>*, *filter_fn=None*, *data_dir=None*)
Loads the ag news dataset.

renn.data.datasets.**goemotions**(*split*, *vocab_file*, *sequence_length=50*, *batch_size=64*, *emotions=None*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*)

Loads the goemotions dataset.

renn.data.datasets.**imdb**(*split*, *vocab_file*, *sequence_length=1000*, *batch_size=64*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*)

Loads the imdb reviews dataset.

renn.data.datasets.**snli**(*split*, *vocab_file*, *sequence_length=75*, *batch_size=64*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*)

Loads the SNLI dataset.

renn.data.datasets.**tokenize_fun**(*tokenizer*)

Standard text processing function.

renn.data.datasets.**mnist**(*split*, *order='row'*, *batch_size=64*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*, *classes=None*)

Loads the serialized MNIST dataset.

Parameters – the subset of classes to keep. (*classes*) – If None, all will be kept

renn.data.datasets.**yelp**(*split*, *num_classes*, *vocab_file*, *sequence_length=1000*, *batch_size=64*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*)

Loads the yelp reviews dataset.

renn.data.datasets.**dbpedia**(*split*, *num_classes*, *vocab_file*, *sequence_length=1000*, *batch_size=64*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*)

Loads the dpedia text classification dataset.

renn.data.datasets.**amazon**(*split*, *num_classes*, *vocab_file*, *sequence_length=250*, *batch_size=64*, *transform=<function identity>*, *filter_fn=None*, *data_dir=None*)

Loads the yelp reviews dataset.

renn.data.synthetic module

Synthetic Datasets.

class renn.data.synthetic.**Unordered**(*num_classes=3*, *batch_size=64*, *length_sampler='Constant'*, *sampler_params={'value': 40}*)

Bases: object

Synthetic dataset representing un-ordered classes, to mimic e.g. text-classification datasets like AG News (unlike, say, star-prediction or sentiment analysis, which features ordered classes

label_batch(*batch*)

Calculates class labels for a batch of sentences

score(*sentence*, *length*)

Calculates the score, i.e. the amount of accumulated evidence in the sentence, for each class

renn.data.tokenizers module

Text processing.

renn.data.tokenizers.**build_vocab**(*corpus_generator*, *vocab_size*, *split_fun=<method 'split' of 'str' objects>*)

Builds a vocab file from a text generator.

renn.data.tokenizers.**load_tokenizer**(*vocab_file*, *default_value=-1*)
Loads a tokenizer from a vocab file.

renn.data.wordpiece_tokenizer_learner_lib module

Algorithm for learning wordpiece vocabulary.

class renn.data.wordpiece_tokenizer_learner_lib.**Params**(*upper_thresh*, *lower_thresh*, *num_iterations*, *max_input_tokens*, *max_token_length*, *max_unique_chars*, *vocab_size*, *slack_ratio*, *include_joiner_token*, *joiner*, *reserved_tokens*)

Bases: tuple

include_joiner_token
Alias for field number 8

joiner
Alias for field number 9

lower_thresh
Alias for field number 1

max_input_tokens
Alias for field number 3

max_token_length
Alias for field number 4

max_unique_chars
Alias for field number 5

num_iterations
Alias for field number 2

reserved_tokens
Alias for field number 10

slack_ratio
Alias for field number 7

upper_thresh
Alias for field number 0

vocab_size
Alias for field number 6

renn.data.wordpiece_tokenizer_learner_lib.**ensure_all_tokens_exist**(*input_tokens*, *output_tokens*, *include_joiner_token*, *joiner*)

Adds all tokens in input_tokens to output_tokens if not already present.

Parameters

- **input_tokens** – set of strings (tokens) we want to include

- **output_tokens** – string to int dictionary mapping token to count
- **include_joiner_token** – bool whether to include joiner token
- **joiner** – string used to indicate suffixes

Returns string to int dictionary with all tokens in input_tokens included

renn.data.wordpiece_tokenizer_learner_lib.**extract_char_tokens**(*word_counts*)

Extracts all single-character tokens from word_counts.

Parameters **word_counts** – list of (string, int) tuples

Returns set of single-character strings contained within word_counts

renn.data.wordpiece_tokenizer_learner_lib.**filter_input_words**(*all_counts*, *allowed_chars*, *max_input_tokens*)

Filters out words with unallowed chars and limits words to max_input_tokens.

Parameters

- **all_counts** – list of (string, int) tuples
- **allowed_chars** – list of single-character strings
- **max_input_tokens** – int, maximum number of tokens accepted as input

Returns list of (string, int) tuples of filtered wordcounts

renn.data.wordpiece_tokenizer_learner_lib.**generate_final_vocabulary**(*reserved_tokens*, *char_tokens*, *curr_tokens*)

Generates final vocab given reserved, single-character, and current tokens.

Parameters

- **reserved_tokens** – list of strings (tokens) that must be included in vocab
- **char_tokens** – set of single-character strings
- **curr_tokens** – string to int dict mapping token to count

Returns list of strings representing final vocabulary

renn.data.wordpiece_tokenizer_learner_lib.**get_allowed_chars**(*all_counts*, *max_unique_chars*)

Get the top max_unique_chars characters within our wordcounts.

We want each character to be in the vocabulary so that we can keep splitting down to the character level if necessary. However, in order not to inflate our vocabulary with rare characters, we only keep the top max_unique_chars characters.

Parameters

- **all_counts** – list of (string, int) tuples
- **max_unique_chars** – int, maximum number of unique single-character tokens

Returns set of strings containing top max_unique_chars characters in all_counts

renn.data.wordpiece_tokenizer_learner_lib.**get_input_words**(*word_counts*, *reserved_tokens*, *max_token_length*)

Filters out words that are longer than max_token_length or are reserved.

Parameters

- **word_counts** – list of (string, int) tuples
- **reserved_tokens** – list of strings
- **max_token_length** – int, maximum length of a token

Returns list of (string, int) tuples of filtered wordcounts

renn.data.wordpiece_tokenizer_learner_lib.**get_search_threshs**(*word_counts*, *upper_thresh*, *lower_thresh*)

Clips the thresholds for binary search based on current word counts.

The upper threshold parameter typically has a large default value that can result in many iterations of unnecessary search. Thus we clip the upper and lower bounds of search to the maximum and the minimum wordcount values.

Parameters

- **word_counts** – list of (string, int) tuples
- **upper_thresh** – int, upper threshold for binary search
- **lower_thresh** – int, lower threshold for binary search

Returns int, clipped upper threshold for binary search lower_search: int, clipped lower threshold for binary search

Return type upper_search

renn.data.wordpiece_tokenizer_learner_lib.**get_split_indices**(*word*, *curr_tokens*, *include_joiner_token*, *joiner*)

Gets indices for valid substrings of word, for iterations > 0.

For iterations > 0, rather than considering every possible substring, we only want to consider starting points corresponding to the start of wordpieces in the current vocabulary.

Parameters

- **word** – string we want to split into substrings
- **curr_tokens** – string to int dict of tokens in vocab (from previous iteration)
- **include_joiner_token** – bool whether to include joiner token
- **joiner** – string used to indicate suffixes

Returns list of ints containing valid starting indices for word

renn.data.wordpiece_tokenizer_learner_lib.**learn**(*word_counts*, *params*)

Takes in wordcounts and returns wordpiece vocabulary.

Parameters

- **word_counts** – list of (string, int) tuples
- **params** – Params namedtuple, parameters for learning

Returns string, final vocabulary with each word separated by newline

renn.data.wordpiece_tokenizer_learner_lib.**learn_binary_search**(*word_counts*, *lower*, *upper*, *params*)

Performs binary search to find wordcount frequency threshold.

Given upper and lower bounds and a list of (word, count) tuples, performs binary search to find the threshold closest to producing a vocabulary of size vocab_size.

Parameters

- **word_counts** – list of (string, int) tuples
- **lower** – int, lower bound for binary search
- **upper** – int, upper bound for binary search
- **params** – Params namedtuple, parameters for learning

Returns list of strings, vocab that is closest to target vocab_size

renn.data.wordpiece_tokenizer_learner_lib.**learn_with_thresh**(*word_counts*, *thresh*, *params*)

Wordpiece learning algorithm to produce a vocab given frequency threshold.

Parameters

- **word_counts** – list of (string, int) tuples
- **thresh** – int, frequency threshold for a token to be included in the vocab
- **params** – Params namedtuple, parameters for learning

Returns list of strings, vocabulary generated for the given thresh

Module contents

7.1.2 renn.metaopt package

Subpackages

renn.metaopt.task_lib package

Submodules

renn.metaopt.task_lib.quadratic module

Defines quadratic loss functions.

renn.metaopt.task_lib.quadratic.**loguniform**(*n*, *lambda_min*, *lambda_max*, *precision=<PrecisionConfig_Precision.HIGHEST: 2>*)

Quadratic loss function with loguniform eigenvalues.

The loss is: f(x) = (1/2) x^T H x + x^T v + b.

The eigenvalues of the Hessian (H) are sampled uniformly on a logarithmic grid from lambda_min to lambda_max.

Parameters

- **n** – int, Problem dimension (number of parameters).
- **lambda_min** – float, Minimum eigenvalue of the Hessian.
- **lambda_max** – float, Maximum eigenvalue of the Hessian.
- **precision** – Which lax precision to use (default: HIGHEST).

Returns

Function that takes a jax PRNGkey and a precision argument and returns an (initial_params, loss_fun) tuple.

Return type problem_fun

renn.metaopt.task_lib.quadratic.**quadform**(*hess*, *x*, *precision*)
Computes a quadratic form (x^T @ H @ x).

Module contents

Submodules

renn.metaopt.api module

Meta-optimization framework.

renn.metaopt.api.**build_metaobj**(*problem_fun*, *optimizer_fun*, *num_inner_steps*, *meta_loss=<function mean>*, *l2_penalty=0.0*, *decorator=<function checkpoint>*)
Builds a meta-objective function.

Parameters

- **problem_fun** – callable, Takes a PRNGKey argument and returns initial parameters and a loss function.
- **optimizer_fun** – callable, Takes a PRNGKey argument and returns an optimizer tuple (as in jax.experimental.optimizers).
- **num_inner_steps** – int, Number of optimization steps.
- **meta_loss** – callable, Function to use to compute a scalar meta-loss.
- **l2_penalty** – float, L2 penalty to apply to the meta-parameters.
- **decorator** – callable, Optional function to wrap the apply_fun argument to lax.scan. By default, this is jax.remat, which will rematerialize the forward computation when computing the gradient, trading off computation for memory. Using the identity function will turn off remat.

Returns

callable, Function that takes meta-parameters and a PRNGKey and returns a scalar meta-objective and the inner loss history.

Return type meta_objective

renn.metaopt.api.**clip**(*x*, *value=inf*)
Clips elements of x to have magnitude less than or equal to value.

renn.metaopt.api.**evaluate**(*opt*, *problem_fun*, *num_steps*, *eval_key*, *num_repeats=64*)
Evaluates an optimizer on a given problem.

Parameters

- **opt** – An optimizer tuple of functions (init_opt, update_opt, get_params) to evaluate.
- **problem_fun** – A function that returns an (initial_params, loss_fun, fetch_data) tuple given a PRNGKey.
- **num_steps** – Number of steps to run the optimizer for.

- **eval_key** – Base PRNGKey used for evaluation.
- **num_repeats** – Number of different evaluation seeds to use.

Returns

Array of loss values with shape (num_repeats, num_steps) containing the training loss curve for each random seed.

Return type losses

renn.metaopt.api.**outer_loop**(*key*, *initial_meta_params*, *meta_objective*, *meta_optimizer*, *steps*, *batch_size=1*, *save_every=None*, *clip_value=inf*)

Meta-trains an optimizer.

Parameters

- **key** – Jax PRNG key, used for initializing the inner problem.
- **initial_meta_params** – pytree, Initial meta-parameters.
- **meta_objective** – function, Computes a (scalar) loss given meta-parameters and an array (batch) of random seeds.
- **meta_optimizer** – tuple of functions, Defines the meta-optimizer to use (for example, a jax.experimental.optimizers Optimizer tuple).
- **steps** – A generator that yields integers from (0, num_steps).
- **batch_size** – int, Number of problems to train per batch.
- **save_every** – int, Specifies how often to store auxiliary information. If None, then information is never stored (Default: None).
- **clip_value** – float, Specifies the gradient clipping value (maximum gradient norm) (Default: np.inf).

Returns Final optimized parameters. store: Dict containing saved auxiliary information during optimization.

Return type final_params

renn.metaopt.api.**unroll_for**(*initial_params*, *loss_fun*, *optimizer*, *extract_state*, *steps*)

Runs an optimizer on a given problem, using a for loop.

Note: this is slower to compile than unroll_scan, but can be used to store intermediate computations (such as the optimizer state or problem parameters) at every iteration, for further analysis.

Parameters

- **initial_params** – Initial parameters.
- **loss_fun** – A function that takes (params, step) and returns a loss.
- **optimizer** – A tuple containing an optimizer init function, an update function, and a get_params function.
- **extract_state** – A function that given some optimizer state, returns what from that optimizer state to store. Note that each optimizer state is different, so this function depends on a particular optimizer.
- **steps** – A generator that yields integers from (0, num_steps).

Returns Dictionary containing results to save.

Return type results

renn.metaopt.api.**unroll_scan**(*initial_params*, *loss_fun*, *optimizer*, *num_steps*, *decorator*)
Runs an optimizer on a given problem, using lax.scan.

Note: this will cache parameters during the unrolled loop, and thus uses a lot of device memory, therefore it is not good for simply evaluating (testing) an optimizer. Instead, it is useful for when we need to compute a _derivative_ of some final loss with respect to the optimizer parameters.

Parameters

- **initial_params** – Initial parameters.
- **loss_fun** – A function that takes (params, step) and returns a loss.
- **optimizer** – A tuple containing an optimizer init function, an update function, and a get_params function.
- **num_steps** – int, number of steps to run the optimizer.
- **decorator** – callable, Optional decorator function used to wrap the apply_fun argument to lax.scan.

Returns Problem parameters after running the optimizer. fs: Loss at every step of the loop.

Return type final_params

renn.metaopt.common module

Update functions for common optimizers.

renn.metaopt.common.**adagrad**(*alpha*, *beta*)

renn.metaopt.common.**adam**(*alpha*, *beta1=0.9*, *beta2=0.999*, *eps=1e-05*)

renn.metaopt.common.**cwrnn**(*cell_apply*, *readout_apply*)

renn.metaopt.common.**momentum**(*alpha*, *beta*)

renn.metaopt.common.**nesterov**(*alpha*, *beta*)

renn.metaopt.common.**rmsprop**(*alpha*, *beta=0.9*, *eps=1e-05*)

renn.metaopt.losses module

Functions for computing a scalar objective from a loss curve.

renn.metaopt.losses.**final**(*fs*)
Returns the final loss value.

renn.metaopt.losses.**mean**(*fs*)
Returns the average over the loss values.

renn.metaopt.losses.**nanmin**(*fs*)
Computes the NaN-aware minimum over the loss curve.

renn.metaopt.models module

Define simple learned optimizer models.

renn.metaopt.models.**aggmo**(*key*, *num_terms*)
Aggregated momentum (aggmo).

renn.metaopt.models.**append_to_sequence**(*sequence*, *element*)
Appends an element to a rolling sequence buffer.

Parameters

- **sequence** – a sequence of ndarrays, concatenated along the first dimension.
- **element** – an ndarray to add to the sequence.

Returns

the updated sequence, with the first element removed, the rest of the elements shifted over, and the new element added.

Return type sequence

renn.metaopt.models.**cwrnn**(*key*, *cell*, *input_scale='raw'*, *output_scale=0.001*)
Component-wise RNN Optimizer.

This optimizer applies an RNN to update the parameters of each problem variable independently (hence the name, component-wise). It follows the same approach as in previous work (Andrychowicz et al 2016, Wichrowska et al 2017) that distribute the parameters along the batch dimension of the RNN. This allows us to easily update each parameter in parallel.

Parameters

- **key** – Jax PRNG key to use for initializing parameters.
- **cell** – An RNNCell to use (see renn/rnn/cells.py)
- **input_scale** – str, Specifies how to scale gradient inputs to the RNN. If 'raw', then the gradients are not scaled. If 'log1p', then the scale and the sign of the inputs are split into a length 2 vector, [log1p(abs(g)), sign(g)].
- **output_scale** – float, Constant used to multiply (rescale) the RNN output.

Returns

A tuple containing the RNN parameters and the readout parameters. The RNN parameters themselves are a namedtuple. The readout parameters are also a tuple containing weights and a bias.

optimizer_fun: A function that takes a set of meta_parameters and initializes an optimizer tuple containing functions to initialize the optimizer state, update the optimizer state, and get parameters from the optimizer state.

Return type meta_parameters

renn.metaopt.models.**gradgram**(*key*, *tau*, *scale_grad*, *scale_gram*, *base_grad=0*, *base_gram=0*)
Optimizer that is a function of gradient history and inner products.

renn.metaopt.models.**lds**(*key*, *num_units*, *h0_init=<function zeros>*, *w_init=<function variance_scaling.<locals>.init>*)
Linear dynamical system (LDS) optimizer.

renn.metaopt.models.**linear**(*key*, *tau*, *scale*, *base=0*)
Optimizer that is a linear function of gradient history.

renn.metaopt.models.**linear_dx**(*key*, *tau*, *scale_grad*, *scale_dx*, *base_grad=0*, *base_gram=0*)
Optimizer that is a linear function of gradient and parameter history.

renn.metaopt.models.**momentum**(*key*)
Wrapper for the momentum optimizer.

renn.metaopt.tasks module

Load tasks from the library.

renn.metaopt.tasks.**quad**(*n*, *lambda_min*, *lambda_max*, *precision=<PrecisionConfig_Precision.HIGHEST: 2>*)

Quadratic loss function with loguniform eigenvalues.

The loss is: f(x) = (1/2) x^T H x + x^T v + b.

The eigenvalues of the Hessian (H) are sampled uniformly on a logarithmic grid from lambda_min to lambda_max.

Parameters

- **n** – int, Problem dimension (number of parameters).
- **lambda_min** – float, Minimum eigenvalue of the Hessian.
- **lambda_max** – float, Maximum eigenvalue of the Hessian.
- **precision** – Which lax precision to use (default: HIGHEST).

Returns

Function that takes a jax PRNGkey and a precision argument and returns an (initial_params, loss_fun) tuple.

Return type problem_fun

renn.metaopt.tasks.**two_moons**(*model*, *num_samples=1024*, *l2_pen=0.005*, *seed=0*)

renn.metaopt.tasks.**logistic_regression**(*model*, *features*, *targets*, *l2_pen=0.0*)

Helper function for logistic regression.

renn.metaopt.tasks.**softmax_regression**(*model*, *features*, *targets*, *num_classes*, *l2_pen=0.0*)

Helper function for softmax regression.

Module contents

Meta-optimization framework.

7.1.3 renn.rnn package

Submodules

renn.rnn.cells module

Recurrent neural network (RNN) cells.

class renn.rnn.cells.**LinearRNN**(*A: jax._src.numpy.lax_numpy.array*, *W: jax._src.numpy.lax_numpy.array*, *b: jax._src.numpy.lax_numpy.array*)

Bases: object

Dataclass for storing parameters of a Linear RNN.

apply(*x*, *h*) → jax._src.numpy.lax_numpy.array

Linear RNN Update.

flatten()

class renn.rnn.cells.**RNNCell**(*num_units*, *h_init=<function zeros>*)
Bases: object

Base class for all RNN Cells.

An RNNCell must implement the following methods: init(PRNGKey, input_shape) -> output_shape, rnn_params apply(params, inputs, state) -> next_state

apply(*params*, *inputs*, *state*)

get_initial_state(*params*, *batch_size=None*)
Gets initial RNN states.

Parameters

- **params** – rnn_parameters
- **batch_size** – batch size of initial states to create.

Returns An ndarray with shape (batch size, num_units).

init(*key*, *input_shape*)

init_initial_state(*key*)

class renn.rnn.cells.**StackedCell**(*layers*)
Bases: *renn.rnn.cells.RNNCell*

Stacks multiple RNN cells together.

A stacked RNN cell is specified by a list of RNN cells and (optional) stax.Dense layers in between them.

Note that the full hidden state for this cell is the concatenation of hidden states from all of the cells in the stack.

apply(*params*, *inputs*, *state*)
Applies a single step of a Stacked RNN.

init(*key*, *input_shape*)
Initializes parameters of a Stacked RNN.

class renn.rnn.cells.**GRU**(*num_units*, *gate_bias=0.0*, *w_init=<function variance_scaling.<locals>.init>*, *b_init=<function zeros>*, *h_init=<function zeros>*)
Bases: *renn.rnn.cells.RNNCell*

Gated recurrent unit.

apply(*params*, *inputs*, *state*)

init(*key*, *input_shape*)

class renn.rnn.cells.**LSTM**(*num_units*, *forget_bias=1.0*, *w_init=<function variance_scaling.<locals>.init>*, *b_init=<function zeros>*, *h_init=<function zeros>*)
Bases: *renn.rnn.cells.RNNCell*

Long-short term memory (LSTM).

apply(*params*, *inputs*, *full_state*)

init(*key*, *input_shape*)

class renn.rnn.cells.**VanillaRNN**(*num_units*, *w_init=<function variance_scaling.<locals>.init>*, *b_init=<function zeros>*, *h_init=<function zeros>*)
Bases: *renn.rnn.cells.RNNCell*

Vanilla RNN Cell.

apply(*params*, *inputs*, *state*)
Applies a single step of a Vanilla RNN.

init(*key*, *input_shape*)
Initializes the parameters of a Vanilla RNN.

class renn.rnn.cells.**UGRNN**(*num_units*, *gate_bias=0.0*, *w_init=<function variance_scaling.<locals>.init>*, *b_init=<function zeros>*, *h_init=<function zeros>*)
Bases: *renn.rnn.cells.RNNCell*

Update-gate RNN Cell.

apply(*params*, *inputs*, *state*)

init(*key*, *input_shape*)

renn.rnn.cells.**embedding**(*vocab_size*, *embedding_size*, *initializer=<function orthogonal.<locals>.init>*)
Builds a token embedding.

Parameters

- **vocab_size** – int, Size of the vocabulary (number of tokens).
- **embedding_size** – int, Dimensionality of the embedding.
- **initializer** – Initializer for the embedding (Default: orthogonal).

Returns callable, Initializes the embedding given a key and input_shape. apply_fun: callable, Converts a set of tokens to embedded vectors.

Return type init_fun

renn.rnn.fixed_points module

Fixed point finding routines.

renn.rnn.fixed_points.**build_fixed_point_loss**(*rnn_cell*, *cell_params*)
Builds function to compute speed of hidden states.

Parameters

- **rnn_cell** – an RNNCell instance.
- **cell_params** – RNN parameters to use when applying the RNN.

Returns

function that takes a batch of hidden states and inputs and computes the speed of the corresponding hidden states.

Return type fixed_point_loss_fun

renn.rnn.fixed_points.**find_fixed_points**(*fp_loss_fun*, *initial_states*, *x_star*, *optimizer*, *tolerance*, *steps=range(0, 1000)*)
Run fixed point optimization.

Parameters

- **fp_loss_fun** – Function that computes fixed point speeds.
- **initial_states** – Initial state seeds.
- **x_star** – Input at which to compute fixed points.

- **optimizer** – A jax.experimental.optimizers tuple.
- **tolerance** – Stopping tolerance threshold.
- **steps** – Iterator over steps.

Returns Array of fixed points for each tolerance. loss_hist: Array containing fixed point loss curve. squared_speeds: Array containing the squared speed of each fixed point.

Return type fixed_points

renn.rnn.fixed_points.**exclude_outliers**(*points*, *threshold=inf*, *verbose=False*)
Remove points that are not within some threshold of another point.

renn.rnn.network module

Recurrent neural network (RNN) helper functions.

renn.rnn.network.**build_rnn**(*num_tokens*, *emb_size*, *cell*, *num_outputs=1*)
Builds an end-to-end recurrent neural network (RNN) model.

Parameters

- **num_tokens** – int, Number of different input tokens.
- **emb_size** – int, Dimensionality of the embedding vectors.
- **cell** – RNNCell to use as the core update function (see cells.py).
- **num_outputs** – int, Number of outputs from the readout (Default: 1).

Returns

function that takes a PRNGkey and input_shape and returns expected shapes and initialized embedding, RNN, and readout parameters.

apply_fun: function that takes a tuple of network parameters and batch of input tokens and applies the RNN to each sequence in the batch.

emb_apply: function to just apply the embedding. readout_apply: function to just apply the readout.

Return type init_fun

renn.rnn.network.**mse**(*y*, *yhat*)
Mean squared error loss.

renn.rnn.network.**eigsorted**(*jac*)
Computes sorted eigenvalues and corresponding eigenvectors of a matrix.

Notes

The eigenvectors are stored in the columns of the returned matrices. The right and left eigenvectors are returned, such that: J=REL^T

Parameters **jac** – numpy array used to compute the eigendecomposition (must be square).

Returns right eigenvectors, as columns in the returned array. eigvals: numpy array of eigenvalues. lefts: left eigenvectors, as columns in the returned array.

Return type rights

renn.rnn.network.**timescale**(*eigenvalues*)
Converts eigenvalues into approximate time constants.

renn.rnn.unroll module

Recurrent neural network (RNN) cells.

renn.rnn.unroll.**unroll_rnn**(*initial_states*, *input_sequences*, *rnn_update*, *readout=<function identity>*)

Unrolls an RNN on a batch of input sequences.

Given a batch of initial RNN states, and a batch of input sequences, this function unrolls application of the RNN along the sequence. The RNN state is updated using the *rnn_update* function, and the *readout* is used to convert the RNN state to outputs (defaults to the identity function).

B: batch size. N: number of RNN units. T: sequence length.

Parameters

- **initial_states** – batch of initial states, with shape (B, N).
- **input_sequences** – batch of inputs, with shape (B, T, N).
- **rnn_update** – updates the RNN hidden state, given (inputs, current_states).
- **readout** – applies the readout, given current states. If this is the identity function, then no readout is applied (returns the hidden states).

Returns batch of outputs (batch_size, sequence_length, num_outputs).

Return type outputs

Module contents

7.2 Submodules

7.3 renn.analysis_utils module

Utilities for analysis.

renn.analysis_utils.**pseudogrid**(*coordinates*, *dimension*)

Constructs a pseudogrid ('pseudo' in that it is not necessarily evenly-spaced) of points in 'dimension'-dimension space from the specified coordinates.

Arguments: coordinates: a mapping between dimensions and

coordinates in those dimensions

dimension: number of dimensions

For all dimensions that are not specified, the coordinate is taken to be 0.

Example

if coordinates = {0: [0, 1, 2], 2: [1]},

and dimension = 4, the coordinates in dimensions 1 and 3 will be taken as [0], yielding the effective coordinate-dictionary

coordinates = {0: [0,1,2], 1: [0], 2: [1], 3: [0]}

Then the resulting pseudogrid will be constructed as: [[0,0,1,0], [1,0,1,0], [2,0,1,0]]

7.4 renn.losses module

Functions for computing loss.

renn.losses.**binary_xent**(*logits*, *labels*)

Cross-entropy loss in in a two-class classification problem, where the model output is a single logit

Parameters

- **logits** – array of shape (batch_size, 1) or just (batch_size)
- **labels** – array of length batch_size, whose elements are either 0 or 1

Returns scalar cross entropy loss

Return type loss

renn.losses.**multiclass_xent**(*logits*, *labels*)

7.5 renn.serialize module

Serialization of pytrees.

renn.serialize.**dump**(*pytree*, *file*)

renn.serialize.**load**(*file*)

renn.serialize.**dumps**(*pytree*)

renn.serialize.**loads**(*bytes*)

7.6 renn.utils module

Utilities for optimization.

renn.utils.**batch_mean**(*fun*, *in_axes*)

Converts a function to a batched version (maps over multiple inputs).

This takes a function that returns a scalar (such as a loss function) and returns a new function that maps the function over multiple arguments (such as over multiple random seeds) and returns the average of the results.

It is useful for generating a batched version of a loss function, where the loss function has stochasticity that depends on a random seed argument.

Parameters

- **fun** – function, Function to batch.
- **in_axes** – tuple, Specifies the arguments to fun to batch over. For example, in_axes=(None, 0) would batch over the second argument.

Returns function, computes the average over a batch.

Return type batch_fun

renn.utils.**norm**(*params*, *order=2*)

Computes the (flattened) norm of a pytree.

renn.utils.**identity**(*x*)

Identity function.

renn.utils.**fst**(*xs*)
Returns the first element from a list.

renn.utils.**snd**(*xs*)
Returns the second element from a list.

renn.utils.**optimize**(*loss_fun*, *x0*, *optimizer*, *steps*, *stop_tol=-inf*)
Run an optimizer on a given loss function.

Parameters

- **loss_fun** – Scalar loss function to optimize.
- **x0** – Initial parameters.
- **optimizer** – An tuple of optimizer functions (init_opt, update_opt, get_params) from a jax.experimental.optimizers instance.
- **steps** – Iterator over steps.
- **stop_tol** – Stop if the loss is below this value (Default: -np.inf).

Returns Array of losses during training. final_params: Optimized parameters.

Return type loss_hist

renn.utils.**one_hot**(*labels*, *num_classes*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*)
Creates a one-hot encoding of an array of labels.

Parameters

- **labels** – array of integers with shape (num_examples,).
- **num_classes** – int, Total number of classes.
- **dtype** – optional, jax datatype for the return array (Default: float32).

Returns array with shape (num_examples, num_classes).

Return type one_hot_labels

renn.utils.**compose**(**funcs*)
Returns a function that is the composition of multiple functions.

7.7 renn.version module

7.8 Module contents

RENN core.

CHAPTER 8

v0.1.0

- **Added**
  - Initial publicized release.
  - Added documentation using sphinx.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

S

T

U

V

Y