tuned-lens

Release 0.2.0

Nora Belrose Zach Furman, Logan Smith, Danny Halawi, Lev McK

Mar 31, 2024

API REFERENCE

Tools for understanding how transformer predictions are built layer-by-layer.

This package provides a simple interface for training and evaluating **tuned lenses**. A tuned lens allows us to peek at the iterative computations a transformer uses to compute the next token.

CHAPTER

ONE

WHAT IS A LENS?

A lens into a transformer with *n* layers allows you to replace the last *m* layers of the model with an affine transformation (we call these affine translators). Each affine translator is trained to minimize the KL divergence between its prediction and the final output distribution of the original model. This means that after training, the tuned lens allows you to skip over these last few layers and see the best prediction that can be made from the model's intermediate representations, i.e., the residual stream, at layer *n* - *m*.

The reason we need to train an affine translator is that the representations may be rotated, shifted, or stretched from layer to layer. This training differentiates this method from simpler approaches that unembed the residual stream of the network directly using the unembedding matrix, i.e., the logit lens. We explain this process and its applications in the paper Eliciting Latent Predictions from Transformers with the Tuned Lens.

1.1 Acknowledgments

Originally conceived by Igor Ostrovsky and Stella Biderman at EleutherAI, this library was built as a collaboration between FAR and EleutherAI researchers.

CHAPTER

TWO

INSTALL INSTRUCTIONS

2.1 Installing from PyPI

First, you will need to install the basic prerequisites into a virtual environment:

- Python 3.9+
- PyTorch 1.13.0+

Then, you can simply install the package using pip.

```
pip install tuned-lens
```

2.2 Installing the container

If you prefer to run the training scripts from within a container, you can use the provided Docker container.

```
docker pull ghcr.io/alignmentresearch/tuned-lens:latest
docker run --rm tuned-lens:latest tuned-lens --help
```

CHAPTER

THREE

CONTRIBUTING

Make sure to install the dev dependencies and install the pre-commit hooks.

```
$ git clone https://github.com/AlignmentResearch/tuned-lens.git
$ pip install -e ".[dev]"
$ pre-commit install
```

CHAPTER

FOUR

CITATION

If you find this library useful, please cite it as:

```
@article{belrose2023eliciting,
  title={Eliciting Latent Predictions from Transformers with the Tuned Lens},
  authors={Belrose, Nora and Furman, Zach and Smith, Logan and Halawi, Danny and␣
↪McKinney, Lev and Ostrovsky, Igor and Biderman, Stella and Steinhardt, Jacob},
  journal={to appear},
  year={2023}
}
```

Warning This package has not reached 1.0. Expect the public interface to change regularly and without a major version bumps.

CHAPTER

FIVE

API REFERENCE

tuned_lens.nn.lenses	Provides lenses for decoding hidden states into logits.
tuned_lens.nn.unembed	Provides a class for mapping transformer hidden states to logits (and vice versa).
tuned_lens.plotting	Provides tools for plotting.
tuned_lens.load_artifacts	Load lens artifacts from the hub or locally storage.

5.1 tuned_lens.nn.lenses

Provides lenses for decoding hidden states into logits.

Classes

class tuned_lens.nn.lenses.**Lens**(*unembed*)

Abstract base class for all Lens.

abstract forward(*h*, *idx*)

Decode hidden states into logits.

Return type
Tensor

abstract transform_hidden(*h*, *idx*)

Convert a hidden state to the final hidden just before the unembedding.

Parameters

- **h** – The hidden state to convert.
- **idx** – The layer of the transformer these hidden states come from.

Return type
Tensor

class tuned_lens.nn.lenses.**LogitLens**(*unembed*)

Unembeds the residual stream into logits.

forward(*h*, *idx*)

Decode a hidden state into logits.

Parameters

- **h** – The hidden state to decode.
- **idx** – the layer of the transformer these hidden states come from.

Return type
Tensor

classmethod from_model(*model*)

Create a LogitLens from a pretrained model.

Parameters
model – A pretrained model from the transformers library you wish to inspect.

Return type
LogitLens

transform_hidden(*h*, *idx*)

For the LogitLens, this is the identity function.

Return type
Tensor

class tuned_lens.nn.lenses.**TunedLens**(*unembed*, *config*)

A tuned lens for decoding hidden states into logits.

forward(*h*, *idx*)

Transform and then decode the hidden states into logits.

Return type
Tensor

classmethod from_model(*model*, *model_revision=None*, *bias=True*)

Create a lens from a pretrained model.

Parameters

- **model** – The model to create the lens from.
- **model_revision** – The git revision of the model to used.
- **bias** – Whether to use a bias in the linear translators.

Return type
TunedLens

Returns
A TunedLens instance.

classmethod from_model_and_pretrained(*model*, *lens_resource_id=None*, ***kwargs*)

Load a tuned lens from a folder or hugging face hub.

Parameters

- **model** – The model to create the lens from.
- **lens_resource_id** – The resource id of the lens to load. Defaults to the model's name_or_path.
- ****kwargs** – Additional arguments to pass to *tuned_lens.load_artifacts.load_lens_artifacts()* and th.load.

Return type
TunedLens

Returns
A TunedLens instance whose unembedding is derived from the given model and whose layer translators are loaded from the given resource id.

classmethod from_unembed_and_pretrained(*unembed*, *lens_resource_id*, ***kwargs*)

Load a tuned lens from a folder or hugging face hub.

Parameters

- **unembed** – The unembed operation to use for the lens.
- **lens_resource_id** – The resource id of the lens to load.
- ****kwargs** – Additional arguments to pass to *tuned_lens.load_artifacts.load_lens_artifacts()* and th.load.

Return type
TunedLens

Returns
A TunedLens instance.

generate(*model*, *layer*, *input_ids*, *do_sample=True*, *temp=1.0*, *max_new_tokens=100*)

Generate from the tuned lens at the given layer.

Parameters

- **model** – The base model the generate from. Usually the model this lens trained on.
- **layer** – The layer to generate from.
- **input_ids** – (batch x prompt_len) The input ids to generate from.
- **do_sample** – Whether to use sampling or greedy decoding.
- **temp** – The temperature to use for sampling.
- **max_new_tokens** – The maximum number of tokens to generate.

Return type
Tensor

Returns
The prompt concatenated with the newly generated tokens.

save(*path*, *ckpt='params.pt'*, *config='config.json'*)

Save the lens to a directory.

Parameters

- **path** – The path to the directory to save the lens to.
- **ckpt** – The name of the checkpoint file to save the parameters to.
- **config** – The name of the config file to save the config to.

Return type
None

transform_hidden(*h*, *idx*)

Transform hidden state from layer *idx*.

Return type
Tensor

class tuned_lens.nn.lenses.**TunedLensConfig**(*base_model_name_or_path*, *d_model*, *num_hidden_layers*, *bias=True*, *base_model_revision=None*, *unembed_hash=None*, *lens_type='linear_tuned_lens'*)

A configuration for a TunedLens.

classmethod from_dict(*config_dict*)

Create a config from a dictionary.

to_dict()

Convert this config to a dictionary.

5.2 tuned_lens.nn.unembed

Provides a class for mapping transformer hidden states to logits (and vice versa).

Classes

class tuned_lens.nn.unembed.**InversionOutput**(*preimage*, *grad_norm*, *kl*, *loss*, *nfev*)

Output of *Unemebd.invert*.

class tuned_lens.nn.unembed.**Unembed**(*model*)

Module that maps transformer hidden states to logits (and vice versa).

forward(*h*)

Convert hidden states into logits.

Return type
Tensor

invert(*logits*, ***, *h0=None*, *max_iter=1000*, *optimizer='lbfgs'*, *prior_weight=0.0*, *prior=None*, *step_size=1.0*, *tol=0.001*, *weight=None*)

Project logits onto the image of the unemebed operation.

When the hidden state dimension is smaller than the vocabulary size, the unembed operation cannot perfectly represent arbitrary logits, since its image is restricted to a subspace; this phenomenon is known as the softmax bottleneck (cf. https://arxiv.org/abs/1711.03953). Because of this, the inverse can only be approximate in general. Here, we use gradient-based optimization to find a hidden state that minimizes the KL divergence from the target distribution p to unembeded logits q(h): h* = argmin_h KL(p || q(h)).

Parameters

- **logits** – Tensor of shape *[..., vocab_size]* containing logits to invert.
- **h0** – Initial guess for the hidden state. If *None*, the least-squares solution of the linear equation xU = logits is used, where U is the unembedding matrix.
- **max_iter** – Maximum number of iterations for the optimizer to take.
- **optimizer** – Optimization algorithm to use. Currently, only "lbfgs" and "sgd" are supported.
- **prior_weight** – The weight of the prior distribution is given in the loss.
- **prior** – Prior distribution over hidden states used to regularize the inversion.
- **step_size** – The step size for the optimizer.
- **tol** – Tolerance for the inversion objective.

- **weight** – Optional tensor of shape *[..., vocab_size]* containing weights for each vocabulary item. If *None*, all classes are weighted equally.

Return type
InversionOutput

unembedding_hash()

Hash the unmbedding matrix to identify the model.

Return type
str

5.3 tuned_lens.plotting

Provides tools for plotting.

Modules

tuned_lens.plotting.prediction_trajectory	Plot a lens table for some given text and model.
tuned_lens.plotting.token_formatter	Contains a class for formatting tokens for display in plots.
tuned_lens.plotting.trajectory_plotting	Contains utility classes for creating heatmap visualizations.

5.3.1 tuned_lens.plotting.prediction_trajectory

Plot a lens table for some given text and model.

Classes

class tuned_lens.plotting.prediction_trajectory.**PredictionTrajectory**(*log_probs*, *input_ids*, *targets=None*, *anti_targets=None*, *tokenizer=None*)

Contains the trajectory predictions for a sequence of tokens.

A prediction trajectory is the set of next token predictions produced by the conjunction of a lens and a model when evaluated on a specific sequence of tokens. This class include multiple methods for visualizing different aspects of the trajectory.

anti_targets: Optional[ndarray[Any, dtype[int64]]] = None

(..., seq_len)

property batch_axes: Sequence[int]

Returns the batch axes for the trajectory.

property batch_shape: Sequence[int]

Returns the batch shape of the trajectory.

`cross_entropy`(***kwargs*)

The cross entropy of the predictions to the targets.

Parameters
****kwargs** – are passed to largest_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the cross entropy of the predictions to the targets.

`entropy`(***kwargs*)

The entropy of the predictions.

Parameters
****kwargs** – are passed to largest_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the entropy of the predictions.

`forward_kl`(***kwargs*)

KL divergence of the lens predictions to the model predictions.

Parameters
****kwargs** – are passed to largest_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the KL divergence of the lens predictions to the final output of the model.

`classmethod from_lens_and_cache`(*lens*, *input_ids*, *cache*, *model_logits*, *targets=None*, *anti_targets=None*, *residual_component='resid_pre'*, *mask_input=False*)

Construct a prediction trajectory from a set of residual stream vectors.

Parameters

- **lens** – A lens to use to produce the predictions.
- **cache** – the activation cache produced by running the model.
- **input_ids** – (..., seq_len) Ids that where input into the model.
- **model_logits** – (..., seq_len x d_vocab) the models final output logits.
- **targets** – (..., seq_len) the targets the model is should predict. Used for *cross_entropy()* and *log_prob_diff()* visualization.
- **anti_targets** – (..., seq_len) the incorrect label the model should not predict. Used for *log_prob_diff()* visualization.
- **residual_component** – Name of the stream vector being visualized.
- **mask_input** – Whether to mask the input ids when computing the log probs.

Return type
PredictionTrajectory

Returns
PredictionTrajectory constructed from the residual stream vectors.

classmethod from_lens_and_model(*lens*, *model*, *input_ids*, *tokenizer=None*, *targets=None*, *anti_targets=None*, *mask_input=False*)

Construct a prediction trajectory from a set of residual stream vectors.

Parameters

- **lens** – A lens to use to produce the predictions. Note this should be compatible with the model.
- **model** – A Hugging Face causal language model to use to produce the predictions.
- **tokenizer** – The tokenizer to use for decoding the input ids.
- **input_ids** – (seq_len) Ids that where input into the model.
- **targets** – (seq_len) the targets the model is should predict. Used for *cross_entropy()* and *log_prob_diff()* visualization.
- **anti_targets** – (seq_len) the incorrect label the model should not predict. Used for *log_prob_diff()* visualization.
- **residual_component** – Name of the stream vector being visualized.
- **mask_input** – Whether to mask the input ids when computing the log probs.

Return type
PredictionTrajectory

Returns
PredictionTrajectory constructed from the residual stream vectors.

input_ids: ndarray[Any, dtype[int64]]

(..., seq_len)

js_divergence(*other*, ***kwargs*)

Compute the JS divergence between self and other prediction trajectory.

Parameters

- **other** – The other prediction trajectory to compare to.
- ****kwargs** – are passed to largest_delta_in_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the JS divergence between self and other.

kl_divergence(*other*, ***kwargs*)

Compute the KL divergence between self and other prediction trajectory.

Parameters

- **other** – The other prediction trajectory to compare to.
- ****kwargs** – are passed to largest_delta_in_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the KL divergence between self and other.

`log_prob_diff`(*delta=False*)

The difference in logits between two tokens.

Return type
TrajectoryStatistic

Returns
The difference between the log probabilities of the two tokens.

`log_probs: ndarray[Any, dtype[float32]]`

(..., n_layers, seq_len, vocab_size) The log probabilities of the predictions for each hidden layer + the models logits

`max_probability`(***kwargs*)

Max probability of the among the predictions.

Parameters
****kwargs** – are passed to largest_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the max probability of the among the predictions.

`property model_log_probs: ndarray[Any, dtype[float32]]`

Returns the log probs of the model (..., seq_len, vocab_size).

`property n_batch_axis: int`

Returns the number of batch dimensions.

`property num_layers: int`

Returns the number of layers in the stream not including the model output.

`property num_tokens: int`

Returns the number of tokens in this slice of the sequence.

`property probs: ndarray[Any, dtype[float32]]`

Returns the probabilities of the predictions.

`rank`(*show_ranks=False*, ***kwargs*)

The rank of the targets among the predictions.

That is, if the target is the most likely prediction, its rank is 1; the second most likely has rank 2, etc.

Parameters

- **show_ranks** – Whether to show the the rank of the target or the top token.
- ****kwargs** – are passed to largest_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the rank of the targets among the predictions.

`slice_sequence`(*slice*)

Create a slice of the prediction trajectory along the sequence dimension.

Return type
PredictionTrajectory

`targets: Optional[ndarray[Any, dtype[int64]]] = None`

(..., seq_len)

`total_variation`(*other*, ***kwargs*)

Total variation distance between self and other prediction trajectory.

Parameters

- **other** – The other prediction trajectory to compare to.
- ****kwargs** – are passed to largest_delta_in_prob_labels.

Return type
TrajectoryStatistic

Returns
A TrajectoryStatistic with the total variational distance between self and other.

`property vocab_size: int`

Returns the size of the vocabulary.

5.3.2 tuned_lens.plotting.token_formatter

Contains a class for formatting tokens for display in plots.

Classes

`class tuned_lens.plotting.token_formatter.TokenFormatter`(*ellipsis='...'*, *newline_replacement='\\\\n'*, *newline_token='Ċ'*, *whitespace_token='Ġ'*, *whitespace_replacement='_'*, *max_string_len=7*)

Format tokens for display in a plots.

`format`(*token*)

Format a token for display in a plot.

Return type
`str`

`pad_token_repr_to_max_len`(*token_repr*)

Pad a token representation to the max string length.

Return type
`str`

5.3.3 tuned_lens.plotting.trajectory_plotting

Contains utility classes for creating heatmap visualizations.

Functions

```
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

tuned_lens.plotting.trajectory_plotting.**trunc_string_left**(*string*, *new_len*)

Truncate a string to the left.

Return type
str

Classes

class tuned_lens.plotting.trajectory_plotting.**TrajectoryLabels**(*label_strings*, *hover_over_entries=None*)

Contains sets of labels for each layer and position in the residual stream.

hover_over_entries: Optional[ndarray[Any, dtype[str_]]] = None

(n_layers x sequence_length x rows x cols) table of strings to display when hovering over a cell. For example, the top k prediction from the lens.

label_strings: ndarray[Any, dtype[str_]]

(n_layers x sequence_length) label for each layer and position in the stream.

stride(*stride*)

Return a new TrajectoryLabels with the given stride.

Parameters
stride – The number of layers between each layer we keep.

Return type
TrajectoryLabels

Returns
A new TrajectoryLabels with the given stride.

template_and_customdata(*col_width_limit=10*)

Construct a template for use with Plotly's hovertemplate.

Return type
Tuple[str, ndarray[Any, dtype[str_]]]

class tuned_lens.plotting.trajectory_plotting.**TrajectoryStatistic**(*name*, *stats*, *sequence_labels=None*, *trajectory_labels=None*, *units=None*, *max=None*, *min=None*, *includes_output=True*, *_layer_labels=None*)

This class represents a trajectory statistic that can be visualized.

For example, the entropy of the lens predictions at each layer.

clip(*min*, *max*)

Return a new TrajectoryStatistic with the given min and max.

Parameters

- **min** – The minimum value to clip to.
- **max** – The maximum value to clip to.

Return type
TrajectoryStatistic

Returns
A new TrajectoryStatistic with the given min and max.

figure(*title=''*, *colorscale='rdbu_r'*, *token_width=80*)

Produce a heatmap plot of the statistic.

Parameters

- **title** – The title of the plot.
- **colorscale** – The colorscale to use for the heatmap.
- **token_width** – The width of each token in the plot.

Return type
Figure

Returns
The plotly heatmap figure.

heatmap(*colorscale='rdbu_r'*, *log_scale=False*, ***kwargs*)

Returns a Plotly Heatmap object for this statistic.

Parameters

- **colorscale** – The colorscale to use for the heatmap.
- **log_scale** – Whether to use a log scale for the colorbar.
- ****kwargs** – Additional keyword arguments to pass to the Heatmap constructor.

Return type
Heatmap

Returns
A plotly Heatmap where the x-axis is the sequence dimension, the y-axis is the layer dimension, and the color of each cell is the value of the statistic.

includes_output: bool = True

Whether the statistic includes the final output layer.

max: Optional[float] = None

The maximum value of the statistic.

min: Optional[float] = None

The minimum value of the statistic.

name: str

The name of the statistic. For example, "entropy".

sequence_labels: Optional[ndarray[Any, dtype[str_]]] = None

(sequence_length) labels for the sequence dimension e.g. input tokens.

stats: ndarray[Any, dtype[float32]]

(n_layers x sequence_length) value of the statistic across layer and position.

stride(*stride*)

Return a new TrajectoryStatistic with the given stride.

Parameters
stride – The number of layers between each layer we keep.

Return type
TrajectoryStatistic

Returns
A new TrajectoryStatistic with the given stride.

trajectory_labels: Optional[*TrajectoryLabels*] = None

Labels for each layer and position in the stream. For example, the top 1 prediction from the lens at each layer.

units: Optional[str] = None

The units of the statistic.

5.4 tuned_lens.load_artifacts

Load lens artifacts from the hub or locally storage.

Functions

```
# empty test needed in case the module has no example usage.
# otherwise, testsetup throws an error
pass
```

tuned_lens.load_artifacts.**available_lens_artifacts**(*repo_id*, *repo_type*, *revision='main'*, *config_file='config.json'*, *ckpt_file='params.pt'*, *subfolder='lens'*)

Get the available lens artifacts from the hub.

Return type
set[str]

tuned_lens.load_artifacts.**load_lens_artifacts**(*resource_id*, *repo_id=None*, *repo_type=None*, *revision='main'*, *config_file='config.json'*, *ckpt_file='params.pt'*, *subfolder='lens'*, *cache_dir=None*)

First checks for lens resource locally then tries to download it from the hub.

Parameters

- **resource_id** – The id of the lens resource.

- **repo_id** – The repository to download the lens from. Defaults to 'AlignmentResearch/tuned-lens'. However, this default can be overridden by setting the TUNED_LENS_REPO_ID environment variable.
- **repo_type** – The type of repository to download the lens from. Defaults to 'space'. However, this default can be overridden by setting the TUNED_LENS_REPO_TYPE environment variable.
- **config_file** – The name of the config file in the folder contain the lens.
- **ckpt_file** – The name of the checkpoint file in the folder contain the lens.
- **revision** – The revision of the lens to download.
- **subfolder** – The subfolder of the repository to download the lens from.
- **cache_dir** – The directory to cache the lens in.

Return type
`tuple[Path, Path]`

Returns

- The path to the config.json file
- The path to the params.pt file

Raises
ValueError – if the lens resource could not be found.

5.5 Loading a pre-trained lens

5.5.1 From the hugging face API

First check if there is a pre-trained lens available in our spaces' pre-trained lenses folder.

Once you have found a lens that you want to use, you can simply load it. A tuned lens is always associated with a model that was used to train it so first load the model and then the lens.

```
>>> import torch
>>> from tuned_lens import TunedLens
>>> from transformers import AutoModelForCausalLM
>>> model = AutoModelForCausalLM.from_pretrained('EleutherAI/pythia-160m-deduped-v0')
>>> tuned_lens = TunedLens.from_model_and_pretrained(model)
```

If you want to load from your own code space you can override the default by providing the correct environment variables see *tuned_lens.load_artifacts*.

5.5.2 From the a local folder

If you have trained a lens and want to load it for inference simply pass the model used to train it and the folder you saved it to.

```
>>> lens = TunedLens.from_model(model)
>>> # Do some thing
>>> lens.save(directory_path)
>>> lens = TunedLens.from_model_and_pretrained(model, directory_path)
```

Note the folder structure must look as follows:

```
path/to/folder
├── config.json
└── params.pt
```

If you saved the model using `tuned_lens.save("path/to/folder")` then this should already be the case.

5.6 Training and evaluating lenses

In this section, we will discuss some of the technical details of training and evaluating your own lenses. First, we will briefly discuss single GPU training and evaluation. Then we will dive into some of the more technical aspects of training a model.

5.6.1 Downloading the Dataset

Before we can start training, we will need to set up our dataset. The experiments in the paper were run on the pythia models by training thus we train our lenses on the validation set of the pile. Let's first go ahead and download the validation and test splits of the pile.

```
wget https://the-eye.eu/public/AI/pile/val.jsonl.zst
unzstd val.jsonl.zst
wget https://the-eye.eu/public/AI/pile/test.jsonl.zst
unzstd test.jsonl.zst
```

5.6.2 Training a Lens

This command will train a tuned lens on *https://github.com/EleutherAI/pythia* with the default hyperparameters. The model will be automatically downloaded from the Hugging Face Hub and cached locally. You can adjust the per GPU batch size to maximize your GPU utilization.

```
python -m tuned_lens train \
    --model.name EleutherAI/pythia-160m-deduped \
    --data.name val.jsonl \
    --per_gpu_batch_size=1 \
    --output my_lenses/EleutherAI/pythia-160m-deduped
```

Once training is completed, this should save the trained lens to the *trained-lenses/pythia-160m-deduped* directory.

5.6.3 Evaluating a Lens

Once you have a lens trained, either by training it yourself, or by loading it from the hub, you can run various evaluations on it using the provided evaluation command.

```
python -m tuned_lens eval \
     --data.name test.jsonl \
     --model.name EleutherAI/pythia-160m-deduped \
     --tokens 16400000 \
     --lens_name my_lenses/EleutherAI/pythia-160m-deduped \
     --output evaluation/EleutherAI/pythia-160m-deduped
```

5.6.4 Distributed Data Parallel Multi-GPU Training

You can also use torch elastic launch to do multi-GPU training. This will default to doing distributed data parallel training for the lens. Note that this still requires the transformer model itself to fit on a single GPU. However, since we are almost always using some form of gradient accumulation, this usually still speeds up training significantly.

```
torchrun \
--standalone \
--nnodes=1 \
--nproc-per-node=<num_gpus> \
-m tuned_lens train \
--model.name EleutherAI/pythia-160m-deduped \
--data.name val.jsonl \
--per_gpu_batch_size=1 \
--output my_lenses/EleutherAI/pythia-160m-deduped
```

5.6.5 Fully Sharded Data Parallel Multi-GPU Training

If the transformer model does not fit on a single GPU, you can also use fully sharded data parallel training. Note that the lens is still trained using DDP, only the transformer itself is sharded. To enable this, you can pass the *–fsdp* flag.

```
torchrun \
--standalone \
--nnodes=1 \
--nproc-per-node=<num_gpus> \
-m tuned_lens train \
--model.name EleutherAI/pythia-160m-deduped \
--data.name val.jsonl \
--per_gpu_batch_size=1 \
--output my_lenses/EleutherAI/pythia-160m-deduped \
--fsdp
```

You can also use cpu offloading to train lenses on very large models while using less VRAM it can be enabled with the `--cpu_offload` flag. However, this substantially slows down training and is still experimental.

5.6.6 Checkpoint Resume

If you are running on a cluster with preemption you may want to be able to run a run with checkpoint resume. This can be enabled by passing the *–checkpoint_freq* flag with a number of steps between checkpoints. By default checkpoints are saved to `<output>/checkpoints` this can be overridden with the `--checkpoint_dir` flag. There is a known issue with combining this with the zero optimizer, see [this issue](https://github.com/AlignmentResearch/tuned-lens/issues/96).

If checkpoints are present in the checkpoints dir, the trainer will automatically resume from the latest one.

5.6.7 Loading the Model Weights in int8

The *–precision int8* flag can be used to load the model's weights in a quantized int8 format. The *bitsandbytes* library must be installed for this to work. This should reduce VRAM usage by roughly a factor of two relative to float16 precision. Unfortunately, this option cannot be combined with *–fsdp* or *–cpu_offload*.

5.6.8 Weights & Biases Logging

To enable logging to `wandb`, you can pass the `--wandb <name-of-run>` flag. This will log the training and evaluation metrics to `wandb`. You will need to set the `WANDB_API_KEY`, `WANDB_ENTITY` and `WANDB_PROJECT` environment variables in your environment. You can find your API key on your wandb profile page. To make this easy, you can create a `.env` file in the root of the project with the following contents.

```
# .env
WANDB_API_KEY= # your-api-key
WANDB_ENTITY= # your-entity
WANDB_PROJECT= # your-project-name
```

Then you can source it when you start your shell by running `source .env`. For additional `wandb` environment variables, see here.

5.6.9 Uploading to the Hub

Once you have trained a lens for a new model if you are feeling generous you can upload it to our hugging face hub space and share it with the world.

To do this first create a pull request on the community tab.

Follow the commands to clone the repo and checkout your pr branch.

> **Warning:** Hugging face hub uses git-lfs to store large files. As a result you should generally work with *GIT_LFS_SKIP_SMUDGE=1* set when running *git clone* and *git checkout* commands.

Once you have checked out your branch you're branch copy the *config.json* and *params.pt* produced by the training run to lens/<model-name> in the repo. Then add and commit the changes.

Note: You shouldn't have to use *GIT_LFS_SKIP_SMUDGE=1* when adding and committing files.

Finally, in your pr description include the following information: * The model name * The dataset used to train the lens * The training command used to train the lens * And ideally, a link to the wandb run

We will review your pr and merge you're lens into the space. Thank you for contributing!

5.7 Comparing prediction trajectories

A prediction trajectory is the set of latent predictions produced by running a lens against each layer of a model. This process creates a sequence of distributions over the next token that in general become more accurate the high in the model they are sourced from. You can think of these distributions as the best guesses that can be made about the final token distribution from by the lenses' affine translator for that layer.

Since we generally care about more than just one token the sequence of predictions is represented as a 3 dimensional tensor we call the prediction trajectory. This tensor has the shape `(num_layers x sequence_length x vocab_size)`. These distributions are typically stored in log space for numerical precision reasons.

In order to start visualizing and playing with prediction trajectories we will first need to load our model and lens.

[1]:
```
import torch
from tuned_lens.nn.lenses import TunedLens
from transformers import AutoModelForCausalLM, AutoTokenizer

device = torch.device('cpu')
# To try a diffrent modle / lens check if the lens is avalible then modify this code
model = AutoModelForCausalLM.from_pretrained('EleutherAI/pythia-160m-deduped')
model = model.to(device)
tokenizer = AutoTokenizer.from_pretrained('EleutherAI/pythia-160m-deduped')
tuned_lens = TunedLens.from_model_and_pretrained(model, map_location=device)
tuned_lens = tuned_lens.to(device)
```

Now lets prepare some interesting text to examine. Here we will use a quote from Tolkien that has some nice repetition. It's also common enough that it was probably in the training data so modifying it will hopefully let us see some conflicts between the model's parametric knowledge and it's in context learning.

[2]:
```
input_ids_ring = tokenizer.encode(
    "One Ring to rule them all,\n"
    "One Ring to find them,\n"
    "One Ring to bring them all\n"
    "and in the darkness bind them"
)

input_ids_model = tokenizer.encode(
    "One Model to rule them all,\n"
    "One Model to find them,\n"
    "One Model to bring them all\n"
    "and in the darkness bind them"
)


targets_ring = input_ids_ring[1:] + [tokenizer.eos_token_id]
targets_model = input_ids_model[1:] + [tokenizer.eos_token_id]
```

Let's validate that the tokenizations line up and this is indeed going to be a one token substitution.

[3]:
```
print(tokenizer.convert_ids_to_tokens(input_ids_ring))
print(tokenizer.convert_ids_to_tokens(input_ids_model))
```

```
['One', 'ĠRing', 'Ġto', 'Ġrule', 'Ġthem', 'Ġall', ',', 'Ċ', 'One', 'ĠRing', 'Ġto', 'Ġfind
↪', 'Ġthem', ',', 'Ċ', 'One', 'ĠRing', 'Ġto', 'Ġbring', 'Ġthem', 'Ġall', 'Ċ', 'and',
↪'Ġin', 'Ġthe', 'Ġdarkness', 'Ġbind', 'Ġthem']
['One', 'ĠModel', 'Ġto', 'Ġrule', 'Ġthem', 'Ġall', ',', 'Ċ', 'One', 'ĠModel', 'Ġto',
↪'Ġfind', 'Ġthem', ',', 'Ċ', 'One', 'ĠModel', 'Ġto', 'Ġbring', 'Ġthem', 'Ġall', 'Ċ',
↪'and', 'Ġin', 'Ġthe', 'Ġdarkness', 'Ġbind', 'Ġthem']
```

Now lets generate a prediction trajectory to examine the third line in tolken's epigrame; That line is consists of tokens [14, 21].

[4]:
```
from tuned_lens.plotting import PredictionTrajectory

third_line = slice(14, 21)

predictition_traj_ring = PredictionTrajectory.from_lens_and_model(
    tuned_lens,
    model,
    tokenizer=tokenizer,
    input_ids=input_ids_ring,
    targets=targets_ring,
).slice_sequence(third_line)
```

Now let's visualize the prediction trajectory for this slice of the tranformers activations. Note that the entire sequence is still being fed to the model we are just visualizing a prediction trajectory for this particular slice (`[14:21]`) of the activations.

[5]:
```
import plotly.io as pio
pio.renderers.default = "sphinx_gallery" # Remove this if you are not seeing the plots
```

[6]:
```
from plotly.subplots import make_subplots

fig = make_subplots(
    rows=4,
    cols=1,
    shared_xaxes=True,
    vertical_spacing=0.03,
    subplot_titles=("Entropy", "Forward KL", "Cross Entropy", "Max Probability"),
)

fig.add_trace(
    predictition_traj_ring.entropy().heatmap(
        colorbar_y=0.89, colorbar_len=0.25, textfont={'size':10}
    ),
    row=1, col=1
)

fig.add_trace(
    predictition_traj_ring.forward_kl().heatmap(
        colorbar_y=0.63, colorbar_len=0.25, textfont={'size':10}
    ),
```

(continues on next page)

(continued from previous page)

```
    row=2, col=1
)

fig.add_trace(
    predictition_traj_ring.cross_entropy().heatmap(
        colorbar_y=0.37, colorbar_len=0.25, textfont={'size':10}
    ),
    row=3, col=1
)

fig.add_trace(
    predictition_traj_ring.max_probability().heatmap(
        colorbar_y=0.11, colorbar_len=0.25, textfont={'size':10}
    ),
    row=4, col=1
)

fig.update_layout(height=800, width=500, title_text="Tolkien's Tokens on visualized with␣
↪the Tuned Lens")
fig.show()
```

Now let's look at the prediction trajectory for our modified sequence.

[7]:
```
predictition_traj_model = PredictionTrajectory.from_lens_and_model(
    tuned_lens,
    model,
    tokenizer=tokenizer,
    input_ids=input_ids_model,
    targets=targets_model,
).slice_sequence(third_line)
```

[8]:
```
from plotly.subplots import make_subplots
import plotly.graph_objects as go

fig = make_subplots(
    rows=4,
    cols=1,
    shared_xaxes=True,
    vertical_spacing=0.03,
    subplot_titles=("Entropy", "Forward KL", "Cross Entropy", "Max Probability"),
)

fig.add_trace(
    predictition_traj_model.entropy().heatmap(
        colorbar_y=0.89, colorbar_len=0.25, textfont={'size':10}
    ),
    row=1, col=1
)

fig.add_trace(
    predictition_traj_model.forward_kl().heatmap(
```

(continues on next page)

(continued from previous page)

```
        colorbar_y=0.63, colorbar_len=0.25, textfont={'size':10}
    ),
    row=2, col=1
)

fig.add_trace(
    predictition_traj_model.cross_entropy().heatmap(
        colorbar_y=0.37, colorbar_len=0.25, textfont={'size':10}
    ),
    row=3, col=1
)

fig.add_trace(
    predictition_traj_model.max_probability().heatmap(
        colorbar_y=0.11, colorbar_len=0.25, textfont={'size':10}
    ),
    row=4, col=1
)

fig.update_layout(height=800, width=500, title_text="Tolkien's Tokens on visualized with␣
↪the Tuned Lens")
fig.show()
```

Now for the fun part, lets use the tools provided by the tuned lens to observe the changes between the original trajectory with `_Ring` and our modified trajectory where we introduced the token `_Model`.

[9]:
```
fig = predictition_traj_ring.total_variation(predictition_traj_model, min_prob_delta=0.
↪05).figure("Total Variation")
fig.show()
```

This seems to stongly sugest that there is an induction head at layer 9 within this model.

5.8 Combing the Tuned Lens and the Transformer Lens

The TransformerLens is an another open source package designed to provide a standard interface for investigating the internals of transformer models. Integrating with TransformerLens and tuned-lens allows you to observe how model edits effect the prediction trajectories, and make use of all of the visualizations provided by the tuned lens package. This is primarily useful for preliminary investigations of circuits. Note this tutorial will be very hard to follow unless you are already familiar with the TransformerLens package.

To demonstrate this we will investigate the greater-than circuit in gpt2-small.

[1]:
```
import torch as th
from tuned_lens.plotting import PredictionTrajectory
from tuned_lens.nn import TunedLens, Unembed, LogitLens
import transformer_lens as tl

model = tl.HookedTransformer.from_pretrained(
    "gpt2",
    device="cpu",
```

(continues on next page)

(continued from previous page)

```
    fold_ln=False, # The tuned lens applies the final layer norm so we should not fold
    # this into the unembed operation.
)
assert model.tokenizer is not None


tuned_lens = TunedLens.from_unembed_and_pretrained(
    unembed=Unembed(model),
    lens_resource_id="gpt2",
)

logit_lens = LogitLens.from_model(model)

def to_targets(input_ids: th.Tensor):
    return th.cat(
        (input_ids[..., 1:], th.full(input_ids.shape[:-1] + (1,), model.tokenizer.eos_
↪token_id)
    ), dim=-1)
```

```
Using pad_token, but it is not set yet.
```

```
Loaded pretrained model gpt2 into HookedTransformer
```

[2]:
```
import plotly.io as pio
pio.renderers.default = "sphinx_gallery" # REMOVE THIS IF YOU ARE NOT SEEING PLOTS
```

[3]:
```
model.generate(" The war lasted from 1754 to 17", max_new_tokens=2, do_sample=True)
```

```
  0%|          | 0/2 [00:00<?, ?it/s]
```

[3]:
```
' The war lasted from 1754 to 1776 and'
```

[4]:
```
str_tokens = model.to_str_tokens(" The war lasted from")

dates = [[12, 21],
         [11, 23],
         [10, 24],
         [16, 89],
         [17, 54],
         [14, 47],
         [15, 36],
         [17, 32],
         [18, 21],
         [11, 57]]

input_ids_strs = [str_tokens + [" " + str(data[0]),  str(data[1]), " to", " " +␣
↪str(data[0])] for data in dates]
input_ids = th.tensor([[model.to_single_token(s) for s in arr] for arr in input_ids_
↪strs])

scrub_ids_strs = [str_tokens + [" " + str(data[0]),  "01", " to", " " + str(data[0])]␣
↪for data in dates]
scrub_ids = th.tensor([[model.to_single_token(s) for s in arr] for arr in scrub_ids_
```

(continues on next page)

(continued from previous page)

```
↪strs])

targets_strs = [str_tokens[1:] + [" " + str(data[0]),  str(data[1]), " to", " " +␣
↪str(data[0]), str(data[1] + 3)] for data in dates]
targets = th.tensor([[model.to_single_token(s) for s in arr] for arr in targets_strs])
anti_targets_strs = [str_tokens[1:] + [" " + str(data[0]),  str(data[1]), " to", " " +␣
↪str(data[0]), str(data[1] - 3)] for data in dates]
anti_targets = th.tensor([[model.to_single_token(s) for s in arr] for arr in anti_
↪targets_strs])
```

[5]:
```
from pprint import pprint

log_prob_range = (-2, 2) # The range of log probabilities to plot
# this makes the different plots comparable.

print("Scrubbed:")
pprint(scrub_ids_strs[:2], width=120)
print("Input:")
pprint(input_ids_strs[:2], width=120)
print("Targets:")
pprint(targets_strs[:2])
print("Anti targets:")
pprint(anti_targets_strs[:2])

with th.inference_mode():
    logits, cache = model.run_with_cache(
        input=input_ids, return_type="logits"
    )

    pred_traj_clean = PredictionTrajectory.from_lens_and_cache(
        lens=tuned_lens,
        cache=cache,
        model_logits=logits,
        input_ids=input_ids,
        targets=targets,
        anti_targets=anti_targets,
    )

    pred_traj_clean_logit = PredictionTrajectory.from_lens_and_cache(
        lens=logit_lens,
        cache=cache,
        model_logits=logits,
        input_ids=input_ids,
        targets=targets,
        anti_targets=anti_targets,
    )
```

```
Scrubbed:
[['<|endoftext|>', ' The', ' war', ' lasted', ' from', ' 12', '01', ' to', ' 12'],
 ['<|endoftext|>', ' The', ' war', ' lasted', ' from', ' 11', '01', ' to', ' 11']]
Input:
[['<|endoftext|>', ' The', ' war', ' lasted', ' from', ' 12', '21', ' to', ' 12'],
```

(continues on next page)

(continued from previous page)

```
 ['<|endoftext|>', ' The', ' war', ' lasted', ' from', ' 11', '23', ' to', ' 11']]
Targets:
[[' The', ' war', ' lasted', ' from', ' 12', '21', ' to', ' 12', '24'],
 [' The', ' war', ' lasted', ' from', ' 11', '23', ' to', ' 11', '26']]
Anti targets:
[[' The', ' war', ' lasted', ' from', ' 12', '21', ' to', ' 12', '18'],
 [' The', ' war', ' lasted', ' from', ' 11', '23', ' to', ' 11', '20']]
```

[6]:
```
pred_traj_clean.slice_sequence(slice(-5, None)).log_prob_diff(delta=True).clip(*log_prob_
↪range).figure(title="Effects of each layer on the target/anti-target ratio")
```

[7]:
```
pred_traj_clean_logit.slice_sequence(slice(-5, None)).log_prob_diff(delta=True).
↪clip(*log_prob_range).figure(title="Same as above but with the logit lens")
```

The above results mostly agree with the results in the paper. We see that the majority of the contributions to the correct logits do indeed come from the layers 8 through 11. Interestingly, it seems like layer 11, in particular the MLP, is actually acting as a regularizer and reducing the confidence in the target prediction.

5.8.1 Lets start removing components!

Now we will show how to ablate the components of this circuit. I recommend, opening up this in google colab and playing with it!

Here are some exercises you can try: * What are the effects of ablating MLP on layer 8? * What happens when we remove MLP 11? * How much collateral damage does this cause? * How do the different types of ablation work? * Why might we prefer a swap ablation to a zero ablation hint. * CHALLENGE: What edit would you make to the model that disrupts the greater-than circuit but that has minimal effects on the models behavior?

[8]:
```
model.cfg.use_attn_result = True
scrubed_logits, scrubed_cache = model.run_with_cache(
    input=scrub_ids, return_type="logits"
)
model.cfg.use_attn_result = False
```

[9]:
```
import transformer_lens.utils as utils
from functools import partial


def zero_ablation_hook(result: th.Tensor, hook: tl.hook_points.HookPoint) -> th.Tensor:
    result[:] = 0
    return result

def swap_ablation_hook(result: th.Tensor, hook: tl.hook_points.HookPoint) -> th.Tensor:
    result[:] = scrubed_cache[hook.name]
    return result


MLPS_TO_ABLATE = [9]
mlp_hooks = [(utils.get_act_name("mlp_out", layer), zero_ablation_hook) for layer in␣
↪MLPS_TO_ABLATE]
```

(continues on next page)

(continued from previous page)

```
ATTN_MODULES_TO_ABLATE = []
attn_hooks = [(utils.get_act_name("result", layer), zero_ablation_hook) for layer in␣
↪ATTN_MODULES_TO_ABLATE]

with model.hooks(fwd_hooks=(mlp_hooks + attn_hooks)), th.inference_mode():
    model.cfg.use_attn_result = True
    logits, cache = model.run_with_cache(
        input=input_ids, return_type="logits"
    )
    model.cfg.use_attn_result = False

    pred_traj_ablated = PredictionTrajectory.from_lens_and_cache(
        lens=tuned_lens,
        cache=cache,
        model_logits=logits,
        input_ids=input_ids,
        targets=targets,
        anti_targets=anti_targets,
    )

    pred_traj_ablated_logit = PredictionTrajectory.from_lens_and_cache(
        lens=logit_lens,
        cache=cache,
        model_logits=logits,
        input_ids=input_ids,
        targets=targets,
        anti_targets=anti_targets,
    )
```

[10]:
```
pred_traj_ablated.slice_sequence(slice(-5, None)).log_prob_diff(delta=True).clip(*log_
↪prob_range).figure(title="Effects of each layer on the target/anti-target ratio after␣
↪ablation")
```

[11]:
```
pred_traj_ablated_logit.log_prob_diff(delta=True).clip(*log_prob_range).figure()
```

5.8.2 How much collateral damage does ablating the above components cause?

An interesting question we can answerer with the tuned lens is what other capabilities have our edits effected?

Note the code bellow assumes you are using zero ablation.

[12]:
```
with model.hooks(fwd_hooks=(mlp_hooks + attn_hooks)), th.inference_mode():
    model.cfg.use_attn_result = True
    control_text = model.generate(" Numbers I love them!", max_new_tokens=10, do_
↪sample=False)
    model.cfg.use_attn_result = False

input_ids_control = model.to_tokens(control_text)
targets_control = to_targets(input_ids_control)
```

(continues on next page)

(continued from previous page)

```
logits, cache = model.run_with_cache(
    input=input_ids_control, return_type="logits"
)

pred_traj_control_clean = PredictionTrajectory.from_lens_and_cache(
    lens=tuned_lens,
    cache=cache,
    model_logits=logits,
    input_ids=input_ids_control,
    targets=targets_control,
)

with model.hooks(fwd_hooks=(mlp_hooks + attn_hooks)), th.inference_mode():
    model.cfg.use_attn_result = True
    logits, cache = model.run_with_cache(
        input=input_ids_control, return_type="logits"
    )
    model.cfg.use_attn_result = False

    pred_traj_control_ablated = PredictionTrajectory.from_lens_and_cache(
        lens=tuned_lens,
        cache=cache,
        model_logits=logits,
        input_ids=input_ids_control,
        targets=targets_control,
    )

pred_traj_control_ablated.kl_divergence(pred_traj_control_clean).figure()
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

5.9 Maintainers Guide

Here are some notes on how to maintain this package mostly focusing on the CI/CD workflow build on top of GitHub actions.

5.9.1 The pull request checks

The majority of the pull request checks are specified in the CI. Specifically, the pre-merge.yaml workflow. There are 4 major components to this workflow:

1. Ensuring that the pre-commit checks configured in `.pre-commit-config.yaml` pass.
2. **Ensuring that the package builds correctly and the `pytest` tests pass on python versions 3.9 - 3.11.**
    - `pytest` is configured in the `pyproject.toml`.
3. **Ensuring that this documentation builds correctly and the code within it runs including the tutorial notebooks.**
    - The documentation is built using sphinx and the tutorial notebooks are run using nbsphinx.

4. **Ensuring that the docker image builds correctly and uploading code coverage reports to codecov.**
    - The code coverage requirements themselves are contained in `.codecov.yml`. Importantly, the code coverage bot itself enforce these requirements, not the CI.

Note that the pre-merge workflow also runs on every push to the main branch. To make sure this passes is best practice to merge main into the branch before merging your PR.

5.9.2 Publishing versions

Publishing new versions is mostly handled by the CI here are the steps to follow to build and publish a new version:

1. **To create a release first update the version in the `pyproject.toml` then commit and push a tag of the form `v<PEP440 Version>`. When making a new release it's a good idea to start with a pre-release version e.g. `v0.0.5a0`.**
    - For more information on versioning see PEP440.
2. **This will start the pre-release.yaml workflow if it succeeds this will automatically create a draft release in GitHub and publish the package to test PyPI.**
    - The specifically the pre-release workflow validates that the tag matches the version in the `pyproject.toml`, and runs a very basic smoke test on the CI. It most of the heavy lifting is done by the pre-merge.yaml workflow.
3. If you are happy with every thing, simply edit the newly created draft adding release notes etc and press the release button. This will run the *publish.yaml <https://github.com/AlignmentResearch/tuned-lens/blob/improved-docs-85/.github/workflows/publish.yml>* workflow which publishes the package to PyPI and uploads the docker image the GitHub package registry are synchronized.
4. Note that if ref is **not** tagged as a pre-release version e.g. `v0.0.5`, then pushing the tag should also automatically build the docs on read the docs.

PYTHON MODULE INDEX

t

INDEX

U

V