



UNIVERSITY OF
TORONTO

CSC485/2501 A2

TA: Andrew Liu



UNIVERSITY OF
TORONTO

From Last Week...

- Word Sense Disambiguation (q0, q1, q2) <- *last tutorial*
- **LLM causal tracing (q3) <- this tutorial!**
- After A2, you will be familiar with...
 - NLTK
 - WordNet
 - The Lesk algorithm
 - Word embeddings methods such as word2vec and BERT
 - **Investigating how language models process input and generate**

Casual Tracing

In question 3, we explore how modern LLMs store facts

- Specifically, how they answer simple, relational questions
- e.g. The capital of France is ____? The CN Tower is located in the city of ____?

In order to find how LLMs generate, we use a metric called the **indirect effect**

- i.e. the impact of disabling some hidden states in the model
- We will go through all the terminologies in this tutorial

Preliminaries

What is a neural network?

Two key components:

1. Preset network architectures (neurons)
2. Task



We do not tune each parameter ourselves, there are too many. We instead use math to approximate the best settings such that our model can complete a task, this is called *learning*.

But the issue is that we do not know what any of the numbers mean. This leads to issues in control and safety. *Interpretability* is the subfield that attempts to understand the model's learned structure.

Preliminaries: BERT



Recall BERT. The goal was *contextualized* word embeddings. We have a model learn to assign representations such that it is conditioned on (“sees”) every other token in the input.

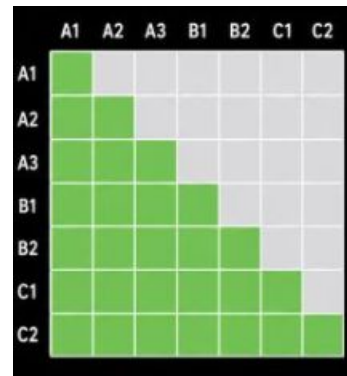
Ex. John walked to the store.

Introduce GPT2. The task is next-token prediction. Given a prefix, learn to predict the next likely token. Why not just use BERT embeddings then?

Preliminaries: Causal Masking and GPT2

BERT mixed all the embeddings together at the attention layer

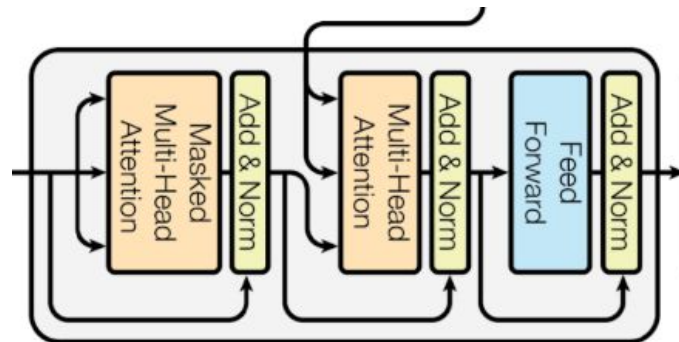
- Every token's representation is a mixture of its whole context



Applying this to GPT2, cheating will occur for the language modeling task.

Therefore each token's representation mixes only with the tokens before it. It never “sees” the tokens that come after it, as that is what it needs to predict.

This is called causal masking, a key property of the decoder block



Preliminaries: Putting it Together

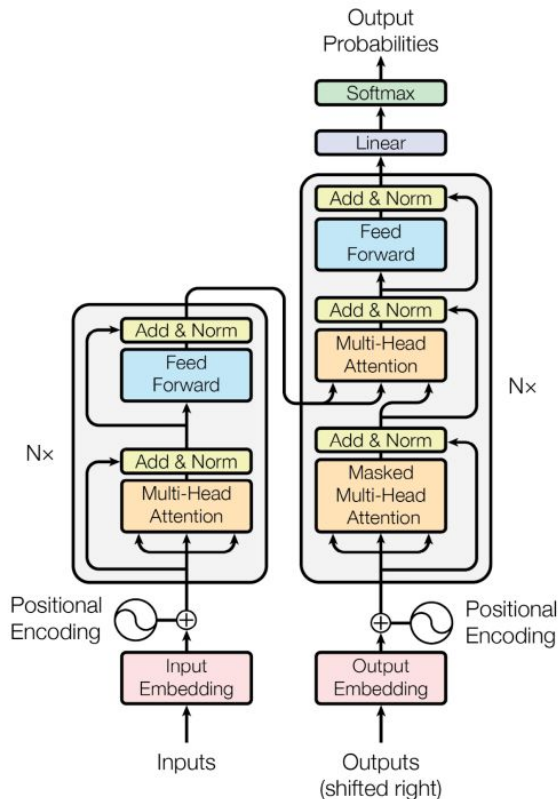
The transformer architecture is essentially two processes repeated over and over again for each block:

1. Attention: each token absorbs other tokens representations. We learn how to do the absorption
2. Feed Forward: Each token undergoes an independent transformation, extracting new features.

Attention for BERT, any token can look at any other token

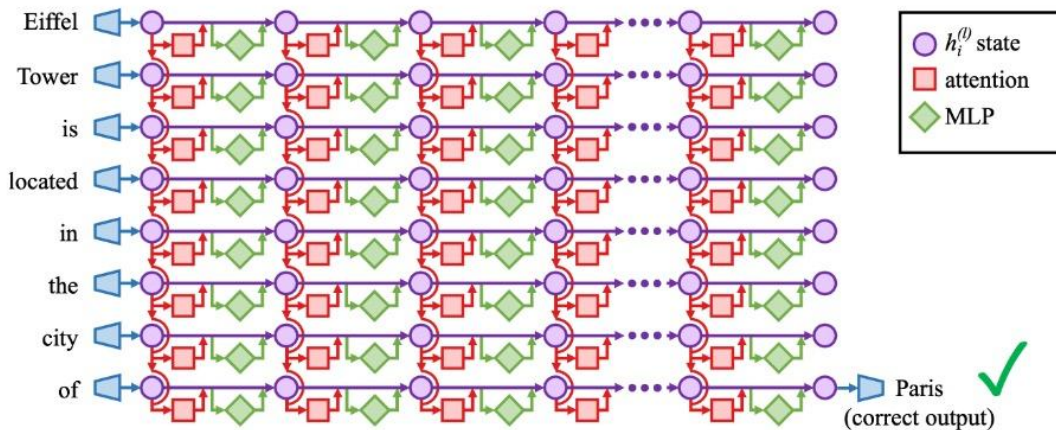
Attention for GPT2, a token can only look at preceding tokens

Repeat for each block

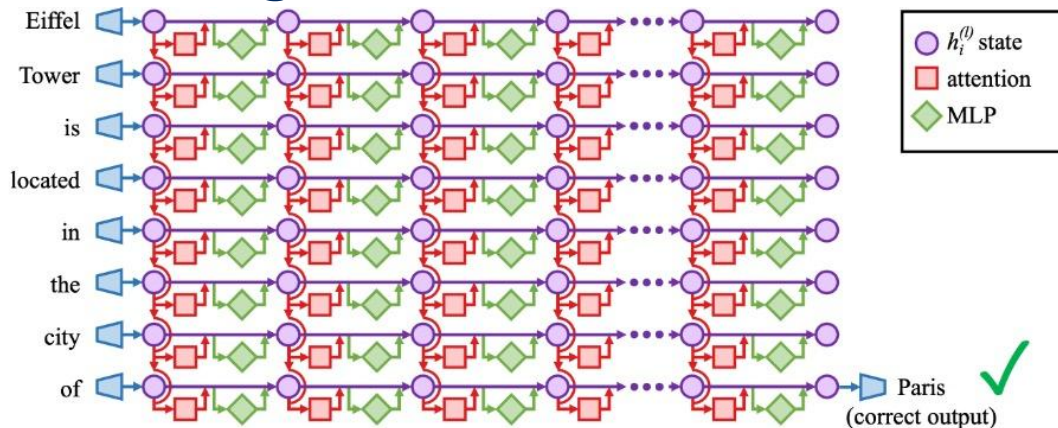


Preliminaries: GPT2

Now that we know how GPT2 is structured, we can visualize its information flow. The information in this refers to the representations.



Casual Tracking



Given the sentence “Eiffel Tower is located in the city of ...”, when does the language model realizes that the answer is “Paris”?

Assumption: the factual knowledges must be stored somewhere in the information stream

Casual Tracking

We explore three different location types:

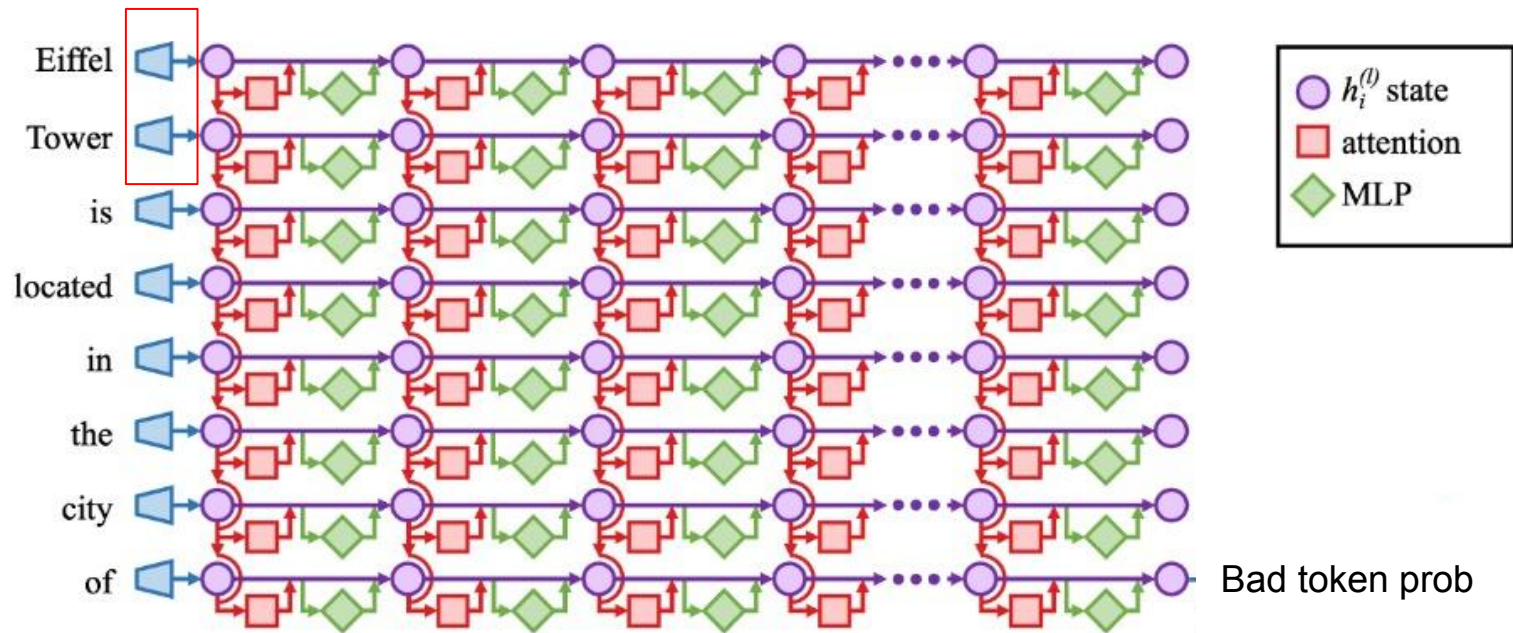
1. Each entire layer;
2. Each attention interval;
3. Each MLP interval (interval = a window of 10)

What we do:

1. Conduct a **corrupted run**: obfuscate some hidden states, compute the output distribution (the logit)
2. Conduct a **corrupted with restoration run**: corrupt the inputs, but restore the hidden state at some token positions and layers. Compute the output dist.
3. **Indirect Effect**: the difference between the corrupted and restored probability

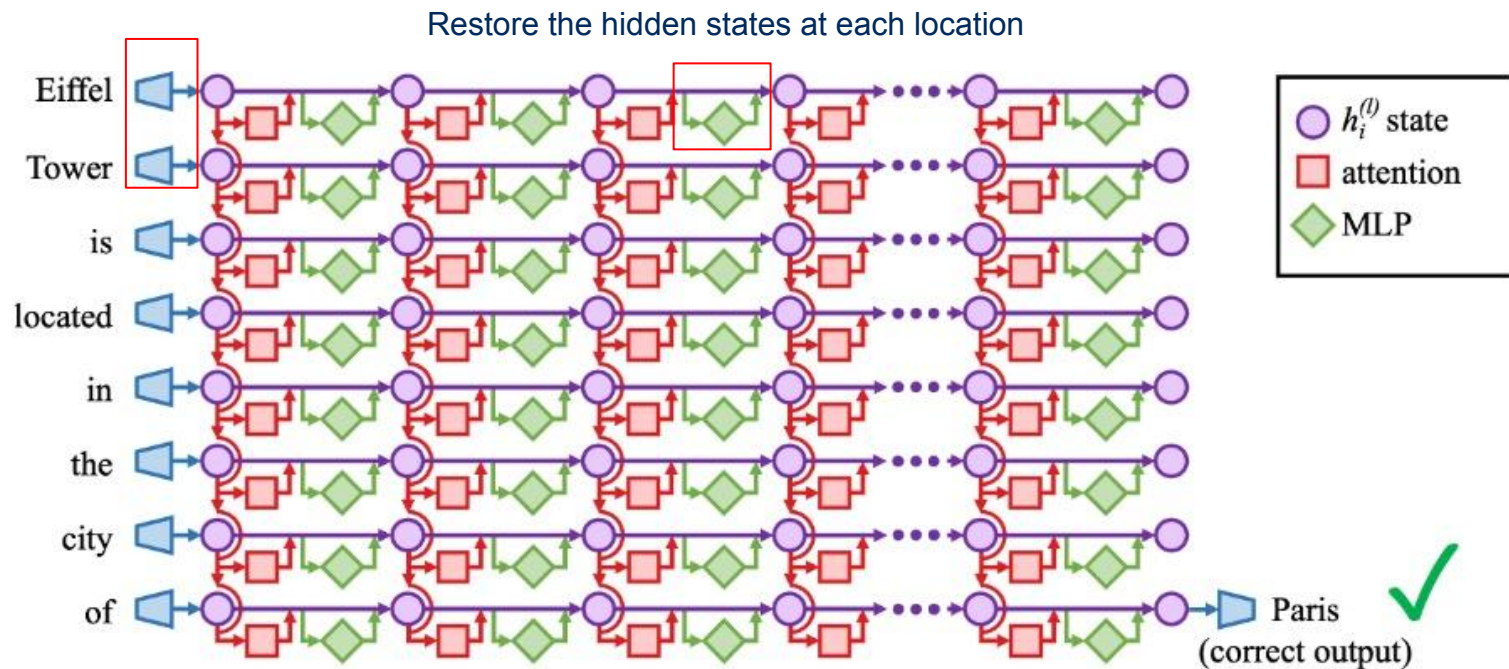
How to corrupt: given the located hidden state, add a random noise (noise * randn)

1. Corrupt



Add random noise to these representations

2. Restore



Add random noise to these representations

$$IE = P_{\text{restored}}(\text{paris}) - P_{\text{bad}}(\text{paris})$$

TransformerLens

A Library we use for mechanistic interpretability of generative language models.

How we run the process:

1. Specify a location (hook)
2. Define a function of what to do at that location
3. Run `model.run_with_hooks`

`run_with_hooks()`:

- Input: the tokenized prompts, and a `*forward_hook*`
- Return: the logits

Hooks: `List[Tuple[str, Callable]]`: a list of hooked LM locations and the relevant functions

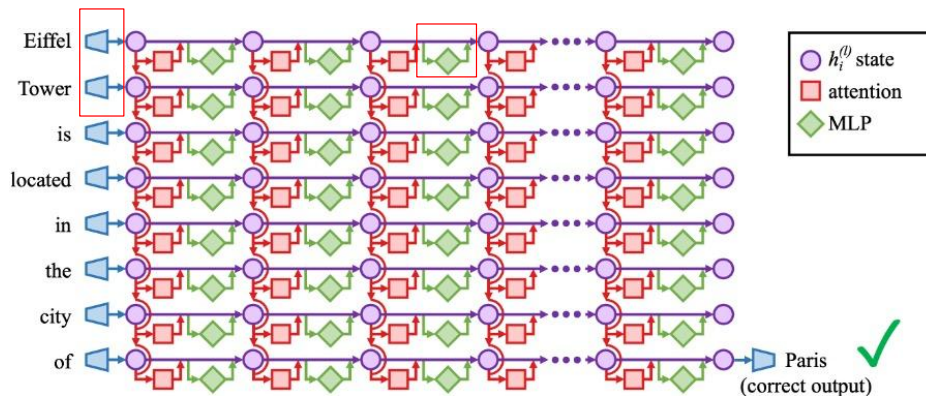
- How to retrieve the locations: `utils.get_act_name(NAME)`
- NAME could be 'embed', 'resid_pre', 'mlp_post', 'attn_out' # don't forget the embedding!

How the Hook Works...

```
hooks = [  
    (utils.get_act_name('embed'), fn),  
    (utils.get_act_name('mlp_post', 2), restore_fn),  
    ...  
]
```

```
logits = run_with_hooks(prompt_tokens, fwd_hooks=hooks)
```

```
# compute your prob based on the logits
```



Functions To-do

We have implemented these for you:

1. `get_target_id()`: get the id of the token (you will need it to get your logit probability)
2. `record_clean_activations()`: provide a record for you to restore the hidden states in the future

You should implement the following methods:

1. `get_restore_fn()`: provide a function that uses the activation record (computed above) to retrieve the activations for the specified token
2. `get_patch_emb_fn()`: provide a function that corrupts the specified span
3. `get_corrupted_probs()`: conduct the **corrupted run**
4. `find_sequence_span()`: find the token indices of the given sequence part
note that tokenization splits 'Needle' into 'Need', '#le'

Functions To-do

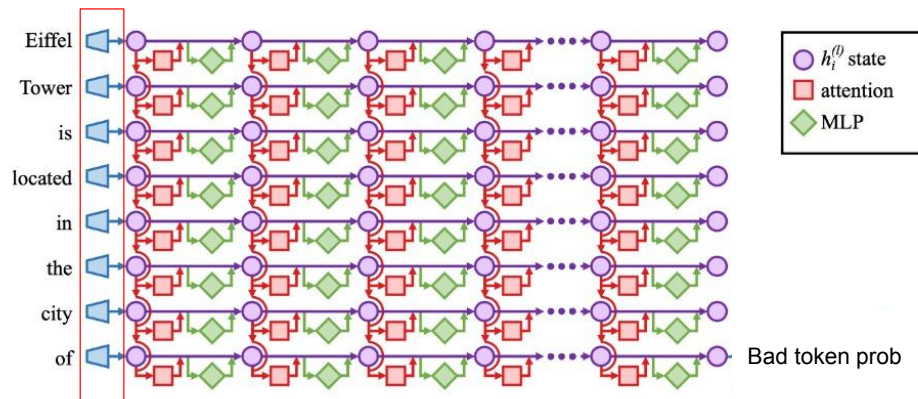
You should implement the following methods:

5. `get_forward_hooks()`: get your hook.
 - Recall: Hooks is `List[Tuple[str, Callable]]`
 - What does this list mean: for each LLM location (described by the str), use the relevant function to do certain computation (either corrupt the embeddings or restore the hidden states)
6. `casual_trace_analysis()`:
 1. Corrupt the model and get the token probability
 2. For each layer and each token in the sequence, get the new probability computed by restoring the specific layer and token
 3. Compute the difference between the restored prob and the corrupted prob
 4. Result: a tensor of shape `(seq_length - 1, layer_num)`
 - # need to remove the start-of-sequence token!

The Complete Process

1. Implement your function to corrupt the embeddings (actually, the function to get that function)
2. Design a hook for the corrupt run
3. Run the corrupt run

Nothing happens inside of the LLM.



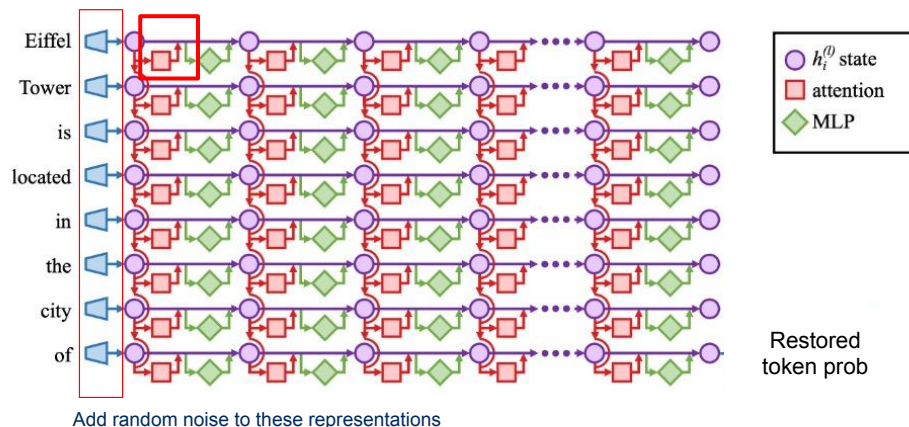
Add random noise to these representations

The Complete Process

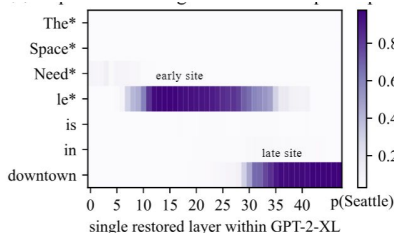
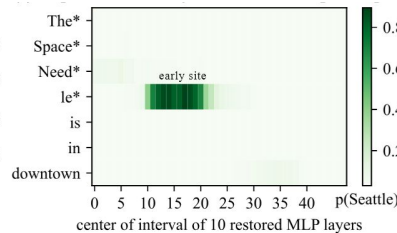
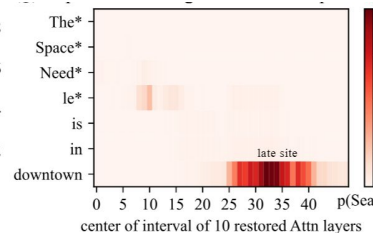
After the corrupted run:

1. Implement your restoration function (actually, the function to get that function)
2. Design a hook for each token index and model layer, and run
3. Compute the probability difference, observe if there is an increase in the final likelihood
4. Form a graph!

run three times: layer, attn, mlp



Restored token prob



GLHF!

- If you have questions...
 - Ask now!
 - Post your questions on piazza (I do monitor the forum frequently)
- Other useful resources:
 - The Official Documentation of TransformerLens
 - <https://rome.baulab.info/>
 - https://stanfordnlp.github.io/pyvene/tutorials/advanced_tutorials/Causal_Tracing.html

Questions?