# Mechanistic Interpretability of Transformers

Alberto Vegara Belmonte

## 1 Introduction

Mechanistic interpretability aims to elucidate the internal workings of machine learning models. A pivotal technique in this domain is activation patching, which involves substituting specific model activations with those from a reference input to assess their impact on the model's output. This method aids in identifying critical activations that influence predictions and in locating potential sources of errors. In this report, we implement activation patching on **minGPT** to analyze how minimal input variations affect the model's predictions. By systematically intervening in the embeddings across different layers and tokens, we generate heatmaps to visualize the significance of each activation.

## 2 Implementation

This section details the implementation process, which closely followed the assignment's prescribed steps, resulting in a straightforward implementation of the required functionality. The modifications primarily focused on augmenting the `GPT` class to enable the storage and manipulation of intermediate embeddings, as well as the logits of the last token. These changes were implemented methodically to ensure compatibility with the existing model structure.

The first step involved defining the attributes of the `GPT` class. Two boolean variables were introduced to control the functionality for saving and patching intermediate embeddings. Additional attributes were added to store the intermediate embeddings and the logits of the last token.

```
self.save_activations_enabled: bool = save_activations_enabled
self.patch_embeddings_enabled: bool = patch_embeddings_enabled
self.saved_activations: list | None = None
self.last_token_logits = None
```

Listing 1.1: GPT Class Attributes

Subsequently, the `forward` method of the model was extended to incorporate the functionality for storing intermediate embeddings and the last token logits. Conditional logic was implemented to ensure that these values are saved only when the corresponding boolean flags are enabled.

```
def forward(self, idx, ..., layer_to_patch=None, embedding_to_patch=None):
    # ... code from minGPT

    # Save the input embedding activations if required
    if self.save_activations_enabled:
        self.saved_activations = []

    for layer_idx, block in enumerate(self.transformer.h):
        # Save the activations for the current layer
```

```
        if self.save_activations_enabled:
            self.saved_activations.append(x.detach().clone())

        # Patch the activations for the specified layer and position
        if (
                self.patch_embeddings_enabled
                and layer_idx == layer_to_patch
                and layer_to_patch is not None
                and embedding_to_patch is not None
        ):
            x[0][embedding_to_patch] = self.saved_activations[
                layer_to_patch][0][embedding_to_patch]

        x = block(x)

    # ... code from minGPT

    # Save the logits of the last token
    self.last_token_logits = logits[:, -1, :]  # (b, vocab_size)

    # ... code from minGPT
```

Listing 1.2: Forward Method

The given code performs an iterative process over the layers of a transformer model, implementing specific operations for saving and manipulating layer-wise activations. At the start of each iteration, the activations of the current layer are preserved if the activation-saving functionality is enabled. This is achieved by storing a detached copy of the current activation state, ensuring that the saved values remain independent of any subsequent computational graph modifications.

Furthermore, the code allows for intervention at a designated layer and position within the transformer, effectively patching the activations. When the indexes criteria are met, the activation at the specified position within the designated layer is replaced by the corresponding saved activation from a previous iteration. This is realized through the assignment:

$$x[0][\text{embedding\_to\_patch}] = \text{self.saved\_activations}[\text{layer\_to\_patch}][0][\text{embedding\_to\_patch}]$$

Here, $x[0][\text{embedding\_to\_patch}]$ represents the activation at the specified embedding position within the current input. It is overwritten by the saved activation corresponding to the same position within the specified layer.

Finally, the modified activations are processed through the current transformer layer, and the iteration continues. This approach provides a controlled mechanism for both analyzing and intervening in the layer-wise dynamics of the transformer.

Pipeline code systematically evaluates how activations are patched across different layers and embeddings in the model when passing clean and corrupted prompts.

First, it initializes a heatmap to store the results for each layer and embedding position. Using the generate function, it processes the clean prompt without activation patching to retrieve baseline logits and tokens. Activation patching is then enabled, and corrupted prompts are passed through the model for each layer and embedding position.

For each test case, the token indices corresponding to the target tokens (test.token_1 and test.token_2) are retrieved using the tokenizer. The logits of these tokens are then extracted, and their difference is calculated. This difference is stored in the heatmap, capturing the impact of activation patching for each layer and embedding position. Lastly, the results are visualized using a heatmap, with layers as rows and embedding positions as columns.

```
    experiments = [
        Experiment(
            clean_text="Michelle Jones was a top-notch student. Michelle",
            corrupted_text="Michelle Smith was a top-notch student.
                Michelle",
            token_1=" Jones",
            token_2=" Smith"
        )
    ]

    number_of_layers = 12
    for experiment in experiments:
        input_length = tokenizer(experiment.clean_text).shape[-1]

        # Run model over the clean text to save the activations
        y1, tokens1, last_token_logits1 = generate(prompt=experiment.
            clean_text)

        index = tokenizer(experiment.token_1).item()
        logit_of_jones = last_token_logits1[0][index]

        probabilities = torch.softmax(last_token_logits1.squeeze(), dim=-1)

        top_probs, top_indices = torch.topk(probabilities, k=10)
        top_tokens = [(tokenizer.decode(torch.tensor([idx])), prob.item())
            for idx, prob in zip(top_indices, top_probs)]

        for token, prob in top_tokens:
            print(f"Token: '{token}', Probability: {prob:.4f}")

        heatmap = np.zeros((number_of_layers, input_length))
        model.save_activations_enabled = False
        model.patch_embeddings_enabled = True

        for i in range(number_of_layers):
            for j in range(input_length):

                # Run model over the corrupted text
                # and inject the activations from the clean run
                y, tokens, last_token_logits = generate(prompt=test.
                    corrupted_text, layer_to_patch=i, embedding_to_patch=j)

                index_token_1 = tokenizer(test.token_1).item()
                index_token_2 = tokenizer(test.token_2).item()
```

```
                logit_of_token_1 = last_token_logits[0][index_token_1]
                logit_of_token_2 = last_token_logits[0][index_token_2]

                # Compute the difference between the logits of the two
                    tokens
                heatmap[i][j] = logit_of_token_1 - logit_of_token_2

        df = pd.DataFrame(
            heatmap,
            index=[f'layer {i}' for i in range(heatmap.shape[0])],
            columns=[f'(embed {i}) {tokens[i]}' for i in range(heatmap.
                shape[1])]
        )

        cmap = sns.color_palette("Blues", as_cmap=True)

        plt.figure(figsize=(10, 8))
        sns.heatmap(df, annot=False, fmt="f", cmap=cmap, cbar_kws={'label':
             'Magnitude'})

        plt.xticks(rotation=45, ha='right')
        plt.yticks(rotation=0)
        plt.tight_layout()

        plt.show()
```

Listing 1.3: Workflow for Patching Intermediate Embeddings

## 3 Approach

As it has been already said, **activation patching** involves replacing specific internal activations of a neural network with those from a different input and observing the resultant changes in the model's output. This method allows for targeted manipulations to identify the components responsible for particular behaviors or knowledge representations within the model.

The paper [3] was provided by the professor as a foundational reference for the multiple ways of performing activation patching, including denoising and noising approaches. Denoising involves substituting activations from a clean input into a corrupted input to investigate which activations restore the model's output to the clean reference. Conversely, the noising approach involves injecting activations from a corrupted input into a clean input to determine which activations degrade the model's output. Both techniques provide complementary insights into the role of activations in the model's behavior, focusing on either repairing or disrupting the flow of information.

- Clean prompt: "Michelle Jones was a top-notch student. Michelle"
- Corrupted prompt: "Michelle Smith was a top-notch student. Michelle"

By running the model with the clean prompt and recording its activations, and then systematically replacing activations in the corrupted prompt with those from the clean prompt, it becomes

possible to pinpoint specific activations that are crucial for aligning the output with the clean reference.

In addition to denoising, some experiments also explored the noising approach. In this case, activations from the corrupted prompt were injected into the clean prompt to observe how and where the model's output begins to deviate from the original prediction. For instance, by inserting activations from the corrupted prompt into the clean prompt, we can evaluate how specific components introduce errors or alter the model's predictions, shedding light on the model's vulnerability to input perturbations.

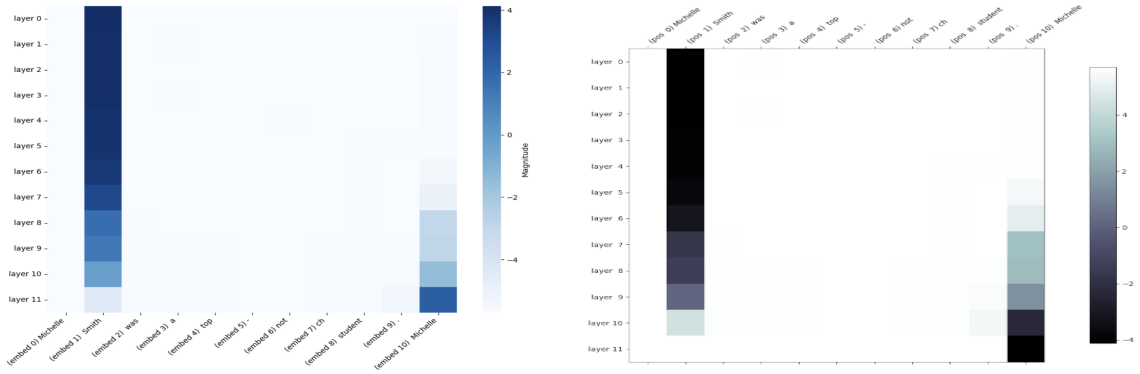
To evaluate the results of the experiments, heatmaps were employed as a visualization tool. These heatmaps allowed for a detailed examination of the contribution of individual layers and token positions to the model's behavior during activation patching. By highlighting the activations with the greatest influence on restoring or disrupting the output, the heatmaps provided a clear and interpretable way to assess the effectiveness of the denoising and noising approaches.

## 4 Results

This section presents the results obtained from the activation patching experiments, focusing on both the expected outputs for the assignment example and more advanced cases inspired by scenarios described in [3]. The goal is to validate the correctness of the implementation, analyze how different types of input corruption affect the model's predictions, and interpret the patterns revealed through activation patching.

### 4.1 Validation Against the Assignment Example

To ensure the implementation behaves as expected, the first experiment reproduces the example provided in the assignment statement. A side-by-side comparison between the implementation results and the expected results confirms that the code correctly executes the denoising approach.




(a) Implementation Result (b) Assignment Expected Result

Fig. 1: Comparison between Implementation Result and Expected Result

### 4.2 Experiment with Single-Token Corruption

After verifying the implementation with the assignment example, more advanced cases were analyzed. In one such experiment, the corruption involves introducing a new token, not present in the clean prompt, to break the reference between tokens. This setup, also suggested in the assignment statement, evaluates the effect of single-token corruption on predictions.

- Clean prompt: "Jessica Jones was a top-notch student. Michelle"
- Corrupted prompt: "Michelle Smith was a top-notch student. Jessica"
- Token to search probability on clean prompt: "Jones"
- Token to search probability on corrupted prompt: "Smith"

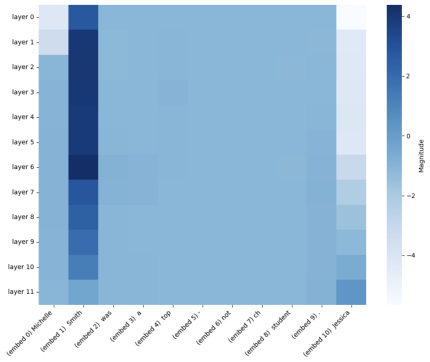



Fig. 2: Impact of Single-Token Corruption

The results showed on 2 indicate, at earlier layers, the corruption of the last token does not immediately disrupt the prediction. However, as the inference progresses, embeddings from the clean prompt begin to interfere, causing corruption. Additionally, the surname tokens ("Jones" and "Smith") also play a significant role, as their alteration directly affects the model's predictions.

### 4.3 Experiment with Multi-Token Corruption

To further investigate the robustness of the model, another experiment was conducted using prompts that differ in more than one token. Unlike the single-token case, this setup introduces more complex disruptions in token relationships, leading to less homogeneous patterns in the heatmaps.

- Clean prompt: "Elon Musk co-founded Tesla and revolutionized the electric car industry. Elon"
- Corrupted prompt: "Steve Jobs co-founded Tesla and revolutionized the electric car industry. Elon"
- Token to search probability on clean prompt: "Musk"
- Token to search probability on corrupted prompt: "Jobs"

As expected, figured 3 showed that multiple tokens are affected by this type of corruption. The heatmaps reveal that all tokens except the final "Elon" token exhibit corrupted activations. Notably, the token "Jobs" displays the most significant impact, as its alteration directly disrupts the prediction for the prompt.

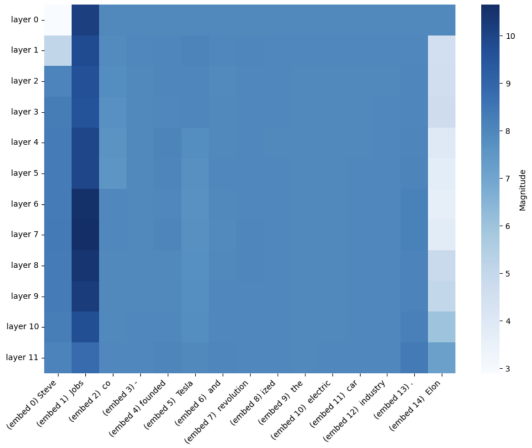



Fig. 3: Impact of Multi-Token Corruption

## 5 Conclusions

This report has explored the application of activation patching techniques to analyze and interpret the behavior of language models when subjected to denoising experiments. The experiments were designed to validate the implementation against the assignment-provided example and to extend the analysis to more complex scenarios involving both single-token and multi-token corruption.

However, due to the length limitations of the report, experiments involving noising approaches could not be carried out as originally planned. This constraint limited the scope of the analysis, particularly with respect to examining the model's behavior under broader noising strategies.

The results demonstrated the effectiveness of activation patching as a tool for identifying and understanding how token relationships evolve across model layers. The implementation successfully reproduced the expected output, confirming its correctness. Subsequent experiments highlighted the sensitivity of the model to token corruption, particularly in cases where key tokens, such as names, were altered or introduced.

A notable observation was the differing impact of single-token and multi-token corruption. While single-token changes primarily affected the immediate context of the altered token, multi-token corruption resulted in more widespread disruption of the model's internal representations. This highlights the importance of contextual relationships in the model's processing and the cascading nature of corruption when multiple tokens are altered.

## References

1. Attribution Patching: Activation Patching At Industrial Scale
   `https://www.neelnanda.io/mechanistic-interpretability/attribution-patching`
2. How-to Transformer Mechanistic Interpretability—in 50 lines of code or less
   `https://www.lesswrong.com/posts/hnzHrdqn3nrjveayv/how-to-transformer-mechanistic-interpretability-in-50-lines`
3. How to use and interpret activation patching
   `https://arxiv.org/abs/2404.15255`