trax.supervised

decoding

Decoding with Trax models.

trax.supervised.decoding.autoregressive_sample_stream(*model, inputs=None, batch_size=1, temperature=1.0, start_id=0, accelerate=True*)

Yields samples from *model*, in autoregressive language model fashion.

This function uses *model* to generate outputs one position at a time, with access to inputs for the current position and all preceding positions. The new output becomes the next position's input, and further calls to *autoregressive_sample_stream* repeat the process for successive positions indefinitely.

Inputs and outputs always come in batches, even if size 1. If *inputs* is present, it must have shape (*batch_size*, inputs_sequence_length), and each output in the stream has shape (*batch_size*, 1).

Parameters:

- **model** – A layer object (subclass of *trax.layers.Layer*) created in *'predict'* mode and initialized from trained weights. The model must have a structure that allows it to run as an autoregressive one-sample-at-a-time predictor (e.g., *trax.models.TransformerLM*).
- **inputs** – Sequence of symbols the model sees as input the first time it generates an output. If None, the model generates the first output based on just the start symbol.
- **batch_size** – Number of sequences to generate in parallel as a batch.
- **temperature** – Parameter that controls the sharpness of the softmax that feeds the sampling process. Values range from 0.0 (all probability mass goes to one candidate; like an argmax) to positive infinity (all candidates have equal probability).
- **start_id** – Integer representing the start symbol for the autoregressive process.
- **accelerate** – If True, create an accelerated version of *model* and use it for generating outputs.

Yields: Tensor of integers with shape (*batch_size*, 1), representing the batch of outputs for the next position in the stream.

trax.supervised.decoding.autoregressive_sample(*model, inputs=None, batch_size=1, temperature=1.0, start_id=0, eos_id=1, max_length=100, accelerate=True*)

Returns a batch of sequences created by autoregressive sampling.

This function uses *model* to generate outputs one position at a time, with access to inputs for the current position and all preceding positions. The new output becomes the next position's input, and this loop repeats until either the model outputs the *eos_id* value or the output sequence reaches *max_length* items.

Parameters:

- **model** – A layer object (subclass of *trax.layers.Layer*) created in *'predict'* mode and initialized from trained weights. The model must have a structure that allows it to run as autoregressive one-sample-at-a-time predictor (e.g., *trax.models.TransformerLM*).
- **inputs** – Sequence of symbols the model sees as input the first time it generates an output. If None, the model must generate the first output with no input to guide it.
- **batch_size** – Number of sequences to generate in parallel as a batch.
- **temperature** – Parameter that controls the sharpness of the softmax that feeds the sampling process. Values range from 0.0 (all probability mass goes to one candidate; like an argmax) to positive infinity (all candidates have equal probability).
- **start_id** – The start symbol (ID/integer) for the autoregressive process.
- **eos_id** – The end-of-sequence symbol (ID/integer) for the autoregressive process.
- **max_length** – Maximum length for generated sequences.
- **accelerate** – If True, create an accelerated version of *model* and use it for generating outputs.

Returns: Tensor of integers with shape (*batch_size*, output_length) representing a batch of output sequences. output_length is the maximum length of the output sequences, where each sequence can be no longer than *max_length*.

lr_schedules

Learning rate (LR) schedules.

In Trax a learning rate schedule is a function: $\mathrm{step} \mapsto \mathrm{learning\_rate}$. This module provides helpers for constructing such functions. For example:

```
constant(0.001)
```

returns a function that always returns *0.001*.

trax.supervised.lr_schedules.constant(*value*)

Returns an LR schedule that is constant from time (step) 1 to infinity.

trax.supervised.lr_schedules.warmup(*n_warmup_steps, max_value*)

Returns an LR schedule with linear warm-up followed by constant value.

Parameters:
- **n_warmup_steps** – Number of steps during which the learning rate rises on a line connecting (0, 0) and (n_warmup_steps, max_value).
- **max_value** – Value for learning rate after warm-up has finished.

---

trax.supervised.lr_schedules.warmup_and_rsqrt_decay(*n_warmup_steps, max_value*)

Returns an LR schedule with warm-up + reciprocal square root decay.

---

trax.supervised.lr_schedules.multifactor(*factors='constant * linear_warmup * rsqrt_decay', constant=0.1, warmup_steps=400, decay_factor=0.5, steps_per_decay=20000, steps_per_cycle=100000, minimum=0*)

Factor-based learning rate schedule.

Interprets factors in the factors string which can consist of: * constant: interpreted as the constant value, * linear_warmup: interpreted as linear warmup until warmup_steps, * rsqrt_decay: divide by square root of max(step, warmup_steps) * decay_every: Every k steps decay the learning rate by decay_factor. * cosine_deay: Cyclic cosine decay, uses steps_per_cycle parameter.

Parameters:
- **factors** – a string with factors separated by '*' that defines the schedule.
- **constant** – float, the starting constant for the learning rate schedule.
- **warmup_steps** – how many steps to warm up for in the warmup schedule.
- **decay_factor** – The amount to decay the learning rate by.
- **steps_per_decay** – How often to decay the learning rate.
- **steps_per_cycle** – Steps per cycle when using cosine decay.
- **minimum** – if the computed rate is below the minimum, then return the minimum.

Returns: float -> {'learning_rate': float}, the step-dependent lr.

Return type: a function learning_rate(step)

training

Simplified API (under development) for supervised learning/training in Trax.

Trax authors expect that this module will replace *trainer_lib.Trainer*.

Key classes:

- Loop: Core training loop for an n-step training session, starting from random initialization.
- TrainTask: Labeled data + feedback mechanism (loss function w/ optimizer) for modifying a model's weights.
- Optimizer: How to compute model weight updates using loss-derived gradients. May contain state ("slots", 1-1 with model weights) that accumulates across training steps. (This class is defined in the optimizers package.)
- EvalTask: How and when to measure model performance as a function of training step number.

---

class **trax.supervised.training.Loop**(*model, tasks, eval_model=None, eval_tasks=None, output_dir=None, checkpoint_at=None, eval_at=None, which_task=None, n_devices=None, random_seed=None, loss_chunk_size=0, use_memory_efficient_trainer=False*)

Bases: `object`

Loop that can run for a given number of steps to train a supervised model.

Can train the model on multiple tasks by interleaving updates according to the which_task() argument.

The typical supervised training process randomly initializes a model and updates its weights via feedback (loss-derived gradients) from a training task, by looping through batches of labeled data. A training loop can also be configured to run periodic evals and save intermediate checkpoints.

For speed, the implementation takes advantage of JAX's composable function transformations (specifically, *jit* and *grad*). It creates JIT-compiled pure functions derived from variants of the core model; schematically:

- training variant: jit(grad(pure_function(model+loss)))
- evals variant: jit(pure_function(model+evals))

In training or during evals, these variants are called with explicit arguments for all relevant input data, model weights/state, optimizer slots, and random number seeds:

- batch: labeled data
- model weights/state: trainable weights and input-related state (e.g., as used by batch norm)
- optimizer slots: weights in the optimizer that evolve during the training process
- random number seeds: JAX PRNG keys that enable high-quality, distributed, repeatable generation of pseudo-random numbers

> **__init__**(*model, tasks, eval_model=None, eval_tasks=None, output_dir=None, checkpoint_at=None, eval_at=None, which_task=None, n_devices=None, random_seed=None, loss_chunk_size=0, use_memory_efficient_trainer=False*)

Configures a training *Loop*, including a random initialization.

Parameters:

- **model** – Trax layer, representing the core model to be trained. Loss functions and eval functions (a.k.a. metrics) are considered to be outside the core model, taking core model output and data labels as their two inputs.
- **tasks** – List of TrainTask instances, which define the training data, loss function, and optimizer to be used in respective tasks in this training loop. It can also be a single TrainTask instance which is treated in the same way as a singleton list.
- **eval_model** – Optional Trax layer, representing model used for evaluation, e.g., with dropout turned off. If None, the training model (model) will be used.
- **eval_tasks** – List of EvalTask instances which define how to evaluate the model: which validation data to use and which metrics to report. Evaluation on each of the tasks and will run and be reported separately which allows to score a model on different subtasks. This argument can also be None, in which case no evals will be run, or a single EvalTask, which wil be treated in the same way as a singleton list.
- **output_dir** – Path telling where to save outputs (evals and checkpoints). Can be None if both *eval_task* and *checkpoint_at* are None.
- **checkpoint_at** – Function (integer –> boolean) telling, for step n, whether that step should have its checkpoint saved. If None, the default is periodic checkpointing at *task.n_steps_per_checkpoint*.
- **eval_at** – Function (integer –> boolean) that says, for training step n, whether that step should run evals. If None, run when checkpointing.
- **which_task** – Function (integer –> integer) indicating which task should be used at which training step. Can be set to None in single-task training.
- **n_devices** – integer or None, the number of devices for this computation.
- **random_seed** – the random seed to use; time/os dependent if None (default).
- **loss_chunk_size** – int, if > 0 use chunks of this size to make loss computation more more memory-efficient.
- **use_memory_efficient_trainer** – whether to use a special memory-efficient trainer.

`run`(*n_steps=1*)

Runs this training loop for n steps.

Optionally runs evals and saves checkpoints at specified points.

Parameters: **n_steps** – Stop training after completing n steps.

`step`

Returns current step number in this training session.

`n_devices`

Returns the number of devices to be used in this computation.

`is_chief`

Returns true if this Loop is the chief.

`model`

Returns the model that is training.

`eval_model`

Returns the model used for evaluation.

`new_rng`()

Returns a new single-use random number generator (JAX PRNG key).

`run_evals`(*summary_writers=None*)

Runs and records evals for this training session.

Parameters: **summary_writers** – List of per-task Jaxboard summary writers to log metrics.

`save_checkpoint`()

Saves checkpoint to disk for the current training step.

`load_checkpoint`(*directory=None, filename=None*)

Loads model weights and step from a checkpoint on disk.

Parameters:

- **directory** – Directory with the checkpoint (self._output_dir by default).
- **filename** – Checkpoint file name (model.pkl.gz by default).

---

class `trax.supervised.training.TrainTask`(*labeled_data, loss_layer, optimizer, lr_schedule=None, n_steps_per_checkpoint=100*)

Bases: `object`

A supervised task (labeled data + feedback mechanism) for training.

`__init__`(*labeled_data, loss_layer, optimizer, lr_schedule=None, n_steps_per_checkpoint=100*)

Configures a training task.

Parameters:

- **labeled_data** – Iterator of batches of labeled data tuples. Each tuple has 1+ data (input value) tensors followed by 1 label (target value) tensor. All tensors are NumPy ndarrays or their JAX counterparts.
- **loss_layer** – Layer that computes a scalar value (the "loss") by comparing model output $\hat{y} = f(x)$ to the target y.
- **optimizer** – Optimizer object that computes model weight updates from loss-function gradients.
- **lr_schedule** – Learning rate schedule, a function step -> learning_rate.
- **n_steps_per_checkpoint** – How many steps to run between checkpoints.

`labeled_data`

`sample_batch`

`next_batch`()

Returns one batch of labeled data: a tuple of input(s) plus label.

`loss_layer`

`n_steps_per_checkpoint`

`optimizer`

`learning_rate`(*step*)

Return the learning rate for the given step.

---

class `trax.supervised.training.EvalTask`(*labeled_data, metrics, metric_names=None, n_eval_batches=1*)

Bases: `object`

Labeled data plus scalar functions for (periodically) measuring a model.

An eval task specifies how (*labeled_data* + *metrics*) and with what precision (*n_eval_batches*) to measure a model as it is training. The variance of each scalar output is reduced by measuring over multiple (*n_eval_batches*) batches and reporting the average from those measurements.

`__init__`(*labeled_data*, *metrics*, *metric_names=None*, *n_eval_batches=1*)

Configures an eval task: named metrics run with a given data source.

Parameters:

- **labeled_data** – Iterator of batches of labeled data tuples. Each tuple has 1+ data tensors (NumPy ndarrays) followed by 1 label (target value) tensor.
- **metrics** – List of layers; each computes a scalar value per batch by comparing model output $\hat{y} = f(x)$ to the target y.
- **metric_names** – List of names, one for each item in *metrics*, in matching order, to be used when recording/reporting eval output. If None, generate default names using layer names from metrics.
- **n_eval_batches** – Integer N that specifies how many eval batches to run; the output is then the average of the outputs from the N batches.

`labeled_data`

`sample_batch`

`next_batch`()

Returns one batch of labeled data: a tuple of input(s) plus label.

`metrics`

`metric_names`

`n_eval_batches`

`trax.supervised.training.pickle_to_file`(*obj, file_path, gzip=False*)

Pickle obj to file_path with gzipping and failure protection.

`trax.supervised.training.unpickle_from_file`(*file_path, gzip=False*)

Unpickle obj from file_path with gzipping.

`trax.supervised.training.init_host_and_devices`(*n_devices=None, random_seed=None*)

Initializes host and device attributes for this trainer.

Parameters:

- **n_devices** – Number of devices this trainer will use. If *None*, get the number from the backend.
- **random_seed** – Random seed as the starting point for all random numbers used by the trainer. If *None*, calculate one from system time and host id.

Returns:

True if this trainer has special chief responsibilities. host_count: Number of hosts in this computation. n_devices: The passed in value of n_devices or a computed default (for this

host).

random_seed: The passed in value of random_seed or a computed default.

Return type:

is_chief