transformer_lens.HookedTransformer

Hooked Transformer.

The Hooked Transformer is the core part of TransformerLens.

In common PyTorch model implementations (e.g. ones from HuggingFace) it’s fairly easy to extract model weights, but much harder to extract activations. TransformerLens aims to simplify this task by attaching hooks to every notable activation within the model. This enables the

inspection and/or alteration of activations in individual components like attention heads and MLP layers, facilitating a deeper understanding of the internal workings of transformers like GPT-2.

```
class transformer_lens.HookedTransformer.HookedTransformer(cfg:
    Union[HookedTransformerConfig, Dict], tokenizer:
    Optional[PreTrainedTokenizerBase] = None, move_to_device: bool = True,
    default_padding_side: Literal['left', 'right'] = 'right')
```

Bases: `HookedRootModule`

Hooked Transformer.

Implements a full Transformer using the components here, with a `transformer_lens.hook_points.HookPoint` on every interesting activation.

TransformerLens comes loaded with >50 GPT-style models. Typically you initialise it with one of these via `from_pretrained()`, although it can also be instantiated with randomly initialized weights via `__init__()`.

Once you've initialized the model, a common next step is to test it can do the task you're investigating. This can be done with `transformer_lens.utils.test_prompt()`.

property OV

property QK

property W_E: Float[Tensor, 'd_vocab d_model']

Convenience to get the embedding matrix.

property W_E_pos: Float[Tensor, 'd_vocab+n_ctx d_model']

Concatenated W_E and W_pos.

Used as a full (overcomplete) basis of the input space, useful for full QK and full OV circuits.

property W_K: Float[Tensor, 'n_layers n_heads d_model d_head']

Stack the key weights across all layers.

property W_O: Float[Tensor, 'n_layers n_heads d_head d_model']

Stack the attn output weights across all layers.

property W_Q: Float[Tensor, 'n_layers n_heads d_model d_head']

Stack the query weights across all layers.

property W_U: Float[Tensor, 'd_model d_vocab']

Convenience to get the unembedding matrix.

I.e. the linear map from the final residual stream to the output logits).

property W_V: Float[Tensor, 'n_layers n_heads d_model d_head']

Stack the value weights across all layers.

property W_gate: Optional[Float[Tensor, 'n_layers d_model d_mlp']]

Stack the MLP gate weights across all layers.

Only works for models with gated MLPs.

property W_in: Float[Tensor, 'n_layers d_model d_mlp']

Stack the MLP input weights across all layers.

property **W_out**: Float[Tensor, 'n_layers d_mlp d_model']

Stack the MLP output weights across all layers.

property **W_pos**: Float[Tensor, 'n_ctx d_model']

Convenience function to get the positional embedding.

Only works on models with absolute positional embeddings!

__init__(cfg: Union[HookedTransformerConfig, Dict], tokenizer: Optional[PreTrainedTokenizerBase] = None, move_to_device: bool = True, default_padding_side: Literal['left', 'right'] = 'right')

Model initialization.

Note that if you want to load the model from pretrained weights, you should use `from_pretrained()` instead.

PARAMETERS:

- **cfg** – The config to use for the model.
- **tokenizer** – The tokenizer to use for the model. If not provided, it is inferred from *cfg.tokenizer_name* or initialized to *None*. If *None*, then the model cannot be passed strings, and d_vocab must be explicitly set.
- **move_to_device** – Whether to move the model to the device specified in cfg. device. Must be true if *n_devices* in the config is greater than 1, since the model's layers will be split across multiple devices.
- **default_padding_side** – Which side to pad on.

accumulated_bias(layer: int, mlp_input: bool = False, include_mlp_biases=True) → Float[Tensor, 'd_model']

Accumulated Bias.

Returns the accumulated bias from all layer outputs (ie the b_Os and b_outs), up to the input of layer L.

PARAMETERS:

- **layer** (*int*) – Layer number, in [0, n_layers]. layer==0 means no layers, layer==n_layers means all layers.
- **mlp_input** (*bool*) – If True, we take the bias up to the input of the MLP of layer L (ie we include the bias from the attention output of the current layer, otherwise just biases from previous layers)
- **include_mlp_biases** (*bool*) – Whether to include the biases of MLP layers. Often useful to have as False if we're expanding attn_out into individual heads, but keeping mlp_out as is.

RETURNS:

[d_model], accumulated bias

RETURN TYPE:

bias (torch.Tensor)

all_composition_scores(mode) → Float[Tensor, 'n_layers n_heads n_layers n_heads']

All Composition Scores.

Returns the Composition scores for all pairs of heads, as a L1, H1, L2, H2 tensor (which is upper triangular on the first and third axes).

See https://transformer-circuits.pub/2021/framework/index.html#:~:text=The%20above%20diagram%20shows%20Q% for three metrics used.

PARAMETERS:

mode (*str*) – One of ["Q", "K", "V"], the mode to use for the composition score.

all_head_labels()

Returns a list of all head names in the model.

property b_K: Float[Tensor, 'n_layers n_heads d_head']

Stack the key biases across all layers.

property b_O: Float[Tensor, 'n_layers d_model']

Stack the attn output biases across all layers.

property b_Q: Float[Tensor, 'n_layers n_heads d_head']

Stack the query biases across all layers.

property b_U: Float[Tensor, 'd_vocab']

property b_V: Float[Tensor, 'n_layers n_heads d_head']

Stack the value biases across all layers.

property **b_in**: Float[Tensor, 'n_layers d_mlp']

Stack the MLP input biases across all layers.

property **b_out**: Float[Tensor, 'n_layers d_model']

Stack the MLP output biases across all layers.

center_unembed(state_dict: Dict[str, Tensor])

Center the unembedding weights W_U.

This is done by subtracting the mean of the weights from the weights themselves. This is done in-place. As softmax is translation invariant, this changes the logits but not the log probs, and makes the model logits (slightly) more interpretable - when trying to understand how components contribute to the logits, we'll be less misled by components that just add something to every logit.

center_writing_weights(state_dict: Dict[str, Tensor])

Center Writing Weights.

Centers the weights of the model that write to the residual stream - W_out, W_E, W_pos and W_out. This is done by subtracting the mean of the weights from the weights themselves. This is done in-place. See fold_layer_norm for more details.

check_hooks_to_add(hook_point, hook_point_name, hook, dir='fwd', is_permanent=False, prepend=False) → None

Override this function to add checks on which hooks should be added

cpu()

Wrapper around cuda that also changes *self.cfg.device*.

cuda()

Wrapper around cuda that also changes *self.cfg.device*.

fold_layer_norm(state_dict: Dict[str, Tensor], fold_biases=True, center_weights=True)

Fold Layer Norm. Can also be used to fold RMS Norm, when fold_biases and center_weights are set to False.

Takes in a state dict from a pretrained model, formatted to be consistent with HookedTransformer but with LayerNorm weights and biases. Folds these into the neighbouring weights. See further_comments.md for more details.

PARAMETERS:

- **state_dict** (*Dict[str, torch.Tensor]*) – State dict of pretrained model.
- **fold_biases** (*bool*) – Enables folding of LN biases. Should be disabled when RMS Norm is used.
- **center_weights** (*bool*) – Enables the centering of weights after folding in LN. Should be disabled when RMS Norm is used.

```
fold_value_biases(state_dict: Dict[str, Tensor])
```

Fold the value biases into the output bias.

Because attention patterns add up to 1, the value biases always have a constant effect on a head's output. Further, as the outputs of each head in a layer add together, each head's value bias has a constant effect on the *layer's* output, which can make it harder to interpret the effect of any given head, and it doesn't matter which head a bias is associated with. We can factor this all into a single output bias to the layer, and make it easier to interpret the head's output. Formally, we take b_O_new = b_O_original + sum_head(b_V_head @ W_O_head).

```
forward(input, return_type: Literal['logits'], loss_per_token: bool = False,
    prepend_bos: Optional[bool] = USE_DEFAULT_VALUE, padding_side:
    Optional[Literal['left', 'right']] = USE_DEFAULT_VALUE, start_at_layer:
    Optional[int] = None, tokens: Optional[Int[Tensor, 'batch pos']] = None,
    shortformer_pos_embed: Optional[Float[Tensor, 'batch pos d_model']] =
    None, attention_mask: Optional[Tensor] = None, stop_at_layer:
    Optional[int] = None, past_kv_cache:
    Optional[HookedTransformerKeyValueCache] = None) → Union[Float[Tensor,
    ''], Float[Tensor, 'batch pos-1']]
forward(input, return_type: Literal['loss'], loss_per_token: bool = False,
    prepend_bos: Optional[bool] = USE_DEFAULT_VALUE, padding_side:
    Optional[Literal['left', 'right']] = USE_DEFAULT_VALUE, start_at_layer:
    Optional[int] = None, tokens: Optional[Int[Tensor, 'batch pos']] = None,
    shortformer_pos_embed: Optional[Float[Tensor, 'batch pos d_model']] =
    None, attention_mask: Optional[Tensor] = None, stop_at_layer:
    Optional[int] = None, past_kv_cache:
    Optional[HookedTransformerKeyValueCache] = None) → Union[Float[Tensor,
    ''], Float[Tensor, 'batch pos-1']]
forward(input, return_type: Literal['both'], loss_per_token: bool = False,
    prepend_bos: Optional[bool] = USE_DEFAULT_VALUE, padding_side:
    Optional[Literal['left', 'right']] = USE_DEFAULT_VALUE, start_at_layer:
    Optional[int] = None, tokens: Optional[Int[Tensor, 'batch pos']] = None,
    shortformer_pos_embed: Optional[Float[Tensor, 'batch pos d_model']] =
    None, attention_mask: Optional[Tensor] = None, stop_at_layer:
    Optional[int] = None, past_kv_cache:
    Optional[HookedTransformerKeyValueCache] = None) → Tuple[Float[Tensor,
    'batch pos d_vocab'], Union[Float[Tensor, ''], Float[Tensor, 'batch pos-
    1']]]
forward(input, return_type: Literal[None], loss_per_token: bool = False,
    prepend_bos: Optional[bool] = USE_DEFAULT_VALUE, padding_side:
    Optional[Literal['left', 'right']] = USE_DEFAULT_VALUE, start_at_layer:
    Optional[int] = None, tokens: Optional[Int[Tensor, 'batch pos']] = None,
    shortformer_pos_embed: Optional[Float[Tensor, 'batch pos d_model']] =
    None, attention_mask: Optional[Tensor] = None, stop_at_layer:
    Optional[int] = None, past_kv_cache:
    Optional[HookedTransformerKeyValueCache] = None) → None
```

Forward Pass.

Input is either a batch of tokens ([batch, pos]) or a text string, a string is automatically tokenized to a batch of a single element. The prepend_bos flag only applies when inputting a text string.

Note that loss is the standard “predict the next token” cross-entropy loss for GPT-2 style language models - if you want a custom loss function, the recommended

behaviour is returning the logits and then applying your custom loss function.

PARAMETERS:

- **Optional[str]** (*return_type*) – The type of output to return. Can be one of: None (return nothing, don't calculate logits), 'logits' (return logits), 'loss' (return cross-entropy loss), 'both' (return logits and loss).
- **bool** (*loss_per_token*) – Whether to return the (next token prediction) loss per token (True) or average (False). Average loss is a scalar (averaged over position *and* batch), per-token loss is a tensor ([batch, position-1]) - position-1 because we're predicting the next token, and there's no specified next token for the final token. Defaults to False.
- **Optional[bool]** (*prepend_bos*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (only applies when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos which is set to True unless specified otherwise. (Even for models not explicitly trained with a prepended BOS token, heads often use the first position as a resting position and accordingly lose information from the first token, so this empirically seems to give better results.) Pass True or False to locally override the default.
- **Optional[Literal["left"** (*padding_side*) – Overrides self.tokenizer.padding_side. Specifies which side to pad on when tokenizing multiple strings of different lengths.
- **"right"]]** – Overrides self.tokenizer.padding_side. Specifies which side to pad on when tokenizing multiple strings of different lengths.
- **Optional[int]** (*stop_at_layer*) – If not None, start the forward pass at the specified layer. Requires input to be the residual stream before the specified layer with shape [batch, pos, d_model]. Inclusive - ie, start_at_layer = 0 skips the embedding then runs the rest of the model. Supports negative indexing. start_at_layer = -1 only runs the final block and the unembedding. Defaults to None (run the full model).
- **tokens** – Optional[Int[torch.Tensor, "batch pos"]]: Tokenized input. Only use if start_at_layer is not None and return type is "loss" or "both".
- **shortformer_pos_embed** – Optional[Float[torch.Tensor, "batch pos d_model"]]: Positional embedding for shortformer models. Only use if start_at_layer is not None and self.cfg.positional_embedding_type == "shortformer".
- **attention_mask** – Optional[torch.Tensor]: Override the attention mask used to ignore padded tokens. If start_at_layer is not None and (self.tokenizer.padding_side == "left" or past_kv_cache is not None), this should be passed as the attention mask is not computed automatically. Defaults to None.
- **Optional[int]** – If not None, stop the forward pass at the specified layer. Exclusive - ie, stop_at_layer = 0 will only run the embedding layer, stop_at_layer = 1 will run the embedding layer and the first transformer block, etc. Supports negative indexing. Useful for analysis of intermediate layers, eg finding neuron activations in layer 3 of a 24 layer model. Defaults to None (run the full model). If not None, we return the last residual stream computed.
- **Optional[HookedTransformerKeyValueCache]** (*past_kv_cache*) – If not None, keys and values will be stored for every attention head (unless the cache is frozen).

If there are keys and values already in the cache, these will be prepended to the keys and values for the new input, so that the new tokens can pay attention to previous tokens. This is useful for generating text, because we don't need to repeat computation for tokens that have already been through the model. Also caches attention_mask so previous tokens are masked correctly (unless frozen). Padding should be ignored in all cases, so it's okay to eg. pass in left padded tokens twice in a row. Warning: Don't accidentally prepend_bos to the second half of a prompt. Defaults to None (don't use caching).

```
classmethod from_pretrained(model_name: str, fold_ln: bool = True,
    center_writing_weights: bool = True, center_unembed: bool = True,
    refactor_factored_attn_matrices: bool = False, checkpoint_index:
    Optional[int] = None, checkpoint_value: Optional[int] = None, hf_model:
    Optional[AutoModelForCausalLM] = None, device: Optional[Union[str,
    device]] = None, n_devices: int = 1, tokenizer:
    Optional[PreTrainedTokenizerBase] = None, move_to_device: bool = True,
    fold_value_biases: bool = True, default_prepend_bos: Optional[bool] =
    None, default_padding_side: Literal['left', 'right'] = 'right',
    dtype='float32', first_n_layers: Optional[int] = None,
    **from_pretrained_kwargs) → T
```

Load in a Pretrained Model.

Load in pretrained model weights to the HookedTransformer format and optionally to do some processing to make the model easier to interpret. Currently supports loading from most autoregressive HuggingFace models (`gpt2`, `neo`, `gptj`, `opt`...) and from a range of toy models and SoLU models trained by Neel Nanda. The full list is available in the docs under model properties. Also supports loading from a checkpoint for checkpointed models (currently, models trained by NeelNanda and the stanford-crfm models (using parameters `checkpoint_index` and `checkpoint_value`).

See `load_and_process_state_dict()` for details on the processing (folding layer norm, centering the unembedding and centering the writing weights).

Example:

```
>>> from transformer_lens import HookedTransformer
>>> model = HookedTransformer.from_pretrained("tiny-stories-1M")
Loaded pretrained model tiny-stories-1M into HookedTransformer
```

PARAMETERS:

- **model_name** – The model name - must be an element of `transformer_lens.loading_from_pretrained.OFFICIAL_MODEL_NAMES` or an alias of one. The full list of available models can be found in the docs under model properties.
- **fold_ln** –

  Whether to fold in the LayerNorm weights to the subsequent linear layer. This does not change the computation.

  LayerNorm is a common regularization technique used in transformers. Unlike BatchNorm, it cannot be turned off at inference time, as it significantly alters the mathematical function implemented by the transformer.

  When *fold_ln* is set to True, LayerNorm (with weights w_{ln} and b_{ln}) followed by a linear layer ($W + b$) is optimized to LayerNormPre (just centering & normalizing) followed by a new linear layer with $W_{eff} = w[:, extNone] * W$ (element-wise multiplication) and $b_{eff} = b + b_{ln}@W$. This transformation is computationally equivalent and simplifies the model's interpretability. It essentially merges LayerNorm weights into the subsequent linear layer's weights, which is handled by HookedTransformer when loading pre-trained weights. Set *fold_ln* to False when loading a state dict if you wish to turn this off.

  Mathematically, LayerNorm is defined as follows:

$$
\begin{aligned}
x_1 &= x_0 - \text{mean}(x_0) \\
x_2 &= \frac{x_1}{\sqrt{\text{mean}(x_1^2)}} \\
x_3 &= x_2 \cdot w \\
x_4 &= x_3 + b
\end{aligned}
$$

  For further details, refer to this document.

- **center_writing_weights** –

  Whether to center weights writing to the residual stream (ie set mean to be zero). Due to LayerNorm this doesn't change the computation.

  A related idea to folding layernorm (`fold_ln`) - *every* component reading an input from the residual stream is preceded by a LayerNorm, which means that the mean of a residual stream vector (ie the component in the direction of all ones) never matters. This means we can remove the all ones component of weights and biases whose output *writes* to the residual stream. Mathematically, `W_writing -= W_writing.mean(dim=1, keepdim=True)`.

- **center_unembed** –

Whether to center W_U (ie set mean to be zero). Softmax is translation invariant so this doesn't affect log probs or loss, but does change logits.

The logits are fed into a softmax. Softmax is translation invariant (eg, adding 1 to every logit doesn't change the output), so we can simplify things by setting the mean of the logits to be zero. This is equivalent to setting the mean of every output vector of `W_U` to zero. In code, `W_U -= W_U.mean(dim=-1, keepdim=True)`.

- **refactor_factored_attn_matrices** – Whether to convert the factored matrices (W_Q & W_K, and W_O & W_V) to be "even". Defaults to False
- **checkpoint_index** – If loading from a checkpoint, the index of the checkpoint to load.
- **checkpoint_value** – If loading from a checkpoint, the value of the checkpoint to load, ie the step or token number (each model has checkpoints labelled with exactly one of these). E.g. `1000` for a checkpoint taken at step 1000 or after 1000 tokens. If *checkpoint_index* is also specified, this will be ignored.
- **hf_model** – If you have already loaded in the HuggingFace model, you can pass it in here rather than needing to recreate the object. Defaults to None.
- **device** – The device to load the model onto. By default will load to CUDA if available, else CPU.
- **n_devices** – The number of devices to split the model across. Defaults to 1. If greater than 1, *device* must be cuda.
- **tokenizer** – The tokenizer to use for the model. If not provided, it is inferred from cfg.tokenizer_name or initialized to None. If None, then the model cannot be passed strings, and d_vocab must be explicitly set.
- **move_to_device** – Whether to move the model to the device specified in cfg. device. Must be true if *n_devices* in the config is greater than 1, since the model's layers will be split across multiple devices.
- **fold_value_biases** –

  Each attention head has a value bias. Values are averaged to create mixed values (`z`), weighted by the attention pattern, but as the bias is constant, its contribution to `z` is exactly the same. The output of a head is `z @ W_O`, and so the value bias just linearly adds to the output of the head. This means that the value bias of a head has nothing to do with the head, and is just a constant added to the attention layer outputs. We can take the sum across these and b_O to get an "effective bias" for the layer. In code, we set `b_V=0`. and `b_O = (b_V @ W_O).sum(dim=0) + b_O`.

  The technical derivation of this is as follows. `v = residual @ W_V[h] + broadcast_b_V[h]` for each head `h` (where `b_V` is broadcast up from shape `d_head` to shape `[position, d_head]`). And `z = pattern[h] @ v = pattern[h] @ residual @ W_V[h] + pattern[h] @ broadcast_b_V[h]`. Because `pattern[h]` is `[destination_position, source_position]` and `broadcast_b_V` is constant along the `(source_)position` dimension, we're basically just multiplying it by the sum of the pattern across the `source_position` dimension, which is just `1`. So it remains exactly the same, and so is just broadcast across the destination positions.

- **default_prepend_bos** –

Default behavior of whether to prepend the BOS token when the methods of HookedTransformer process input text to tokenize (only when input is a string). Resolution order for default_prepend_bos: 1. If user passes value explicitly, use that value 2. Model-specific default from cfg_dict if it exists (e.g. for bloom models it's False) 3. Global default (True)

Even for models not explicitly trained with the BOS token, heads often use the first position as a resting position and accordingly lose information from the first token, so this empirically seems to give better results. Note that you can also locally override the default behavior by passing in prepend_bos=True/False when you call a method that processes the input string.

- **from_pretrained_kwargs** – Any other optional argument passed to HuggingFace's from_pretrained (e.g. "cache_dir" or "torch_dtype"). Also passed to other HuggingFace functions when compatible. For some models or arguments it doesn't work, especially for models that are not internally loaded with HuggingFace's from_pretrained (e.g. SoLU models).
- **dtype** – What data type to load the model in (also sets the dtype of the HuggingFace model). Set to bfloat16 or float16 if you get out of memory errors when loading the model.
- **default_padding_side** – Which side to pad on when tokenizing. Defaults to "right".
- **first_n_layers** – If specified, only load the first n layers of the model.

```
classmethod from_pretrained_no_processing(model_name: str, fold_ln=False,
    center_writing_weights=False, center_unembed=False,
    refactor_factored_attn_matrices=False, fold_value_biases=False,
    dtype=torch.float32, default_prepend_bos=None,
    default_padding_side='right', **from_pretrained_kwargs)
```

Wrapper for from_pretrained.

Wrapper for from_pretrained with all boolean flags related to simplifying the model set to False. Refer to from_pretrained for details.

```
generate(input: Union[str, List[str], Int[Tensor, 'batch pos'], Float[Tensor,
    'batch pos hidden_size']] = '', max_new_tokens: int = 10, stop_at_eos:
    bool = True, eos_token_id: Optional[int] = None, do_sample: bool = True,
    top_k: Optional[int] = None, top_p: Optional[float] = None, temperature:
    float = 1.0, freq_penalty: float = 0.0, use_past_kv_cache: bool = True,
    prepend_bos: Optional[bool] = None, padding_side: Optional[Literal['left',
    'right']] = None, return_type: Optional[str] = 'input', verbose: bool =
    True) → Union[str, List[str], Int[Tensor, 'batch pos_plus_new_tokens'],
    Float[Tensor, 'batch pos_plus_new_tokens hidden_size']]
```

Sample Tokens from the Model.

Sample tokens from the model until the model outputs eos_token or max_new_tokens is reached.

To avoid fiddling with ragged tensors, if we input a batch of text and some sequences finish (by producing an EOT token), we keep running the model on the entire batch, but

throw away the output for a finished sequence and just keep adding EOTs to pad.

PARAMETERS:

- **input** (*Union[str, List[str], Int[torch.Tensor, "batch pos"], Float[torch.Tensor, "batch pos hidden_size"]]*) – A text string (this will be converted to a batch of tokens with batch size 1), a list of strings, batch of tokens or a tensor of precomputed embeddings of shape [batch, pos, hidden_size].
- **max_new_tokens** (*int*) – Maximum number of tokens to generate.
- **stop_at_eos** (*bool*) – If True, stop generating tokens when the model outputs eos_token.
- **eos_token_id** (*Optional[Union[int, Sequence]]*) – The token ID to use for end of sentence. If None, use the tokenizer's eos_token_id - required if using stop_at_eos. It's also possible to provide a list of token IDs (not just the eos_token_id), in which case the generation will stop when any of them are output (useful e.g. for stable_lm).
- **do_sample** (*bool*) – If True, sample from the model's output distribution. Otherwise, use greedy search (take the max logit each time).
- **top_k** (*int*) – Number of tokens to sample from. If None, sample from all tokens.
- **top_p** (*float*) – Probability mass to sample from. If 1.0, sample from all tokens. If <1.0, we take the top tokens with cumulative probability >= top_p.
- **temperature** (*float*) – Temperature for sampling. Higher values will make the model more random (limit of temp -> 0 is just taking the top token, limit of temp -> inf is sampling from a uniform distribution).
- **freq_penalty** (*float*) – Frequency penalty for sampling - how much to penalise previous tokens. Higher values will make the model more random. Works only with str and tokens input.
- **use_past_kv_cache** (*bool*) – If True, create and use cache to speed up generation.
- **prepend_bos** (*bool, optional*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (applicable when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos (default is True unless specified otherwise). Pass True or False to override the default.
- **padding_side** (*Union[Literal["left", "right"], None], optional*) – Overrides self.tokenizer.padding_side. Specifies which side to pad when tokenizing multiple strings of different lengths.
- **return_type** (*Optional[str]*) – The type of the output to return - a string or a list of strings ('str'), a tensor of tokens ('tokens'), a tensor of output embeddings ('embeds') or whatever the format of the input was ('input').
- **verbose** (*bool*) – If True, show tqdm progress bars for generation.

RETURNS:

outputs (str, List[str], Int[torch.Tensor, “batch pos_plus_new_tokens”], Float[torch.Tensor,

”batch pos_plus_new_tokens hidden_size”]): generated sequence. Str, tokens or embeddings. If input is embeddings and return type is tokens or string, returns only new generated sequence. In other cases returns sequence including input sequence.

```
get_pos_offset(past_kv_cache, batch_size)
```

```
get_residual(embed, pos_offset, prepend_bos=None, attention_mask=None,
    tokens=None, return_shortformer_pos_embed=True, device=None)
```

```
get_token_position(single_token: Union[str, int], input: Union[str,
    Float[Tensor, 'pos'], Float[Tensor, '1 pos']], mode='first', prepend_bos:
    Optional[bool] = None, padding_side: Optional[Literal['left', 'right']] =
    None)
```

Get the position of a single_token in a string or sequence of tokens.

Raises an error if the token is not present.

Gotcha: If you’re inputting a string, it’ll automatically be tokenized. Be careful about the setting for prepend_bos! When a string is input to the model, a BOS (beginning of sequence) token is prepended by default when the string is tokenized because self.cfg.default_prepend_bos is set to True unless specified otherwise. But this should only be done at the START of the input, not when inputting part of the prompt. If you’re getting weird off-by-one errors, check carefully for what the setting should be!

PARAMETERS:

- **single_token** (*Union[str, int]*) – The token to search for. Can be a token index, or a string (but the string must correspond to a single token).
- **input** (*Union[str, torch.Tensor]*) – The sequence to search in. Can be a string or a rank 1 tensor of tokens or a rank 2 tensor of tokens with a dummy batch dimension.
- **mode** (*str, optional*) – If there are multiple matches, which match to return. Supports “first” or “last”. Defaults to “first”.
- **prepend_bos** (*bool, optional*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (only applies when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos which is set to True unless specified otherwise. Pass True or False to locally override the default.
- **padding_side** (*Union[Literal["left", "right"], None], optional*) – Overrides self.tokenizer.padding_side. Specifies which side to pad when tokenizing multiple strings of different lengths.

```
init_weights()
```

Initialize weights.

LayerNorm weights are already initialized to 1.0, and all biases are initialized to 0.0 (including LayerNorm), so this just initializes weight matrices.

Weight matrices are set to empty by default (to save space + compute, since they're the bulk of the parameters), so it is important to call this if you are not loading in pretrained weights! Note that this function assumes that weight names being with *W_*.

Set seed here to ensure determinism.

This does NOT follow the PyTorch scheme, which as far as I can tell is super out of date but no one has gotten round to updating it? https://github.com/pytorch/pytorch/issues/18182

The default PyTorch scheme is the following: all linear layers use uniform(-1/sqrt(fan_in), 1/sqrt(fan_in)) for weights, and uniform(-1/sqrt(fan_in), 1/sqrt(fan_in)) for biases. For biases, fan_in is computed using the fan_in for the weight matrix of the linear layer. Note tha it *does not actually* use Kaiming initialization, despite the fact that it calls the function.

However, for Transformer blocks, it instead initializes biases to zero and weights using Xavier uniform, that is: uniform(-sqrt(6 / (fan_in + fan_out)), sqrt(6 / (fan_in + fan_out))) for weights.

PyTorch Transformers are especially bad - TransformerEncoder initializes all layers to the exact same weights?! https://github.com/pytorch/pytorch/issues/72253.

The best paper I've found on transformer initialization is the muP paper, but haven't integrated those ideas yet: https://arxiv.org/abs/2203.03466

We split off the initialization into separate functions because muP initialization handles different parts of the model differently.

```
input_to_embed(input: Union[str, List[str], Int[Tensor, 'batch pos']],
    prepend_bos: Optional[bool] = None, padding_side: Optional[Literal['left',
    'right']] = None, attention_mask: Optional[Tensor] = None, past_kv_cache:
    Optional[HookedTransformerKeyValueCache] = None) → Tuple[Float[Tensor,
    'batch pos d_model'], Optional[Int[Tensor, 'batch pos']],
    Optional[Float[Tensor, 'batch pos d_model']], Optional[Tensor]]
```

Convert input to first residual stream.

PARAMETERS:

- **input** (*Union[str, List[str], Int[torch.Tensor, "batch pos"]]*) – The input to the model.
- **prepend_bos** (*bool, optional*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (only applies when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos which is set to True unless specified otherwise. Pass True or False to locally override the default.
- **padding_side** (*[Literal["left", "right"], optional*) – Overrides self.tokenizer.padding_side. Specifies which side to pad when tokenizing multiple strings of different lengths.
- **past_kv_cache** (*HookedTransformerKeyValueCache, optional*) – If passed, we're doing caching and attention_mask will be stored in the cache.

```
ln_final: Module
```

load_and_process_state_dict(state_dict: Dict[str, Tensor], fold_ln: bool = True, center_writing_weights: bool = True, center_unembed: bool = True, fold_value_biases: bool = True, refactor_factored_attn_matrices: bool = False)

Load & Process State Dict.

Load a state dict into the model, and to apply processing to simplify it. The state dict is assumed to be in the HookedTransformer format.

See the relevant method (same name as the flag) for more details on the folding, centering and processing flags.

PARAMETERS:

- **state_dict** (*dict*) – The state dict of the model, in HookedTransformer format. fold_ln
- **fold_ln** (*bool, optional*) – Whether to fold in the LayerNorm weights to the subsequent linear layer. This does not change the computation. Defaults to True.
- **center_writing_weights** (*bool, optional*) – Whether to center weights writing to the residual stream (ie set mean to be zero). Due to LayerNorm this doesn't change the computation. Defaults to True.
- **center_unembed** (*bool, optional*) – Whether to center W_U (ie set mean to be zero). Softmax is translation invariant so this doesn't affect log probs or loss, but does change logits. Defaults to True.
- **fold_value_biases** (*bool, optional*) – Whether to fold the value biases into the output bias. Because attention patterns add up to 1, the value biases always have a constant effect on a layer's output, and it doesn't matter which head a bias is associated with. We can factor this all into a single output bias to the layer, and make it easier to interpret the head's output.
- **refactor_factored_attn_matrices** (*bool, optional*) – Whether to convert the factored matrices (W_Q & W_K, and W_O & W_V) to be "even". Defaults to False.
- **model_name** (*str, optional*) – checks the model name for special cases of state dict loading. Only used for Redwood 2L model currently.

load_sample_training_dataset(**kwargs)

Load Sample Training Dataset.

Helper function to load in a 10K-20K dataset of elements from the model's training data distribution.

Wrapper around utils.get_dataset, which identifies the appropriate dataset the pretrained models. Each dataset has a 'text' field, which contains the relevant info, some have several meta data fields.

Kwargs will be passed to utils.get_dataset (e.g. cache_dir to set download location)

Notes:

- PT-2's training data is not open source. OpenWebText is a replication (links with >3 karma on Reddit)
- OPT's training data is not open source, and is a mess of different things that is hard to replicate. I default to the Pile, which covers some of it, but imperfectly.

(Some models will have actually been trained on the data supplied here, for some it's from the validation set).

```
loss_fn(logits: Float[Tensor, 'batch pos d_vocab'], tokens: Int[Tensor, 'batch pos'], attention_mask: Optional[Int[Tensor, 'batch pos']] = None, per_token: bool = False)
```

Wrapper around *utils.lm_cross_entropy_loss*.

Used in forward() with return_type=="loss" or "both".

```
move_model_modules_to_device()
```

```
mps()
```

Wrapper around mps that also changes *self.cfg.device*.

```
process_weights_(fold_ln: bool = True, center_writing_weights: bool = True, center_unembed: bool = True, refactor_factored_attn_matrices: bool = False)
```

Wrapper around *load_and_process_state_dict*.

Wrapper around load_and_process_state_dict to allow for in-place processing of the weights. This is useful if using HookedTransformer for training, if we then want to analyse a cleaner version of the same model.

```
refactor_factored_attn_matrices(state_dict: Dict[str, Tensor])
```

Experimental method for managing queries, keys and values.

As argued in [A Mathematical Framework for Transformer Circuits](https://transformer-circuits.pub/2021/framework/index.html), queries, keys and values are somewhat arbitrary intermediate terms when computing with the low rank factored matrices W_QK = W_Q @ W_K.T and W_OV = W_V @ W_O, and these matrices are the only thing determining head behaviour. But there are many ways to find a low rank factorization to a given matrix, and hopefully some of these are more interpretable than others! This method is one attempt, which makes all of the matrices have orthogonal rows or columns, W_O into a rotation and W_Q and W_K having the nth column in each having the same norm. The formula is $W_V = U@S, W_O = Vh.T, W_Q = U@S.sqrt(), W_K = Vh@S.sqrt()$.

More details:

If W_OV = U @ S @ Vh.T in its singular value decomposition, (where S is in R^d_head not R^d_model, as W_OV is low rank), W_OV = (U @ S) @ (Vh.T) is an equivalent low

rank factorisation, where rows/columns of each matrix are orthogonal! So setting $W_V = US$ and $W_O = Vh.T$ works just as well. I *think* this is a more interpretable setup, because now W_O is just a rotation, and doesn't change the norm, so z has the same norm as the result of the head.

For $W_QK = W_Q@W_K.T$ we use the refactor $W_Q = U@S.sqrt()$ and $W_K = Vh@S.sqrt()$, which is also equivalent ($S == S.sqrt()@S.sqrt()$ as S is diagonal). Here we keep the matrices as having the same norm, since there's not an obvious asymmetry between the keys and queries.

Biases are more fiddly to deal with. For OV it's pretty easy - we just need (x @ W_V + b_V) @ W_O + b_O to be preserved, so we can set b_V' = 0. and b_O' = b_V @ W_O + b_O (note that b_V in R^{head_index x d_head} while b_O in R^{d_model}, so we need to sum b_V @ W_O along the head_index dimension too).

For QK it's messy - we need to preserve the bilinear form of (x @ W_Q + b_Q) * (y @ W_K + b_K), which is fairly messy. To deal with the biases, we concatenate them to W_Q and W_K to simulate a d_model+1 dimensional input (whose final coordinate is always 1), do the SVD factorization on this effective matrix, then separate out into final weights and biases.

```
run_with_cache(*model_args, return_cache_object: Literal[True] = True,
    **kwargs) → Tuple[Output, ActivationCache]
```

```
run_with_cache(*model_args, return_cache_object: Literal[False], **kwargs) →
    Tuple[Output, Dict[str, Tensor]]
```

Wrapper around *run_with_cache* in HookedRootModule.

If return_cache_object is True, this will return an ActivationCache object, with a bunch of useful HookedTransformer specific methods, otherwise it will return a dictionary of activations as in HookedRootModule.

```
sample_datapoint(tokenize: bool = False, prepend_bos: Optional[bool] = None,
    padding_side: Optional[Literal['left', 'right']] = None) → Union[str,
    Float[Tensor, '1 pos']]
```

Sample Data Point from Dataset.

Helper function to randomly sample a data point from self.dataset, a small dataset from the data distribution the model was trained on.

Implicitly calls self.load_sample_training_dataset if it hasn't already been called. Only works for pretrained models with an associated dataset. But you can manually replace self.dataset with a dataset of your choice if you want.

PARAMETERS:

- **tokenize** (*bool*) – Whether to return tokens (instead of text). Defaults to False. Note that the returned tokens will be automatically truncated to the model's max context size.
- **prepend_bos** (*bool, optional*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (applicable when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos (default is True unless specified otherwise). Pass True or False to override the default.
- **padding_side** (*Union[Literal["left", "right"], None], optional*) – Overrides self.tokenizer.padding_side. Specifies which side to pad when tokenizing multiple strings of different lengths.

set_tokenizer(tokenizer, default_padding_side='right')

Set the tokenizer to use for this model.

PARAMETERS:

- **tokenizer** (*PreTrainedTokenizer*) – a pretrained HuggingFace tokenizer.
- **default_padding_side** (*str*) – "right" or "left", which side to pad on.

set_ungroup_grouped_query_attention(ungroup_grouped_query_attention: bool)

Toggles whether to ungroup the grouped key and value heads in models with grouped query attention (GQA).

set_use_attn_in(use_attn_in: bool)

Toggles whether to allow editing of inputs to each attention head.

set_use_attn_result(use_attn_result: bool)

Toggle whether to explicitly calculate and expose the result for each attention head.

Useful for interpretability but can easily burn through GPU memory.

set_use_hook_mlp_in(use_hook_mlp_in: bool)

Toggles whether to allow storing and editing inputs to each MLP layer.

set_use_split_qkv_input(use_split_qkv_input: bool)

Toggles whether to allow editing of inputs to each attention head.

to(device_or_dtype: Union[device, str, dtype], print_details: bool = True)

Move and/or cast the parameters and buffers.

This can be called as

```
to(device=None, dtype=None, non_blocking=False)

to(dtype, non_blocking=False)

to(tensor, non_blocking=False)

to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

> Note
>
> This method modifies the module in-place.

PARAMETERS:

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

RETURNS:

self

RETURN TYPE:

Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_single_str_token(int_token: int) → str

to_single_token(string)

Map a string that makes up a single token to the id for that token.

Raises an error for strings that are not a single token! If uncertain use to_tokens.

to_str_tokens(input: Union[str, Int[Tensor, 'pos'], Int[Tensor, '1 pos'], Int[ndarray, 'pos'], Int[ndarray, '1 pos'], list], prepend_bos: Optional[bool] = None, padding_side: Optional[Literal['left', 'right']] = None) → Union[List[str], List[List[str]]]

Map text, a list of text or tokens to a list of tokens as strings.

Gotcha: prepend_bos prepends a beginning of string token. This is a recommended default when inputting a prompt to the model as the first token is often treated weirdly, but should only be done at the START of the prompt. If prepend_bos=None is passed, it

implies the usage of self.cfg.default_prepend_bos which is set to True unless specified otherwise. Therefore, make sure to locally turn it off by passing prepend_bos=False if you're looking at the tokenization of part of the prompt! (Note: some models eg GPT-2 were not trained with a BOS token, others (OPT and my models) were)

Gotcha2: Tokenization of a string depends on whether there is a preceding space and whether the first letter is capitalized. It's easy to shoot yourself in the foot here if you're not careful!

Gotcha3: If passing a string that exceeds the model's context length (model.cfg.n_ctx), it will be truncated.

PARAMETERS:

- **input** (*Union[str, list, torch.Tensor]*) – The input - either a string or a tensor of tokens. If tokens, should be a tensor of shape [pos] or [1, pos].
- **prepend_bos** (*bool, optional*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (only applies when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos which is set to True unless specified otherwise. Pass True or False to locally override the default.
- **padding_side** (*Union[Literal["left", "right"], None], optional*) – Overrides self.tokenizer.padding_side. Specifies which side to pad when tokenizing multiple strings of different lengths.

RETURNS:

List of individual tokens as strings

RETURN TYPE:

str_tokens

```
to_string(tokens: Union[List[int], Int[Tensor, ''], Int[Tensor, 'batch pos'],
    Int[Tensor, 'pos'], ndarray, List[Int[Tensor, 'pos']]]) → Union[str,
    List[str]]
```

Tokens to String(s).

Converts a tensor of tokens to a string (if rank 1) or a list of strings (if rank 2).

Accepts lists of tokens and numpy arrays as inputs too (and converts to tensors internally)

```
to_tokens(input: Union[str, List[str]], prepend_bos: Optional[bool] = None,
    padding_side: Optional[Literal['left', 'right']] = None, move_to_device:
    bool = True, truncate: bool = True) → Int[Tensor, 'batch pos']
```

Converts a string to a tensor of tokens.

If prepend_bos is True, prepends the BOS token to the input - this is recommended when creating a sequence of tokens to be input to a model.

Gotcha: prepend_bos prepends a beginning of string token. This is a recommended default when inputting a prompt to the model as the first token is often treated weirdly, but should only be done at the START of the prompt. Make sure to turn it off if you're

looking at the tokenization of part of the prompt! (Note: some models eg GPT-2 were not trained with a BOS token, others (OPT and my models) were)

Gotcha2: Tokenization of a string depends on whether there is a preceding space and whether the first letter is capitalized. It's easy to shoot yourself in the foot here if you're not careful!

PARAMETERS:

- **input** (*Union[str, List[str]]*) – The input to tokenize.
- **prepend_bos** (*bool, optional*) – Overrides self.cfg.default_prepend_bos. Whether to prepend the BOS token to the input (only applies when input is a string). Defaults to None, implying usage of self.cfg.default_prepend_bos which is set to True unless specified otherwise. Pass True or False to locally override the default.
- **padding_side** (*Union[Literal["left", "right"], None], optional*) – Overrides self.tokenizer.padding_side. Specifies which side to pad when tokenizing multiple strings of different lengths.
- **move_to_device** (*bool*) – Whether to move the output tensor of tokens to the device the model lives on. Defaults to True truncate (bool): If the output tokens are too long, whether to truncate the output tokens to the model's max context window. Does nothing for shorter inputs. Defaults to True.

```
tokens_to_residual_directions(tokens: Union[str, int, Int[Tensor, ''],
    Int[Tensor, 'pos'], Int[Tensor, 'batch pos']]) → Union[Float[Tensor,
    'd_model'], Float[Tensor, 'pos d_model'], Float[Tensor, 'batch pos
    d_model']]
```

Map tokens to a tensor with the unembedding vector for those tokens.

I.e. the vector in the residual stream that we dot with to the get the logit for that token.

WARNING: If you use this without folding in LayerNorm, the results will be misleading and may be incorrect, as the LN weights change the unembed map. This is done automatically with the fold_ln flag on from_pretrained

WARNING 2: LayerNorm scaling will scale up or down the effective direction in the residual stream for each output token on any given input token position.

ActivationCache.apply_ln_to_stack will apply the appropriate scaling to these directions.

PARAMETERS:

tokens (*Union[str, int, torch.Tensor]*) – The token(s). If a single token, can be a single element tensor, an integer, or string. If string, will be mapped to a single token using to_single_token, and an error raised if it's multiple tokens. The method also works for a batch of input tokens.

RETURNS:

The unembedding vector for the token(s), a stack of [d_model] tensor.

RETURN TYPE:

residual_direction torch.Tensor

```
class transformer_lens.HookedTransformer.Output(logits: Float[Tensor, 'batch pos d_vocab'], loss: Union[Float[Tensor, ''], Float[Tensor, 'batch pos-1']])
```

Bases: `NamedTuple`

Output Named Tuple.

Named tuple object for if we want to output both logits and loss.

```
logits: Float[Tensor, 'batch pos d_vocab']
```

Alias for field number 0

```
loss: Union[Float[Tensor, ''], Float[Tensor, 'batch pos-1']]
```

Alias for field number 1