transformer_lens.ActivationCache

Activation Cache.

The `ActivationCache` is at the core of Transformer Lens. It is a wrapper that stores all important activations from a forward pass of the model, and provides a variety of helper functions to investigate them.

Getting Started:

When reading these docs for the first time, we recommend reading the main `ActivationCache` class first, including the examples, and then skimming the available methods. You can then refer back to these docs depending on what you need to do.

class transformer_lens.ActivationCache.ActivationCache(cache_dict: Dict[str, Tensor], model, has_batch_dim: bool = True)

Bases: `object`

Activation Cache.

A wrapper that stores all important activations from a forward pass of the model, and provides a variety of helper functions to investigate them.

The `ActivationCache` is at the core of Transformer Lens. It is a wrapper that stores all important activations from a forward pass of the model, and provides a variety of helper functions to investigate them. The common way to access it is to run the model with `transformer_lens.HookedTransformer.run_with_cache()`.

Examples:

When investigating a particular behaviour of a modal, a very common first step is to try and understand which components of the model are most responsible for that behaviour. For example, if you're investigating the prompt "Why did the chicken cross the" -> " road", you might want to understand if there is a specific sublayer (mlp or multi-head attention) that is responsible for the model predicting "road". This kind of analysis commonly falls under the category of "logit attribution" or "direct logit attribution" (DLA).

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

```
>>> _logits, cache = model.run_with_cache("Why did the chicken cross the")
>>> residual_stream, labels = cache.decompose_resid(return_labels=True, mode="attn"
>>> print(labels[0:3])
['embed', 'pos_embed', '0_attn_out']
```

```
>>> answer = " road" # Note the proceeding space to match the model's tokenization
>>> logit_attrs = cache.logit_attrs(residual_stream, answer)
>>> print(logit_attrs.shape) # Attention layers
torch.Size([10, 1, 7])
```

```
>>> most_important_component_idx = torch.argmax(logit_attrs)
>>> print(labels[most_important_component_idx])
3_attn_out
```

You can also dig in with more granularity, using `get_full_resid_decomposition()` to get the residual stream by individual component (mlp neurons and individual attention heads). This creates a larger residual stack, but the approach of using :meth"*logit_attrs* remains the same.

Equally you might want to find out if the model struggles to construct such excellent jokes until the very last layers, or if it is trivial and the first few layers are enough. This kind of analysis is called "logit lens", and you can find out more about how to do that with `ActivationCache.accumulated_resid()`.

Warning:

`ActivationCache` is designed to be used with `transformer_lens.HookedTransformer`, and will not work with other models. It's also designed to be used with all activations of `transformer_lens.HookedTransformer` being cached, and some internal methods will break without that.

The biggest footgun and source of bugs in this code will be keeping track of indexes, dimensions, and the numbers of each. There are several kinds of activations:

- Internal attn head vectors: q, k, v, z. Shape [batch, pos, head_index, d_head].
- Internal attn pattern style results: pattern (post softmax), attn_scores (pre-softmax). Shape [batch, head_index, query_pos, key_pos].
- Attn head results: result. Shape [batch, pos, head_index, d_model].
- Internal MLP vectors: pre, post, mid (only used for solu_ln - the part between activation + layernorm). Shape [batch, pos, d_mlp].
- Residual stream vectors: resid_pre, resid_mid, resid_post, attn_out, mlp_out, embed, pos_embed, normalized (output of each LN or LNPre). Shape [batch, pos, d_model].
- LayerNorm Scale: scale. Shape [batch, pos, 1].

Sometimes the batch dimension will be missing because we applied *remove_batch_dim* (used when batch_size=1), and as such all library functions *should* be robust to that.

Type annotations are in the following form:

- layers_covered is the number of layers queried in functions that stack the residual stream.
- batch_and_pos_dims is the set of dimensions from batch and pos - by default this is ["batch", "pos"], but is only ["pos"] if we've removed the batch dimension and is [()] if we've removed batch dimension and are applying a pos slice which indexes a specific position.

PARAMETERS:

- **cache_dict** – A dictionary of cached activations from a model run.
- **model** – The model that the activations are from.
- **has_batch_dim** – Whether the activations have a batch dimension.

```
accumulated_resid(layer: Optional[int] = None, incl_mid: bool = False,
    apply_ln: bool = False, pos_slice: Optional[Union[Slice, int, Tuple[int],
    Tuple[int, int], Tuple[int, int, int], List[int], Tensor, ndarray]] =
    None, mlp_input: bool = False, return_labels: bool = False) →
    Union[Float[Tensor, 'layers_covered *batch_and_pos_dims d_model'],
    Tuple[Float[Tensor, 'layers_covered *batch_and_pos_dims d_model'],
    List[str]]]
```

Accumulated Residual Stream.

Returns the accumulated residual stream at each layer/sub-layer. This is useful for *Logit Lens <https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens>* style analysis, where it can be thought of as what the model "believes" at each point in the residual stream.

To project this into the vocabulary space, remember that there is a final layer norm in most decoder-only transformers. Therefore, you need to first apply the final layer norm (which can be done with *apply_ln*), and then multiply by the unembedding matrix (W_U).

If you instead want to look at contributions to the residual stream from each component (e.g. for direct logit attribution), see `decompose_resid()` instead, or `get_full_resid_decomposition()` if you want contributions broken down further into each MLP neuron.

Examples:

Logit Lens analysis can be done as follows:

```
>>> from transformer_lens import HookedTransformer
>>> from einops import einsum
>>> import torch
>>> import pandas as pd
```

```
>>> model = HookedTransformer.from_pretrained("tiny-stories-1M", device="cpu")
Loaded pretrained model tiny-stories-1M into HookedTransformer
```

```
>>> prompt = "Why did the chicken cross the"
>>> answer = " road"
>>> logits, cache = model.run_with_cache("Why did the chicken cross the")
>>> answer_token = model.to_single_token(answer)
>>> print(answer_token)
2975
```

```
>>> accum_resid, labels = cache.accumulated_resid(return_labels=True, apply_ln=
>>> last_token_accum = accum_resid[:, 0, -1, :]  # layer, batch, pos, d_model
>>> print(last_token_accum.shape)  # layer, d_model
torch.Size([9, 64])
```

```
>>> W_U = model.W_U
>>> print(W_U.shape)
torch.Size([64, 50257])
```

```
>>> layers_unembedded = einsum(
...         last_token_accum,
...         W_U,
...         "layer d_model, d_model d_vocab -> layer d_vocab"
...     )
>>> print(layers_unembedded.shape)
torch.Size([9, 50257])
```

```
>>> # Get the rank of the correct answer by layer
>>> sorted_indices = torch.argsort(layers_unembedded, dim=1, descending=True)
>>> rank_answer = (sorted_indices == 2975).nonzero(as_tuple=True)[1]
>>> print(pd.Series(rank_answer, index=labels))
0_pre         4442
1_pre          382
2_pre          982
3_pre         1160
4_pre          408
5_pre          145
6_pre           78
7_pre          387
final_post       6
dtype: int64
```

PARAMETERS:

- **layer** – The layer to take components up to - by default includes resid_pre for that layer and excludes resid_mid and resid_post for that layer. If set as *n_layers*, *-1* or *None* it will return all residual streams, including the final one (i.e. immediately pre logits). The indices are taken such that this gives the accumulated streams up to the input to layer l.
- **incl_mid** – Whether to return *resid_mid* for all previous layers.
- **apply_ln** – Whether to apply LayerNorm to the stack.
- **pos_slice** – A slice object to apply to the pos dimension. Defaults to None, do nothing.
- **mlp_input** – Whether to include resid_mid for the current layer. This essentially gives the MLP input rather than the attention input.
- **return_labels** – Whether to return a list of labels for the residual stream components. Useful for labelling graphs.

RETURNS:

A tensor of the accumulated residual streams. If *return_labels* is True, also returns a list of labels for the components (as a tuple in the form *(components, labels)*).

```
apply_ln_to_stack(residual_stack: Float[Tensor, 'num_components
    *batch_and_pos_dims d_model'], layer: Optional[int] = None, mlp_input:
    bool = False, pos_slice: Optional[Union[Slice, int, Tuple[int], Tuple[int,
    int], Tuple[int, int, int], List[int], Tensor, ndarray]] = None,
    batch_slice: Optional[Union[Slice, int, Tuple[int], Tuple[int, int],
    Tuple[int, int, int], List[int], Tensor, ndarray]] = None, has_batch_dim:
    bool = True) → Float[Tensor, 'num_components *batch_and_pos_dims_out
    d_model']
```

Apply Layer Norm to a Stack.

Takes a stack of components of the residual stream (eg outputs of decompose_resid or accumulated_resid), treats them as the input to a specific layer, and applies the layer norm scaling of that layer to them, using the cached scale factors - simulating what that component of the residual stream contributes to that layer's input.

The layernorm scale is global across the entire residual stream for each layer, batch element and position, which is why we need to use the cached scale factors rather than just applying a new LayerNorm.

If the model does not use LayerNorm or RMSNorm, it returns the residual stack unchanged.

PARAMETERS:

- **residual_stack** – A tensor, whose final dimension is d_model. The other trailing dimensions are assumed to be the same as the stored hook_scale - which may or may not include batch or position dimensions.
- **layer** – The layer we're taking the input to. In [0, n_layers], n_layers means the unembed. None maps to the n_layers case, ie the unembed.
- **mlp_input** – Whether the input is to the MLP or attn (ie ln2 vs ln1). Defaults to False, ie ln1. If layer==n_layers, must be False, and we use ln_final
- **pos_slice** – The slice to take of positions, if residual_stack is not over the full context, None means do nothing. It is assumed that pos_slice has already been applied to residual_stack, and this is only applied to the scale. See utils.Slice for details. Defaults to None, do nothing.
- **batch_slice** – The slice to take on the batch dimension. Defaults to None, do nothing.
- **has_batch_dim** – Whether residual_stack has a batch dimension.

```
apply_slice_to_batch_dim(batch_slice: Optional[Union[Slice, int, Tuple[int],
    Tuple[int, int], Tuple[int, int, int], List[int], Tensor, ndarray]]) →
    ActivationCache
```

Apply a Slice to the Batch Dimension.

PARAMETERS:

batch_slice – The slice to apply to the batch dimension.

RETURNS:

The ActivationCache with the batch dimension sliced.

```
compute_head_results()
```

Compute Head Results.

Computes and caches the results for each attention head, ie the amount contributed to the residual stream from that head. attn_out for a layer is the sum of head results plus b_O. Intended use is to enable use_attn_results when running and caching the model, but this can be useful if you forget.

```
decompose_resid(layer: Optional[int] = None, mlp_input: bool = False, mode:
    Literal['all', 'mlp', 'attn'] = 'all', apply_ln: bool = False, pos_slice:
    Optional[Union[Slice, int, Tuple[int], Tuple[int, int], Tuple[int, int,
    int], List[int], Tensor, ndarray]] = None, incl_embeds: bool = True,
    return_labels: bool = False) → Union[Float[Tensor, 'layers_covered
    *batch_and_pos_dims d_model'], Tuple[Float[Tensor, 'layers_covered
    *batch_and_pos_dims d_model'], List[str]]]
```

Decompose the Residual Stream.

Decomposes the residual stream input to layer L into a stack of the output of previous layers. The sum of these is the input to layer L (plus embedding and pos embedding). This is useful for attributing model behaviour to different components of the residual stream

PARAMETERS:

- **layer** – The layer to take components up to - by default includes resid_pre for that layer and excludes resid_mid and resid_post for that layer. layer==n_layers means to return all layer outputs incl in the final layer, layer==0 means just embed and pos_embed. The indices are taken such that this gives the accumulated streams up to the input to layer l
- **mlp_input** – Whether to include attn_out for the current layer - essentially decomposing the residual stream that's input to the MLP input rather than the Attn input.
- **mode** – Values are "all", "mlp" or "attn". "all" returns all components, "mlp" returns only the MLP components, and "attn" returns only the attention components. Defaults to "all".
- **apply_ln** – Whether to apply LayerNorm to the stack.
- **pos_slice** – A slice object to apply to the pos dimension. Defaults to None, do nothing.
- **incl_embeds** – Whether to include embed & pos_embed
- **return_labels** – Whether to return a list of labels for the residual stream components. Useful for labelling graphs.

RETURNS:

A tensor of the accumulated residual streams. If *return_labels* is True, also returns a list of labels for the components (as a tuple in the form *(components, labels)*).

```
get_full_resid_decomposition(layer: Optional[int] = None, mlp_input: bool =
    False, expand_neurons: bool = True, apply_ln: bool = False, pos_slice:
    Optional[Union[Slice, int, Tuple[int], Tuple[int, int], Tuple[int, int,
    int], List[int], Tensor, ndarray]] = None, return_labels: bool = False) →
    Union[Float[Tensor, 'num_components *batch_and_pos_dims d_model'],
    Tuple[Float[Tensor, 'num_components *batch_and_pos_dims d_model'],
    List[str]]]
```

Get the full Residual Decomposition.

Returns the full decomposition of the residual stream into embed, pos_embed, each head result, each neuron result, and the accumulated biases. We break down the residual stream that is input into some layer.

PARAMETERS:

- **layer** – The layer we're inputting into. layer is in [0, n_layers], if layer==n_layers (or None) we're inputting into the unembed (the entire stream), if layer==0 then it's just embed and pos_embed
- **mlp_input** – Are we inputting to the MLP in that layer or the attn? Must be False for final layer, since that's the unembed.
- **expand_neurons** – Whether to expand the MLP outputs to give every neuron's result or just return the MLP layer outputs.
- **apply_ln** – Whether to apply LayerNorm to the stack.
- **pos_slice** – Slice of the positions to take.
- **return_labels** – Whether to return the labels.

```
get_neuron_results(layer: int, neuron_slice: Optional[Union[Slice, int,
    Tuple[int], Tuple[int, int], Tuple[int, int, int], List[int], Tensor,
    ndarray]] = None, pos_slice: Optional[Union[Slice, int, Tuple[int],
    Tuple[int, int], Tuple[int, int, int], List[int], Tensor, ndarray]] =
    None) → Float[Tensor, '*batch_and_pos_dims num_neurons d_model']
```

Get Neuron Results.

Get the results of for neurons in a specific layer (i.e, how much each neuron contributes to the residual stream). Does it for the subset of neurons specified by neuron_slice, defaults to all of them. Does *not* cache these because it's expensive in space and cheap to compute.

PARAMETERS:

- **layer** – Layer index.
- **neuron_slice** – Slice of the neuron.
- **pos_slice** – Slice of the positions.

RETURNS:

Tensor of the results.

```
items()
```

Items of the ActivationCache.

RETURNS:

List of all items ((key, value) tuples).

```
keys()
```

Keys of the ActivationCache.

EXAMPLES

```
>>> from transformer_lens import HookedTransformer
>>> model = HookedTransformer.from_pretrained("tiny-stories-1M")
Loaded pretrained model tiny-stories-1M into HookedTransformer
>>> _logits, cache = model.run_with_cache("Some prompt")
>>> list(cache.keys())[0:3]
['hook_embed', 'hook_pos_embed', 'blocks.0.hook_resid_pre']
```

RETURNS:

List of all keys.

```
logit_attrs(residual_stack: Float[Tensor, 'num_components *batch_and_pos_dims
    d_model'], tokens: Union[str, int, Int[Tensor, ''], Int[Tensor, 'batch'],
    Int[Tensor, 'batch position']], incorrect_tokens: Optional[Union[str, int,
    Int[Tensor, ''], Int[Tensor, 'batch'], Int[Tensor, 'batch position']]] =
    None, pos_slice: Optional[Union[Slice, int, Tuple[int], Tuple[int, int],
    Tuple[int, int, int], List[int], Tensor, ndarray]] = None, batch_slice:
    Optional[Union[Slice, int, Tuple[int], Tuple[int, int], Tuple[int, int,
    int], List[int], Tensor, ndarray]] = None, has_batch_dim: bool = True) →
    Float[Tensor, 'num_components *batch_and_pos_dims_out']
```

Logit Attributions.

Takes a residual stack (typically the residual stream decomposed by components), and calculates how much each item in the stack "contributes" to specific tokens.

It does this by:

1. Getting the residual directions of the tokens (i.e. reversing the unembed)
2. Taking the dot product of each item in the residual stack, with the token residual directions.

Note that if incorrect tokens are provided, it instead takes the difference between the correct and incorrect tokens (to calculate the residual directions). This is useful as sometimes we want to know e.g. which components are most responsible for selecting the correct token rather than an incorrect one. For example in the *Interpretability in the Wild paper <https://arxiv.org/abs/2211.00593>* prompts such as "John and Mary went to the shops, John gave a bag to" were investigated, and it was therefore useful to calculate attribution for the $\mathrm{Mary} - \mathrm{John}$ residual direction.

Warning:

Choosing the correct *tokens* and *incorrect_tokens* is both important and difficult. When investigating specific components it's also useful to look at it's impact on all tokens (i.e. $\text{final\_ln}(\text{residual\_stack\_item})W_U$).

PARAMETERS:

- **residual_stack** – Stack of components of residual stream to get logit attributions for.
- **tokens** – Tokens to compute logit attributions on.
- **incorrect_tokens** – If provided, compute attributions on logit difference between tokens and incorrect_tokens. Must have the same shape as tokens.
- **pos_slice** – The slice to apply layer norm scaling on. Defaults to None, do nothing.
- **batch_slice** – The slice to take on the batch dimension during layer norm scaling. Defaults to None, do nothing.
- **has_batch_dim** – Whether residual_stack has a batch dimension. Defaults to True.

RETURNS:

A tensor of the logit attributions or logit difference attributions if incorrect_tokens was provided.

```
remove_batch_dim() → ActivationCache
```

Remove the Batch Dimension (if a single batch item).

RETURNS:

The ActivationCache with the batch dimension removed.

```
stack_activation(activation_name: str, layer: int = -1, sublayer_type:
    Optional[str] = None) → Float[Tensor, 'layers_covered ...']
```

Stack Activations.

Flexible way to stack activations with a given name.

PARAMETERS:

- **activation_name** – The name of the activation to be stacked
- **layer** – 'Layer index - heads' at all layers strictly before this are included. layer must be in [1, n_layers-1], or any of (n_layers, -1, None), which all mean the final layer.
- **sublayer_type** – The sub layer type of the activation, passed to utils.get_act_name. Can normally be inferred.
- **incl_remainder** – Whether to return a final term which is "the rest of the residual stream".

```
stack_head_results(layer: int = -1, return_labels: bool = False,
    incl_remainder: bool = False, pos_slice: Optional[Union[Slice, int,
    Tuple[int], Tuple[int, int], Tuple[int, int, int], List[int], Tensor,
    ndarray]] = None, apply_ln: bool = False) → Union[Float[Tensor,
    'num_components *batch_and_pos_dims d_model'], Tuple[Float[Tensor,
    'num_components *batch_and_pos_dims d_model'], List[str]]]
```

Stack Head Results.

Returns a stack of all head results (ie residual stream contribution) up to layer L. A good way to decompose the outputs of attention layers into attribution by specific heads. Note that the num_components axis has length layer x n_heads ((layer head_index) in einops notation).

PARAMETERS:

- **layer** – Layer index - heads at all layers strictly before this are included. layer must be in [1, n_layers-1], or any of (n_layers, -1, None), which all mean the final layer.
- **return_labels** – Whether to also return a list of labels of the form "L0H0" for the heads.
- **incl_remainder** – Whether to return a final term which is "the rest of the residual stream".
- **pos_slice** – A slice object to apply to the pos dimension. Defaults to None, do nothing.
- **apply_ln** – Whether to apply LayerNorm to the stack.

```
stack_neuron_results(layer: int, pos_slice: Optional[Union[Slice, int,
    Tuple[int], Tuple[int, int], Tuple[int, int, int], List[int], Tensor,
    ndarray]] = None, neuron_slice: Optional[Union[Slice, int, Tuple[int],
    Tuple[int, int], Tuple[int, int, int], List[int], Tensor, ndarray]] =
    None, return_labels: bool = False, incl_remainder: bool = False, apply_ln:
    bool = False) → Union[Float[Tensor, 'num_components *batch_and_pos_dims
    d_model'], Tuple[Float[Tensor, 'num_components *batch_and_pos_dims
    d_model'], List[str]]]
```

Stack Neuron Results

Returns a stack of all neuron results (ie residual stream contribution) up to layer L - ie the amount each individual neuron contributes to the residual stream. Also returns a list of labels of the form "L0N0" for the neurons. A good way to decompose the outputs of MLP layers into attribution by specific neurons.

Note that doing this for all neurons is SUPER expensive on GPU memory and only works for small models or short inputs.

PARAMETERS:

- **layer** – Layer index - heads at all layers strictly before this are included. layer must be in [1, n_layers]
- **pos_slice** – Slice of the positions.
- **neuron_slice** – Slice of the neurons.
- **return_labels** – Whether to also return a list of labels of the form "L0H0" for the heads.
- **incl_remainder** – Whether to return a final term which is "the rest of the residual stream".
- **apply_ln** – Whether to apply LayerNorm to the stack.

```
to(device: Union[str, device], move_model=False) → ActivationCache
```

Move the Cache to a Device.

Mostly useful for moving the cache to the CPU after model computation finishes to save GPU memory. Note however that operations will be much slower on the CPU. Note also that some methods will break unless the model is also moved to the same device, eg *compute_head_results*.

PARAMETERS:

- **device** – The device to move the cache to (e.g. *torch.device.cpu*).
- **move_model** – Whether to also move the model to the same device. @deprecated

`toggle_autodiff(mode: bool = False)`

Toggle Autodiff Globally.

Applies *torch.set_grad_enabled(mode)* to the global state (not just TransformerLens).

Warning:

This is pretty dangerous, since autodiff is global state - this turns off torch's ability to take gradients completely and it's easy to get a bunch of errors if you don't realise what you're doing.

But autodiff consumes a LOT of GPU memory (since every intermediate activation is cached until all downstream activations are deleted - this means that computing the loss and storing it in a list will keep every activation sticking around!). So often when you're analysing a model's activations, and don't need to do any training, autodiff is more trouble than its worth.

If you don't want to mess with global state, using torch.inference_mode as a context manager or decorator achieves similar effects:

```
>>> with torch.inference_mode():
...     y = torch.Tensor([1., 2, 3])
>>> y.requires_grad
False
```

`values()`

Values of the ActivationCache.

RETURNS:

List of all values.