

Documentación: trace_extractor.py

Descripción General

trace_extractor.py es un script de extracción de trazas de activaciones (hidden states) y atenciones (attention weights) de modelos de lenguaje tipo Transformer. Está diseñado específicamente para la detección de alucinaciones mediante el análisis de la estructura interna del modelo durante la generación de respuestas.

Objetivo del Proyecto

Este script forma parte de un proyecto de investigación que combina y extiende metodologías de papers como **CHARM** y **HalluShift** para detectar alucinaciones en LLMs. La innovación principal consiste en:

1. **Modelar la generación como un grafo:** Cada token es un nodo, las atenciones son arcos ponderados.
 2. **Capturar el estado final completo:** En lugar de analizar paso a paso, se toma una “foto final” después de generar toda la respuesta.
 3. **Incluir interacciones prompt-respuesta:** Las atenciones y activaciones incluyen tanto el prompt como la respuesta generada, permitiendo analizar cómo el modelo usa el contexto.
-

Arquitectura y Flujo de Trabajo

1. Inicialización del Modelo

El script soporta modelos de Hugging Face con cuantización automática:

- **Pre-cuantizados** (e.g., unsloth/Meta-Llama-3.1-8B-Instruct-bnb-4bit): Se cargan directamente.
- **Sin cuantizar:** Se aplica cuantización de 4-bit mediante BitsAndBytes para reducir el uso de VRAM.

Parámetros clave de cuantización:

```
BitsAndBytesConfig(  
    load_in_4bit=True, # Reduce memoria ~75%  
    bnb_4bit_compute_dtype=torch.float16, # Mantiene precisión en cálculos  
    bnb_4bit_quant_type="nf4" # NormalFloat4: óptimo para LLMs  
)
```

2. Carga de Datasets

Soporta dos datasets de benchmarking:

Dataset	Split	Campo ID	Campo Pregunta	Uso
TriviaQA	validation (rc.nocontext)	question_id	question	Pre-guntas fac-tuales con re-spuestas conocidas
TruthfulQA	validation (genera-tion)	Índice	question	Pre-guntas dis-eñadas para inducir alu-cina-ciones

3. Extracción de Trazas

Método: extract_activations_and_attentions()

Este es el núcleo del script. Realiza lo siguiente:

A. Construcción del Prompt

```
prompt_text = f"Answer the question concisely in one sentence.\n\nQuestion:  
{question}\nAnswer:"
```

B. Generación con Beam Search

```
generation_output = model.generate(  
    num_beams=5,                      # Búsqueda de haz para diversidad  
    do_sample=False,                   # Generación determinista  
    max_new_tokens=64,                 # Límite de tokens  
    early_stopping=True,               # Detener en EOS  
    return_dict_in_generate=True,  
    output_attentions=True,           # ✓ Capturar matrices de atención  
    output_hidden_states=True # ✓ Capturar activaciones  
)
```

C. Extracción del Estado Final

En lugar de guardar estados intermedios (token 0, token 1, ..., token N), se captura **solo el estado final** que contiene toda la información de la secuencia completa:

```
final_step_idx = len(generation_output.hidden_states) - 1  
seq_len_total = len(prompt) + len(respuesta)
```

Hidden States (activaciones por capa): - **Shape**: [seq_len_total, hidden_dim] - **Contenido**: Activaciones de TODOS los tokens (prompt + respuesta) después del último forward pass. - **Por qué es suficiente**: En Transformers, cada token tiene acceso a todos los anteriores mediante atención. El estado final refleja la composición completa.

Attentions (pesos de atención por capa): - **Shape**: [num_heads, seq_len_total, seq_len_total] - **Contenido**: Matriz completa de atención donde attn[i, j] indica cuánto atiende el token i al token j. - **Incluye**: - Atenciones prompt → prompt - Atenciones respuesta → prompt (uso del contexto) - Atenciones respuesta → respuesta (coherencia interna)

D. Tokens Completos

Para consistencia con las activaciones y atenciones, se guardan TODOS los tokens:

```
all_tokens = generation_output.sequences[0, :seq_len_total] # IDs
tokens_decoded = [tokenizer.decode([tid]) for tid in all_tokens] # Strings
```

Razón: Los índices de hidden_states[layer][i] y attentions[layer][:, i, j] corresponden a posiciones en tokens[i]. Incluir solo los tokens de la respuesta rompería esta correspondencia.

Estructura de Datos Guardada

Cada trace (ejemplo procesado) contiene:

```
{
    'question_id': str,           # ID único del dataset (e.g., "tc_123")
    'generated_answer_clean': str, # Respuesta generada (solo texto, sin
                                   # prompt)
    'hidden_states': List[ndarray], # [num_layers] cada uno: [seq_len_total,
                                   # hidden_dim]
    'attentions': List[ndarray],   # [num_layers] cada uno: [num_heads,
                                   # seq_len_total, seq_len_total]
    'tokens': ndarray,            # [seq_len_total] IDs de tokens (prompt +
                                   # respuesta)
    'tokens_decoded': List[str]   # [seq_len_total] Tokens como strings
}
```

Ejemplo de dimensiones (Llama-2-7B):

- **Modelo**: 32 capas, 32 cabezas de atención, dim=4096
- **Secuencia**: 50 tokens de prompt + 20 tokens de respuesta = 70 tokens totales

```
trace['hidden_states'][0].shape # (70, 4096)
trace['attentions'][0].shape   # (32, 70, 70)
trace['tokens'].shape         # (70, )
len(trace['tokens_decoded']) # 70
```

Gestión de Memoria: Sistema de Batches

Dado que cada trace puede pesar ~130 MB, procesar miles de ejemplos requiere estrategia:

Parámetros

- **BATCH_SIZE**: 500 traces por archivo
- **Memoria estimada**: $500 \times 130 \text{ MB} \approx 65 \text{ GB} \rightarrow$ compresión pickle reduce a ~5 GB/batch

Flujo

1. Procesar ejemplos secuencialmente
2. Acumular en lista `current_batch`
3. Al alcanzar 500 traces:
 - Guardar como `{modelo}_{dataset}_batch_{num:04d}.pkl`
 - Limpiar lista y liberar memoria (`gc.collect()`)
4. Repetir hasta terminar el dataset

Nomenclatura de Archivos

`llama2_chat_7B_triviaqa_batch_0000.pkl`
`llama2_chat_7B_triviaqa_batch_0001.pkl`
`llama2_chat_7B_truthfulqa_batch_0000.pkl`

Uso del Script

Argumentos de Línea de Comandos

```
python trace_extractor.py \
--model llama2_chat_7B \
--dataset triviaqa \
--num-samples 1000
```

Argumento	Tipo	Default	Descripción
<code>--model</code>	str	<code>llama2_chat_7B</code>	ID del modelo (ver HF_NAMES)
<code>--dataset</code>	str	<code>triviaqa</code>	Dataset: <code>triviaqa</code> o <code>truthfulqa</code>
<code>--num-samples</code>	int	<code>None</code>	Número de muestras (None = todas)

Modelos Soportados (HF_NAMES)

Modificar el diccionario `HF_NAMES` para agregar modelos:

```
HF_NAMES = {
    'qwen_2.5_6B': '__', # Placeholder
    'llama2_chat_7B': 'meta-llama/Llama-2-7b-chat-hf',
}
```

Consideraciones Técnicas

1. ¿Por qué solo el estado final?

En modelos autoregresivos Transformer, generar token t requiere: 1. Codificar prompt + tokens $[0, \dots, t-1]$ 2. Aplicar atención causal (solo tokens pasados) 3. Predecir token t

Al final de la generación, el último forward pass contiene: - Activaciones de TODOS los tokens procesados - Atenciones acumuladas de toda la secuencia

No se necesitan estados intermedios porque el grafo final ya refleja todas las interacciones. Guardarlos solo aumentaría el costo de almacenamiento sin aportar información adicional para el análisis estructural.

2. Prompt vs. Respuesta en las Trazas

Componente	Incluye Prompt	Razón
hidden_states	✓ Sí	Necesario para analizar cómo las capas procesan contexto
attentions	✓ Sí	Crítico: Las atenciones respuesta→prompt revelan uso del contexto
tokens	✓ Sí	Consistencia: índices corresponden a posiciones en activaciones
tokens_decoded	✓ Sí	Visualización: etiquetas de nodos en el grafo
generated_answer_clean	✗ No	Solo la respuesta para validación/evaluación

3. Detección de Alucinaciones mediante Grafos

Cada capa genera un grafo: - **Nodos:** Tokens (prompt + respuesta) - **Atributos de nodo:** Activaciones `hidden_states[layer][token_idx]` - **Arcos:** Pesos de atención `attentions[layer][head, i, j]`

Hipótesis del proyecto: - Respuestas alucinadas muestran **menor atención al contexto** (arcos débiles respuesta→prompt) - **Patrones anómalos** en la evolución de activaciones entre capas - **Estructuras topológicas** distintivas (e.g., clustering, centralidad)

Salida del Script

Durante la Ejecución

```
Cargando modelo: meta-llama/Llama-2-7b-chat-hf
⚙️ Aplicando cuantización de 4-bit con BitsAndBytes...
Número de capas del modelo: 32
```

```
Cargando dataset triviaqa...
Número de muestras a procesar: 5000
Tamaño de batch: 500 traces por archivo
Archivos esperados: 10
```

```
--- Ejemplo 0 (Batch actual: 1/500) ---
```

```
Question ID: tc_123
Pregunta: What is the capital of France?...
Respuesta limpia: The capital of France is Paris....
Total de tokens (prompt + respuesta): 68
Primeros 5 tokens decodificados: ['Answer', ' the', ' question', ' con',
'cis']...
Últimos 5 tokens decodificados: [' is', ' Paris', '.', '</s>']...
```

```
💾 Guardando batch 0 en llama2_chat_7B_triviaqa_batch_0000.pkl...
✓ Batch 0 guardado: 500 traces, 5234.56 MB
```

Resumen Final

```
✓ PROCESO COMPLETADO
```

```
Total de ejemplos procesados: 5000
Total de errores: 2
Total de batches guardados: 10
Directorio de salida: /path/to/traces_data
```

```
📁 Archivos generados:
• llama2_chat_7B_triviaqa_batch_0000.pkl: 5234.56 MB
• llama2_chat_7B_triviaqa_batch_0001.pkl: 5198.23 MB
...
```

```
💾 Tamaño total en disco: 52134.45 MB (50.91 GB)
```

```
--- Análisis del primer batch ---
```

```
Estructura de cada trace:
```

```
- Campos guardados: ['question_id', 'generated_answer_clean',
'hidden_states', 'attentions', 'tokens', 'tokens_decoded']
- Número de capas: 32
- Total tokens en secuencia completa: 68
- Shape de hidden state (capa 0): (68, 4096)
  → seq_len=68 (prompt + respuesta), hidden_dim=4096
- Shape de attention (capa 0): (32, 68, 68)
  → num_heads=32, seq_len=68x68
```

Optimizaciones y Limitaciones

Optimizaciones Implementadas

1. **Cuantización 4-bit**: Reduce VRAM de ~28 GB a ~7 GB (Llama-2-7B)
2. **Batching**: Evita OOM al procesar datasets grandes
3. **Garbage collection**: Liberación explícita de memoria entre batches
4. **Estado final único**: Ahorra ~90% de almacenamiento vs. guardar todos los pasos

Limitaciones

1. **Tamaño de archivos**: Batches de 5 GB requieren almacenamiento significativo
2. **Tiempo de procesamiento**: Beam search con num_beams=5 es lento (~2-5 seg/ejemplo)
3. **VRAM**: Requiere GPU de al menos 8 GB para modelos 7B cuantizados

4. **Tokens largos:** Secuencias >512 tokens pueden exceder memoria
-

Siguiente Paso: Construcción de Grafos

Los datos guardados permiten:

1. **Cargar batches:**

```
import pickle
with open('batch_0000.pkl', 'rb') as f:
    traces = pickle.load(f)
```

2. **Construir grafo por capa:**

```
import networkx as nx

trace = traces[0]
layer = 15 # Capa intermedia

G = nx.DiGraph()

# Nodos con activaciones
for i, token in enumerate(trace['tokens_decoded']):
    G.add_node(i,
               label=token,
               features=trace['hidden_states'][layer][i])

# Arcos con atenciones (promedio sobre cabezas)
attn = trace['attentions'][layer].mean(axis=0) # [seq_len, seq_len]
for i in range(len(attn)):
    for j in range(len(attn)):
        if attn[i, j] > 0.01: # Umbral
            G.add_edge(j, i, weight[attn[i, j]])
```

3. **Análisis GML:**

- Graph Neural Networks (GCN, GAT)
 - Métricas topológicas (betweenness, PageRank)
 - Comparación grafos alucinación vs. verdad
-

Dependencias

```
torch>=2.0.0
transformers>=4.35.0
datasets>=2.14.0
bitsandbytes>=0.41.0
accelerate>=0.24.0
huggingface_hub>=0.17.0
numpy>=1.24.0
tqdm>=4.65.0
```

Referencias

- **CHARM:** Chen et al., “Characterizing Hallucination in LLMs via Attention Mechanisms”

- **HalluShift**: Wang et al., “Detecting Hallucinations through Attention Pattern Shifts”
 - **Transformers**: Vaswani et al., “Attention Is All You Need” (2017)
 - **BitsAndBytes**: Dettmers et al., “LLM.int8(): 8-bit Matrix Multiplication for Transformers” (2022)
-

Autor y Licencia

Proyecto: Detección de Alucinaciones con Graph Machine Learning

Curso: IIC3641 - Universidad Santa María

Año: 2025

Este código es parte de un proyecto de investigación académica.