UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

FORMALIZACIÓN ENFOCADA A OPERACIONES DE TENSORES PYTORCH

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JORGE ANDRÉS CRUCES ORTIZ

PROFESOR GUÍA:
MATÍAS TORO I.
PROFESOR GUÍA 2:
ERIC TANTER

MIEMBROS DE LA COMISIÓN:
MATÍAS TORO I.
ÉRIC TANTER
FABIÁN VILLENA R.
VALENTIN BARRIERE

SANTIAGO DE CHILE
2025

Resumen

Este trabajo aborda el problema de los errores relacionados con la incompatibilidad de dimensiones en operaciones tensoriales dentro de programas desarrollados en PyTorch, una problemática frecuente en el contexto del aprendizaje profundo. Como primer paso, se utilizó la herramienta TEOSC para analizar 11 proyectos reales, abarcando un total de 4.427 archivos. Este análisis permitió identificar un conjunto representativo de 15 operaciones fundamentales, que constituyen el núcleo básico de la manipulación tensorial en PyTorch y definieron el alcance del trabajo.

A continuación, se realizó una revisión de la literatura orientada a explorar enfoques previos en verificación de tipos, control de dimensiones y formalización de operaciones tensoriales. Esta revisión permitió sentar las bases conceptuales de lo que se buscaba como solución, aportando aprendizajes claves que guiaron las decisiones de diseño e implementación adoptadas posteriormente. En particular, facilitó la delimitación del enfoque del trabajo, destacando la importancia de una formalización clara y que permitiera una fácil implementación futura.

Luego, se procedió al diseño y desarrollo de una gramática formal complementada con un conjunto de restricciones que describen el comportamiento correcto de las operaciones tensoriales identificadas. Esta gramática permite estructurar y formalizar las operaciones, facilitando su análisis, extensión y adaptación futura. La gramática formal desarrollada, captura las formas de los tensores de entrada y salida, junto con las restricciones asociadas a cada operación, constituye un aporte hacia la construcción de herramientas automáticas capaces de verificar la compatibilidad dimensional en operaciones tensoriales.

Como etapa final, se realizó una evaluación práctica para validar la aplicabilidad de la gramática formal. Se implementó un ejemplo en Python utilizando el solver Z3, centrado en la operación `reshape`, con el objetivo de comprobar que las restricciones pueden traducirse directamente a un sistema de verificación simbólica. Aunque se utilizaron tensores con dimensiones explícitas y se simplificaron ciertas restricciones con fines ilustrativos, los resultados obtenidos respaldan la viabilidad del enfoque propuesto.

Como conclusión, el trabajo desarrollado entrega un modelo formal que permite establecer las restricciones esenciales en operaciones tensoriales, basado en un enfoque de tipos y diseñado para integrarse directamente en SMT solvers u otras herramientas existentes. Este aporte sienta las bases para líneas de trabajo futuras, como la incorporación de gradual typing, el soporte a operaciones más complejas y el desarrollo de aplicaciones prácticas que hagan uso directo de esta formalización.

A mi familia, amigos, a mis profesores guías y a mi.

Agradecimientos

A todos aquellos que me acompañaron en este proceso: mi familia por su apoyo incondicional, mis amigos por estar ahí en los momentos difíciles, y especialmente al café y mate que mantuvieron mi energía durante las largas noches de trabajo.

Agradezco a Ho Young Jhoo de Seoul National University por su valioso aporte al resolver dudas relacionadas con el paper *A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code* [7]. También extiendo mi agradecimiento al profesor Jens Palsberg y a Zeina Migeed, ambos de UCLA (y Zeina además afiliada a Meta), por su disposición y claridad al responder preguntas sobre el trabajo *Generalizing Shape Analysis with Gradual Types* [12].

Finalmente, un agradecimiento a los profesores de mi facultad, al profesor Felipe Bravo por proporcionar material de estudio del curso de Procesamiento de Lenguaje Natural, y especialmente a mis profesores guías por sus reuniones semanales, al profesor Matías Toro por sus innumerables revisiones, correcciones y paciencia durante el proceso, sin cuya guía esta memoria no sería posible.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

En los últimos años se ha observado un auge en el área de inteligencia artificial, particularmente en campos como el procesamiento de lenguaje natural, análisis de imágenes, generación de videos y el desarrollo de modelos de lenguaje que evolucionan constantemente. La importancia de estas tecnologías ha aumentado significativamente debido al crecimiento exponencial de datos textuales, los avances en las técnicas de aprendizaje profundo y su aplicación en herramientas comerciales como asistentes virtuales, traducción automática y análisis de sentimientos. Hoy, la inteligencia artificial ha dejado de ser una promesa lejana para convertirse en una realidad cotidiana que transforma nuestra forma de interactuar con la tecnología.

En este contexto, el lenguaje de programación Python ha adquirido un papel esencial gracias a su simplicidad, versatilidad y a la sólida integración con frameworks de aprendizaje profundo como TensorFlow [1] y PyTorch [13], que permiten a los desarrolladores implementar y experimentar con modelos de manera sencilla y eficiente. Su amplia adopción en la comunidad científica y en la industria ha consolidado a Python como el lenguaje predilecto para proyectos de inteligencia artificial, impulsando tanto la investigación como el desarrollo de aplicaciones prácticas que acercan los avances del aprendizaje profundo a diversos sectores.

Asimismo, un elemento clave en la implementación de proyectos de aprendizaje profundo son los tensores, estructuras de datos multidimensionales que generalizan escalares, vectores y matrices. Los tensores constituyen el núcleo de los cálculos en este campo, ya que permiten representar y manipular eficientemente datos complejos, como imágenes, secuencias de texto y más. Su capacidad para conservar la información dimensional y facilitar operaciones vectorizadas los convierte en herramientas indispensables para desarrollar algoritmos de aprendizaje automático.

En Python, especialmente al trabajar con frameworks como TensorFlow y PyTorch, gestionar correctamente las dimensiones de los tensores es fundamental. Durante el desarrollo de modelos, las operaciones con tensores ocurren constantemente y requieren que sus dimensiones estén alineadas para evitar errores y asegurar que los modelos funcionen correctamente. Un manejo adecuado de estas dimensiones también optimiza el rendimiento y facilita la implementación de arquitecturas complejas.

Sin embargo, la naturaleza dinámica de Python, que permite una gran flexibilidad y escritura concisa de código, también facilita la aparición de errores relacionados con las operaciones con tensores. A diferencia de los lenguajes estáticamente tipados, Python no realiza comprobaciones rigurosas de tipos ni de dimensiones antes de la ejecución, lo que deja espacio para que errores de alineación de tensores se manifiesten solo en tiempo de ejecución. Esto resulta especialmente problemático en proyectos de aprendizaje profundo, donde una incorrecta compatibilidad de dimensiones puede pasar desapercibida hasta que el modelo arroje un error, a menudo después de un largo tiempo de ejecución, complicando así la depuración y la resolución de problemas.

Es por estas razones que en esta memoria se plantea desarrollar una formalización que permita implementar a futuro una herramienta para ayudar a verificar la compatibilidad de dimensiones de los tensores previo a la ejecución de un programa. Este documento se centra específicamente en el framework PyTorch, que actualmente representa el estándar de facto en la comunidad de aprendizaje profundo. PyTorch ha emergido como la plataforma dominante gracias a su diseño intuitivo, su adopción masiva en investigación académica y su integración nativa con Python, lo que lo convierte en la elección natural para este tipo de análisis.

Para abordar de manera sistemática el problema de verificación de dimensiones en operaciones tensoriales, se han establecido los siguientes objetivos que guiarán el desarrollo de esta memoria:

1.1. Objetivos

Objetivo General

Desarrollar una formalización para las operaciones tensoriales más utilizadas en proyectos de deep learning basados en PyTorch, con el fin de verificar automáticamente la coherencia dimensional en programas que combinan múltiples operaciones tensoriales.

Objetivos especificos

1. **Desarrollar una herramienta para el análisis de operaciones con tensores en proyectos de deep learning con PyTorch.**

   Dicha herramienta debe permitir identificar las operaciones realizadas con tensores, enfocándose en las más comunes y proporcionando información sobre su frecuencia de uso en repositorios de distintos ámbitos. Su aplicación sobre un conjunto de proyectos recopilados debe permitir obtener un conjunto representativo de operaciones frecuentes utilizadas en diferentes contextos del desarrollo con PyTorch.

2. **Evaluar herramientas y literatura actual del problema.**

   Analizar las soluciones existentes y los enfoques presentes en la literatura relacionados con el análisis de dimensiones en tensores mediante sistemas de tipos u otras técnicas formales. Esta evaluación tiene como objetivo identificar los métodos más relevantes y

sus limitaciones, estableciendo una base sólida para avanzar hacia una formalización propia del problema.

3. **Formalizar las restricciones de dimensionalidad en las operaciones más importantes de tensores.**
   Desarrollar una formalización rigurosa de las restricciones de dimensionalidad asociadas a las operaciones más utilizadas con tensores en proyectos de deep learning con PyTorch. Esta formalización servirá como base teórica sólida para una futura implementación de herramientas capaces de verificar automáticamente la coherencia dimensional en programas que combinan múltiples operaciones tensoriales, contribuyendo así a prevenir errores y mejorar la fiabilidad del desarrollo.

1.2. Metodología

Para alcanzar los objetivos propuestos, se ha ejecutado una metodología estructurada que permitió un desarrollo sistemático y progresivo. Esta metodología se organizó en tres categorías principales que corresponden a los próximos capítulos.

1. **Recolección de operaciones**
   Identificar las operaciones tensoriales más frecuentes en proyectos de deep learning desarrollados en Python utilizando PyTorch. Para ello, se empleó una herramienta propia denominada TEOSC. La recolección se basó en el estudio de 11 repositorios representativos del estado del arte, abarcando diversos dominios como modelos de lenguaje, frameworks, recursos académicos, visión computacional, modelos multimodales e inteligencia artificial generativa.

2. **Revisión de la literatura y herramientas existentes**
   Explorar el estado del arte en el análisis de dimensiones tensoriales, con el objetivo de comprender cómo se ha abordado este problema previamente y cuáles son los métodos y enfoques más relevantes. Para ello, se revisó la literatura especializada y se evaluaron herramientas actuales como Pythia, Ariadne y PyTea, analizando sus capacidades, limitaciones y el alcance de sus soluciones. Esta revisión permite establecer una base sólida de conocimiento desde la cual desarrollar una formalización propia fundamentada en enfoques existentes y buenas prácticas consolidadas.

3. **Gramática formal**
   Diseñar una gramática formal basada en las operaciones tensoriales más frecuentes identificadas previamente. Para ello, se recopiló una lista de errores comunes asociados a cada operación y, a partir de esta, se extrajeron las restricciones que definen los límites y condiciones bajo los cuales cada operación puede ejecutarse correctamente. Estas restricciones, junto con la literatura revisada y el conocimiento adquirido en las etapas anteriores, sirvieron de base para la construcción de la gramática formal. Dicha gramática describe la sintaxis de las operaciones e incorpora un sistema de tipos que permite validar la corrección de las expresiones. De esta manera, se proporciona un marco sólido para la verificación sistemática de dimensiones, facilitando la detección anticipada de inconsistencias antes de la ejecución de un programa.

4. **Evaluación**

   Evaluar la aplicabilidad práctica de la gramática formal mediante su implementación en un ejemplo concreto. Para ello, se desarrolló un caso de estudio en Python utilizando el módulo oficial de Z3, enfocado en la operación `reshape`. Este ejemplo permite verificar que las restricciones definidas en la gramática pueden trasladarse directamente a un sistema de verificación simbólica. Se incluyen tres casos que representan distintos escenarios de validación, lo que facilita la observación empírica del comportamiento de las restricciones. Esta prueba de concepto evidencia la utilidad de la gramática como base para construir sistemas de verificación estática de dimensiones.

1.3. Definición de Conceptos

En esta sección se presenta una breve introducción a los conceptos que serán utilizados a lo largo de este trabajo de título, con el objetivo de aclararlos y facilitar su comprensión para un mejor entendimiento del documento.

Tensor

Un tensor es una estructura matemática que generaliza los conceptos de escalares (números), vectores y matrices a dimensiones superiores. En el ámbito de la computación, un tensor es simplemente un arreglo multidimensional de datos, fundamental para representar y manipular información en aplicaciones de inteligencia artificial y aprendizaje automático. En este trabajo de título, el enfoque principal está en asegurar que sus operaciones no presenten problemas de dimensionalidad.

PyTorch

PyTorch es una biblioteca de código abierto para aprendizaje automático desarrollada por Meta AI. Está enfocada en el cálculo numérico y en el desarrollo de redes neuronales profundas, destacando por su flexibilidad y facilidad de uso. En el contexto de este trabajo de título, PyTorch fue elegido como framework principal por su amplia adopción en la comunidad de investigación, su robustez y su soporte continuo. Su popularidad garantiza acceso a documentación y recursos actualizados. En esta memoria se utiliza la versión 2.7.0 de la documentación oficial.

Gramatica Formal

Una gramática formal es un conjunto de reglas que describe cómo se pueden generar cadenas de un lenguaje. Se utiliza en teoría de lenguajes, compiladores e inteligencia artificial

para definir estructuras válidas y analizar expresiones. El uso de gramáticas formales representa un enfoque común para la creación de un sistema de tipos, con el fin de verificar la dimensionalidad de tensores y garantizar que las operaciones sobre ellos sean coherentes y correctas.

SMT Solvers

Los SMT Solvers (Satisfiability Modulo Theories Solvers) son herramientas que permiten verificar si una fórmula lógica es satisfacible bajo ciertas teorías matemáticas (como aritmética, álgebra o teoría de conjuntos). Se emplean en verificación de software, model checking y resolución de problemas complejos en inteligencia artificial. En este documento se utiliza el SMT Solver Z3, una herramienta ampliamente reconocida y desarrollada por Microsoft Research, para comprobar, a partir de un conjunto de restricciones, si una operación tensorial es válida o inválida, asegurando así la coherencia y corrección de las operaciones sobre tensores.

Capítulo 2

Recolección de Operaciones

Antes de comenzar a hablar de formalización, es fundamental comprender cuáles son las operaciones que se desea formalizar. Este capítulo tiene como objetivo identificar las operaciones de PyTorch más relevantes en la práctica, cuantificar su uso en proyectos reales del estado del arte, y seleccionar aquellas operaciones más importantes para establecer el alcance del trabajo.

En un primer intento, se identificaron las operaciones revisando la documentación oficial de PyTorch y basándose en la experiencia adquirida en proyectos anteriores. No obstante, este enfoque resultó limitado, ya que carecía de un criterio sistemático para determinar qué operaciones debían ser consideradas. Esta falta de delimitación dificultaba establecer una base sólida sobre que operaciones trabajar.

Por esta razón, se reconoció la necesidad de elaborar una lista más precisa y completa de las operaciones a partir de un análisis sistemático de repositorios representativos del estado del arte en proyectos basados en Pytorch, incluyendo tanto proyectos de referencia ampliamente utilizados como un repositorio de carácter académico.

Para alcanzar estos objetivos, se siguieron de manera estructurada los siguientes pasos: (1) selección de repositorios representativos del estado del arte en proyectos de deep learning con PyTorch, (2) desarrollo de una herramienta denominada TEOSC (Tensor Operations Static Counter) para cuantificar operaciones tensoriales, (3) análisis de la frecuencia de uso de dichas operaciones en proyectos reales, y (4) selección de las 15 operaciones más relevantes como base para su posterior formalización.

2.1. Repositorios representativos del estado del arte

A continuación se presentan 11 repositorios representativos del estado del arte en proyectos desarrollados con PyTorch. Cada uno de ellos se acompaña del nombre de la organización o entidad responsable, así como de una breve descripción de su propósito, con el fin de contextualizar el dominio de aplicación al que pertenece. Estos repositorios cubren una amplia gama de áreas, incluyendo modelos de lenguaje, visión computacional, aprendizaje multi-

modal, inteligencia artificial generativa, bibliotecas de desarrollo y recursos académicos. Su análisis permitió identificar y clasificar las operaciones tensoriales más frecuentes en distintos contextos reales de uso de PyTorch.

1. **BERT** [3] (NVIDIA): implementación de referencia de BERT en PyTorch con ejemplos de pre-entrenamiento, fine-tuning e inferencia para tareas de NLP.
2. **Whisper** [15] (OpenAI): modelo de reconocimiento de voz automático multilingüe basado en transformers para transcripción de audio a texto.
3. **LLaMA** [17] (Meta AI): familia de modelos de lenguaje para generación de texto y conversación, base para numerosos modelos derivados.
4. **Transformers** [18] (Hugging Face): biblioteca con modelos pre-entrenados para NLP, visión y audio, incluyendo BERT, GPT, T5, ViT y CLIP.
5. **CC6205** (Universidad de Chile): materiales académicos del curso de Procesamiento de Lenguaje Natural de la Universidad de Chile, que incluyen tareas prácticas desarrolladas entre los años 2020 y 2024. El repositorio presenta técnicas fundamentales de procesamiento de lenguaje natural y contiene implementaciones en código que utilizan PyTorch como framework principal para el desarrollo de modelos.
6. **Detectron2** [19] (Meta AI): framework de detección y segmentación de objetos implementando Mask R-CNN, Faster R-CNN y RetinaNet.
7. **YOLOv5** [8] (Ultralytics): implementación de YOLO para detección de objetos en tiempo real.
8. **Segment Anything** [9] (Meta AI): modelo fundacional para segmentación de imágenes mediante prompts interactivos con arquitectura transformer.
9. **CLIP** [14] (OpenAI): modelo visión-lenguaje que aprende representaciones visuales y textuales.
10. **BLIP** [11] (Salesforce): modelo visión-lenguaje.
11. **Stable Diffusion** [16] (Stability AI): modelo de generación de imágenes por texto.

Una vez definidos y recopilados los repositorios de referencia, el siguiente paso consistió en desarrollar una herramienta que permitiera cuantificar las operaciones tensoriales más frecuentes presentes en estos proyectos, a través del análisis de su código fuente. Con este propósito se creó TEOSC, una herramienta diseñada específicamente para este tipo de análisis.

2.2. TEOSC: Tensor Operations Static Counter

TEOSC, por su nombre en inglés, *Tensor Operations Static Counter*, es una herramienta que permite contar operaciones de tensores de PyTorch dentro de archivos. Se encuentra

disponible en GitHub bajo licencia MIT en TEOSC (Tensor Operations Static Counter) con sus respectivas instrucciones de instalación y uso.

A partir de la documentación oficial de PyTorch 2.7 (versión estable a julio de 2025), se elaboró un archivo de referencia disponible en GitHub bajo el nombre `pytorch_operations.txt`. Este archivo reúne las operaciones de las secciones `tensor` y `nn`, considerando solo aquellas funciones que reciben, modifican o devuelven tensores. Sirve como lista principal para identificar las operaciones que deben ser contadas en los proyectos analizados.

Inicialmente, se consideró un enfoque basado en búsquedas de texto para identificar operaciones de PyTorch, pero esta estrategia resultó poco fiable, ya que podía generar falsos positivos al coincidir con comentarios, cadenas de texto u otros contextos donde los nombres de las funciones aparecen sin ser llamadas reales. Para superar esta limitación, se optó por analizar el árbol de sintaxis abstracta (AST) de los archivos, utilizando el módulo estándar `ast` de Python. A través de funciones como `ast.walk()`, TEOSC recorre la estructura del código, identifica todas las llamadas a funciones (`ast.Call`) y verifica si corresponden a operaciones de PyTorch incluidas en el archivo de referencia.

Este enfoque basado en el AST mejora la precisión del conteo, ya que permite distinguir con mayor fiabilidad las verdaderas llamadas a funciones dentro del código. Sin embargo, no es completamente infalible: algunas operaciones pueden quedar fuera del análisis, por ejemplo, construcciones complejas o llamadas indirectas que no pueden ser resueltas estáticamente. Aun con estas limitaciones, el método permite obtener un recuento de operaciones de manera razonable en proyectos reales.

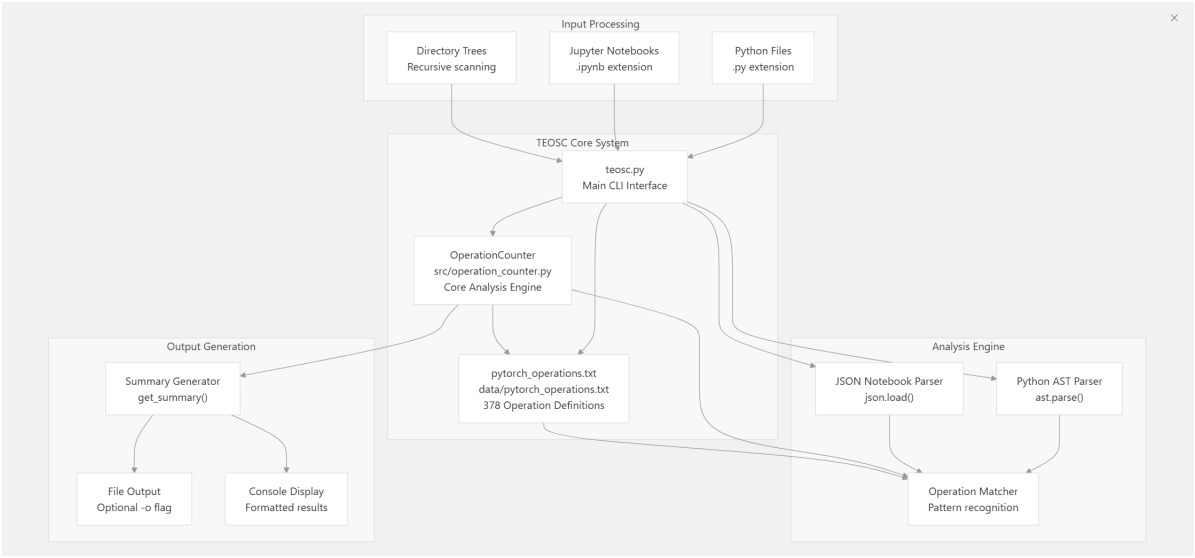

Figura 2.1: Arquitectura general de TEOSC.

El diseño general del sistema, ilustrado en la Figura 2.1, se organiza en una arquitectura dividida en diferentes componentes los cuales se resumen en la Tabla 2.1.

Tabla 2.1: Arquitectura general de TEOSC.

Componente	**Ubicación del Archivo**	**Responsabilidad Principal**
Input Processing	`teosc.py`	Explora recursivamente los directorios o archivos especificados.
Core System	`teosc.py`, `operation_counter.py` y `pytorch_operations.txt`	Coordina todo el proceso: analiza argumentos de entrada, busca archivos a procesar, define las operaciones de PyTorch permitidas y gestiona la ejecución del sistema.
Analysis Engine	`operation_counter.py`	Procesa el código fuente transformándolo en árboles de sintaxis abstracta (AST) para su análisis. Interpreta también el contenido de los Notebooks Jupyter. Su función principal es identificar y contar las operaciones específicas de PyTorch dentro del código analizado.
Output Generation	`teosc.py`	Resume los resultados del análisis y genera los archivos de salida con la información procesada.

Una vez ejecutado sobre un conjunto de archivos o repositorios, TEOSC genera un resumen con el conteo de operaciones de tensores encontradas. El resultado se presenta de forma legible por consola, mostrando los archivos analizados y un desglose ordenado de las funciones de PyTorch detectadas junto con su frecuencia. La siguiente salida es un ejemplo del formato mostrado por consola.

```
[RESULTS] Total Operation Summary:
--------------------------------------------------
Files scanned (3):
  - /path/to/file1.py
  - /path/to/file2.py
  - /path/to/notebook.ipynb
--------------------------------------------------
Operation Count Summary:
tensor: 5
add: 3
matmul: 2
...
--------------------------------------------------
```

Para facilitar su aplicación en contextos diversos, TEOSC ofrece soporte tanto para archivos Python como para notebooks de Jupyter, ambos ampliamente utilizados en proyectos de

deep learning con PyTorch. Además, permite analizar directorios completos de forma recursiva y ofrece la opción de exportar los resultados a un archivo externo. Estas funcionalidades hacen que la herramienta sea adaptable a distintos flujos de trabajo y escenarios de análisis. A continuación, se presentan algunos ejemplos de uso desde la línea de comandos:

```
# Analyze a single Python file
python teosc.py path/to/file.py

# Analyze a Jupyter notebook
python teosc.py path/to/notebook.ipynb

# Analyze an entire directory recursively
python teosc.py path/to/directory

# Save results to a file
python teosc.py path/to/file.py -o results.txt
```

Finalmente, se ejecutó la herramienta TEOSC para analizar los 11 repositorios de forma recursiva, procesando un total de 4,427 archivos distribuidos en los diferentes proyectos. Los resultados completos del análisis están disponibles en `results_repo_analysis.txt`

```
[RESULTS] Total Operation Summary:
------------------------------------------------
Files scanned (4427):
  - ..\ptoc_repos\BERT\bind_pyt.py
  - ..\ptoc_repos\BERT\create_pretraining_data.py
  ...
```

2.3. Resultados del Analisis

A continuación se presentan los resultados del análisis realizado con TEOSC sobre los 11 repositorios representativos del estado del arte. En total se identificaron 185 operaciones diferentes, mostrando cada una la frecuencia con la que ocurren dentro de los repositorios analizados. Se muestran únicamente las 40 operaciones más importantes encontradas en el análisis de 4,427 archivos Python, ordenadas por número de ocurrencias de mayor a menor frecuencia.

Tabla 2.2: Top 40 operaciones tensoriales detectadas y su frecuencia

#	**Operación**	**Frecuencia**
1	`range`	5038
2	`view`	4375
3	`size`	4140
4	`nn.Linear`	3862
5	`tensor`	3300
6	`reshape`	3277
7	`transpose`	3101
8	`cat`	2357
9	`sum`	2146
10	`split`	2059
11	`unsqueeze`	2012
12	`zeros`	1907
13	`arange`	1807
14	`ones`	1643
15	`max`	1570
16	`nn.LayerNorm`	1366
17	`squeeze`	1306
18	`expand`	1288
19	`any`	1284
20	`allclose`	1224
21	`nn.Dropout`	1195
22	`permute`	1152
23	`min`	951
24	`mean`	939
25	`matmul`	852
26	`stack`	786
27	`all`	716
28	`sqrt`	682
29	`flatten`	661
30	`nn.Embedding`	660
31	`where`	595
32	`abs`	553
33	`repeat`	551
34	`nn.Conv2d`	530
35	`clamp`	484
36	`rand`	478
37	`log`	468
38	`einsum`	447
39	`concat`	435
40	`asarray`	422

Finalmente, se eliminaron las operaciones `size`, `allclose` y todas las pertenecientes al módulo `nn` (asociadas a capas de redes neuronales). Esto permitió seleccionar las 15 operaciones de tensores más utilizadas para trabajar en el proceso de formalización. Las operaciones mencionadas fueron descartadas por las siguientes razones:

- `size`: es un método propio del tensor que devuelve su tamaño. Puede retornar un único entero o un objeto de tipo `torch.Size`, que es una tupla de números enteros indicando las dimensiones del tensor. No retorna un tensor como tal, sino información estructural sobre su forma.
- `allclose`: aplica una condición lógica sobre dos tensores y retorna un valor booleano, sin generar o modificar un Tensor
- Operaciones `nn`: requieren un patrón de uso diferente (declaración + uso), como se describe a continuación.

A diferencia de los métodos anteriores, que operan directamente sobre tensores y retornan nuevos tensores o lista de tensores, las operaciones del módulo `torch.nn`, utilizadas comúnmente para definir y aplicar capas de redes neuronales, presentan un patrón de uso diferente al de las operaciones tensoriales elementales. En general, estas requieren una etapa de inicialización previa, seguida de su aplicación sobre un tensor de entrada. El siguiente fragmento de código ilustra este patrón utilizando la función `nn.Linear` la cual aplica una transformación linear a un input:

```
import torch
import torch.nn as nn

m = nn.Linear(20, 30)
input = torch.randn(128, 20)
output = m(input)
print(output.size())
```

Si bien este tipo de operaciones es central en la construcción de modelos en PyTorch, no fueron incluidas en la etapa inicial del proceso de formalización. El enfoque de este trabajo se centró en las operaciones fundamentales sobre tensores, las cuales modifican o generan nuevos tensores más que aplicar una operación sobre un input dado.

Una vez establecida una formalización rigurosa para estas operaciones básicas, la extensión hacia las abstracciones del módulo `torch.nn` se vuelve un paso natural. En este sentido, la documentación oficial de PyTorch para el módulo `nn` resulta especialmente detallada y estructurada, proporcionando especificaciones claras sobre la forma de los tensores de entrada y salida, lo que contrasta con la naturaleza más general de la documentación asociada a las operaciones tensoriales. (véase `torch.nn.Linear`).

2.4. Listado de Operaciones

Finalmente, tras el análisis exhaustivo de los 4,427 archivos procesados por TEOSC, se seleccionaron las siguientes quince operaciones para concentrarse en la formalización:

#	**Operación**	**Breve descripción**
1	`range`	Crea un tensor con una secuencia de enteros en un rango especificado.
2	`view`	Reinterpreta la memoria del tensor con una nueva forma, sin copiar datos. Requiere que la nueva forma sea compatible con la disposición contigua de los datos en memoria.
3	`reshape`	Cambia la forma de un tensor sin alterar sus datos.
4	`transpose`	Intercambia dos dimensiones de un tensor.
5	`cat`	Concatena una lista de tensores a lo largo de una dimensión específica.
6	`sum`	Calcula la suma de los elementos de un tensor a lo largo de dimensiones especificadas.
7	`split`	Divide un tensor en sub-tensores según tamaños o número de secciones.
8	`unsqueeze`	Añade una dimensión de tamaño uno en la posición especificada.
9	`zeros`	Crea un tensor lleno de ceros con la forma dada.
10	`arange`	Similar a `range`, genera un tensor con valores igualmente espaciados en un intervalo.
11	`ones`	Crea un tensor lleno de unos con la forma dada.
12	`max`	Devuelve el valor máximo y, opcionalmente, el índice a lo largo de una dimensión.
13	`squeeze`	Elimina dimensiones de tamaño uno de un tensor.
14	`expand`	Expande un tensor a una nueva forma sin copiar datos, replicando valores según sea necesario.
15	`any`	Devuelve un tensor con booleanos dependiendo si algún elemento del tensor cumple una condición.

Tabla 2.3: Operaciones de PyTorch seleccionadas para la formalización de restricciones

Estas operaciones constituyen las funciones principales para la manipulación y transformación de tensores, siendo las más recurrentes en proyectos basados en PyTorch. Su selección delimita el alcance del trabajo y asegura que el análisis se enfoque en operaciones pertinentes para el estudio de restricciones dimensionales.

Capítulo 3

Revisión de la literatura y herramientas existentes

En esta sección se analizan las principales herramientas desarrolladas para abordar el problema de la manipulación de tensores y la verificación de sus dimensiones. Examinaremos diferentes enfoques metodológicos, desde sistemas de tipos graduales hasta análisis de flujo de datos, identificando las fortalezas y limitaciones de cada propuesta. El objetivo es extraer lo más importante de estos trabajos para construir una base sólida que permita diseñar una solución más robusta y completa. Cada herramienta será evaluada en términos de su capacidad para detectar errores, su integración con ecosistemas existentes, y su aplicabilidad práctica en el desarrollo de aplicaciones de machine learning.

Es importante destacar que esta revisión tiene un propósito estrictamente práctico: destacar los aportes y limitaciones que resultan útiles para esta memoria y, en particular, para el contexto de PyTorch. No pretende emitir juicios de valor sobre los trabajos ni sus autores, sino recopilar la información que pueda servir de base para nuestro enfoque.

3.1. GraTen

GraTen, del paper *Gradual Tensor Shape Checking* [5], presenta un sistema de tipado gradual para detectar inconsistencias de forma en tensores dentro de programas de deep learning. Su propuesta combina inferencia de tipos por mejor esfuerzo con verificación estática y la inserción automática de chequeos dinámicos cuando la información estática no basta. Además, ofrece pruebas formales de seguridad y garantiza la propiedad de gradualidad, todo ello validado en un prototipo implementado sobre OCaml-Torch. Este fue el primer trabajo que revisamos y constituyó la principal motivación para iniciar esta memoria.

Aportes destacados

- **Gramática expresiva:** Para atacar el problema de mismatch de tensores utiliza una gramática capaz de capturar estos problemas.

- **Inferencia por mejor esfuerzo**: deduce tipos y formas sin requerir anotaciones exhaustivas.
- **Chequeo híbrido**: combina comprobación estática con assertions dinámicas insertadas automáticamente.
- **Tipado gradual**: permite refinar progresivamente las garantías sin romper código existente y cumple la gradual guarantee.
- **Prueba formal de seguridad**: cuenta con una demostración formal de seguridad asegurando consistencia de formas de los tensores.

Limitaciones

- El sistema y el prototipo están acotados al ecosistema OCaml–Torch, generalizarlo a otros lenguajes o frameworks requiere trabajo adicional.
- La resolución de restricciones es tan compleja que los autores optan por implementar su propio solucionador heurístico, en lugar de utilizar solucionadores SMT (Satisfiability Modulo Theories) como Z3 [2], que verifican la satisfacibilidad de fórmulas lógicas bajo teorías formales específicas. Esta decisión añade complejidad adicional al sistema.
- La formalización y las notaciones empleadas son densas, lo que implica que replicar la herramienta o los experimentos de manera práctica requeriría un esfuerzo de implementación significativo.
- El prototipo está desarrollado sobre OCaml-Torch, una elección poco común en la comunidad de deep learning, que utiliza predominantemente PyTorch en Python. Esto limita tanto su adopción como su compatibilidad con herramientas y bibliotecas existentes.

3.2. Pythia

Pythia, presentado en el paper de *Static Analysis of Shape in TensorFlow Programs* [10], es una herramienta diseñada específicamente para programas de Python que usan TensorFlow. Su objetivo es rastrear las formas de los tensores a lo largo del código Python, incluyendo llamadas a funciones de la biblioteca TensorFlow. Con este seguimiento, Pythia puede detectar posibles errores de desajuste de formas antes de que el código se ejecute.

Aportes destacados

- **Análisis basado en operaciones matemáticas:** Utiliza fórmulas matemáticas para describir el comportamiento de operaciones centrales de TensorFlow.
- **Integración con WALA:** Aprovecha el framework WALA para construir grafos de código y realizar análisis de flujo de datos.

- **Enfoque práctico:** Se centra en casos de uso reales, dejando de lado casos complicados como tensores con dimensiones desconocidas o reglas de broadcasting.

Limitaciones

- El paper muestra la relación de restricciones en base a algoritmos y funciones, pero no profundiza en aspectos técnicos formales.
- Está limitado al ecosistema TensorFlow, lo que dificulta la adaptación directa a otros frameworks como PyTorch.
- WALA fue originalmente diseñado para JavaScript y Java, lo que puede introducir limitaciones al analizar código Python.
- Se enfoca más en el enfoque práctico que en la formalización rigurosa del sistema.

3.3. Ariadne: Analysis for Machine Learning Programs

Ariadne, del paper *Ariadne: Analysis for Machine Learning Programs* [4], es una herramienta para programas Python que emplean TensorFlow. Aprovecha el framework WALA para construir grafos de flujo de datos, define un sistema de tipos para tensores y ejecuta un análisis que propaga la información de forma a lo largo del programa. El estudio se concentra en operaciones de tipo batch y muestra cómo este enfoque puede detectar errores de forma antes de la ejecución.

Aportes destacados

- **Dataflow detallado:** Utiliza WALA para generar grafos de flujo de datos que rastrean el recorrido de los tensores.
- **Enfoque en operaciones de batch:** Las reglas y casos de estudio se centran en patrones comunes de procesamiento por lotes.

Limitaciones

- **Gramática poco expresiva:** El sistema de tipos captura información limitada y no modela restricciones complejas.
- **Cobertura incompleta de la API:** Sólo se modela un subconjunto reducido de TensorFlow.
- **Prototipo preliminar:** Resultados exploratorios sin formalización rigurosa ni evaluación amplia.

3.4. PyTea

PyTea, presentado en el artículo A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code [7], es uno de los trabajos que más influyó en este trabajo de título. Se enfoca en analizar programas en Python que usan PyTorch. PyTea interpreta las operaciones con tensores y genera restricciones a partir del código, diferenciando entre restricciones fuertes (que siempre deben cumplirse) y débiles (que dependen de ciertas condiciones en la ejecución). Dichas restricciones se resuelven con el solucionador SMT Z3, lo que permite captar relaciones aritméticas complejas entre dimensiones. Para cubrir la vasta API de PyTorch, los autores formalizan sólo un conjunto reducido de primitivas y luego definen el resto de las APIs como composiciones de éstas. El análisis explora de forma simbólica todas las rutas factibles y abstrae bucles típicos del entrenamiento por lotes, logrando así una buena escalabilidad.

Aportes destacados

- **Cobertura PyTorch**: Fue hecho específicamente orientado al ecosistema PyTorch.
- **Gramática expresiva:** Las operaciones de PyTorch se traducen a expresiones cuya sintaxis incorpora directamente las restricciones sobre las formas de los tensores. De este modo, variables, constantes y operadores aritméticos conviven naturalmente con las restricciones en una misma representación.
- **Restricciones fuertes vs. débiles**: Diferencia restricciones obligatorias de aquellas que sólo se activan en ciertas rutas.
- **Integración de Z3**: Emplea un solver SMT industrial de alto rendimiento para optimizar la verificación y resolución de restricciones en el sistema.

Limitaciones

- **Formalización parcial de la API**: En comunicación con la autora, Ho Young Jhoo, se indicó que solo un conjunto esencial de operaciones fue formalizado, mientras que el resto de la API se define mediante composición.
- **Integración limitada**: Existe un plugin de VSCode legado cuya compatibilidad con versiones actuales es incierta.

3.5. Generalizing Shape Analysis with Gradual Types

El trabajo *Generalizing Shape Analysis with Gradual Types* [12] propone un enfoque para el análisis de formas de tensores basado en el uso de tipado gradual. Desarrollado durante una pasantía en Meta por la autora Zeina Migeed y supervisado por el profesor Jens Palsberg, el proyecto introduce el Gradual Tensor Calculus, una gramática expresiva que permite razonar sobre formas de tensores de manera incremental. La propuesta se centra en tres problemas

fundamentales: migración estática de tipos, migración con restricciones aritméticas y eliminación de ramas dependientes de formas. El trabajo integra herramientas nativas de PyTorch como `torch.fx` y utiliza el solver SMT Z3 para resolver las restricciones.

Aportes destacados

- **Gramática expresiva**: Introduce el Gradual Tensor Calculus, un sistema formal que captura relaciones complejas entre dimensiones de tensores de manera más natural que trabajos anteriores.
- **Tipado gradual**: Permite introducir anotaciones de tipo de forma incremental, reduciendo la barrera de entrada para programadores y facilitando la adopción gradual del sistema.
- **Desarrollo industrial**: Desarrollado durante una pasantía en Meta por Zeina Migeed, lo que implica una mayor relevancia práctica y aplicabilidad, considerando que es la misma compañía que desarrolla PyTorch, lo que refuerza su alineación con las necesidades reales del framework.
- **Uso de Z3**: Integra el solver SMT Z3 para resolver restricciones complejas.
- **Referencia a trabajos previos**: Hace referencia explícita al primer paper de GraTen, reconociendo su contribución pionera en el análisis de formas de tensores con tipado gradual.

Limitaciones

- **Acceso limitado**: El código y las herramientas descritos en el artículo no están disponibles públicamente, lo que limita su reproducibilidad y adopción por parte de la comunidad. Además, el enfoque presentado se restringe a las operaciones `reshape`, `Conv2D` (con parámetros limitados) y `add`, excluyendo otras operaciones comunes en el manejo de tensores.
- **Rigurosidad matemática limitada del sistema de tipos**: Algunos juicios y demostraciones no cumplen con el rigor matemático esperado, recurriendo al lenguaje natural para expresar conceptos que podrían formalizarse. Por ejemplo, en las reglas de tipo de la gramática, se usa lenguaje natural para indicar que "Dyn ocurre exactamente una vez" o "Dyn ocurre más de una vez con al menos una ocurrencia", donde `Dyn` representa la dimensión desconocida de un tensor, en lugar de explicitar estas condiciones de manera formal y precisa.
- **Dependencia de funciones auxiliares**: Utiliza numerosas funciones auxiliares para establecer semántica en lugar de capturarla directamente en el sistema de tipos, lo que puede introducir complejidad adicional y posibles inconsistencias.

3.6. Aprendizajes obtenidos de la revisión de literatura

Tras analizar las herramientas más relevantes de este problema, se pueden extraer varios aprendizajes clave que guiarán el diseño de nuestra propuesta:

- **Necesidad de una gramática formal expresiva**: Contar con una gramática formal que integre de manera directa las restricciones en las expresiones no solo permite representar de forma natural y precisa las relaciones complejas entre dimensiones de tensores, sino que también constituye una estrategia probada y eficaz para abordar este tipo de problema.
- **Balance entre formalización y practicidad**: Las herramientas revisadas muestran una tensión entre la rigurosidad formal (como en GraTen y PyTea) y la aplicabilidad práctica en entornos reales (como en Pythia y Ariadne). Diseñar una solución efectiva requiere equilibrar la formalización matemática, que garantiza robustez, con la facilidad de integración y adopción en flujos de trabajo existentes, especialmente en el contexto de PyTorch, donde la comunidad prioriza la flexibilidad y rapidez de desarrollo.
- **Ventajas del uso de solvers SMT**: La integración de solvers SMT industriales, como Z3, optimiza la verificación precisa y eficiente de restricciones, eliminando la necesidad de soluciones personalizadas.
- **Relevancia del contexto PyTorch**: Un número considerable de herramientas se ha desarrollado con enfoque en TensorFlow, evidenciando la necesidad de herramientas adaptadas al ecosistema PyTorch.

Estos aprendizajes sientan las bases para desarrollar una solución que integre las principales fortalezas de los trabajos analizados, abordando sus limitaciones y adaptándose de manera efectiva al contexto de PyTorch.

Capítulo 4

Gramática formal

4.1. Obtención de restricciones de las operaciones

Una vez revisada la literatura y los trabajos relacionados, el siguiente paso consistió en identificar y sistematizar los requisitos necesarios para el uso correcto de cada operación. Para ello, fue fundamental analizar en detalle cómo se invocan estas funciones, qué condiciones deben cumplir sus argumentos y qué relaciones dimensionales se deben preservar entre los tensores involucrados.

Con este objetivo, se planteó una representación estructurada de cada operación, incluyendo sus parámetros, así como las formas de los tensores de entrada y salida. El primer paso fue establecer una notación precisa que permitiera describir estos elementos de manera formal. Por ejemplo, para la operación `flatten`, se propuso la siguiente notación:

```
flatten :: Tensor ([x_1,..., x_n], dim=n) -> Tensor([x_1 * x_2 * ... * x_n], dim=1)
```

Esta notación sentó las bases para formalizar las operaciones en términos de sus dimensiones. No obstante, la representación inicial solo capturaba una parte del comportamiento esperado de cada operación, por lo que fue necesario profundizar en la identificación de restricciones.

La primera estrategia para identificar las restricciones fue consultar la documentación oficial de PyTorch. Sin embargo, este enfoque resultó poco efectivo, ya que la documentación presenta limitaciones importantes con respecto a las operaciones tensoriales: no describe con suficiente detalle los parámetros de entrada, ni especifica claramente las formas esperadas de los tensores de entrada y salida. Además, omite información sobre los posibles errores de ejecución y no explicita las restricciones que deben cumplir los tensores para que la operación sea válida.

Dado que la documentación oficial de PyTorch no resultó ser una fuente confiable para extraer las restricciones necesarias, fue preciso adoptar un enfoque alternativo de carácter empírico. Este consistió en probar individualmente cada una de las operaciones involucradas, utilizando parámetros en casos bordes, y registrar los errores que se producían en tiempo

de ejecución. Si bien en un principio esta estrategia fue percibida como poco rigurosa, se intentó validar su legitimidad consultando otras fuentes oficiales. Finalmente, gracias a la colaboración de la autora Zeina Migeed, se confirmó que dicho enfoque es empleado en la práctica incluso en trabajos de carácter riguroso.

Este enfoque permitió elaborar una lista detallada de errores observados, junto con las condiciones específicas en las que se producen, para cada una de las operaciones. A continuación, se presenta un ejemplo correspondiente a la operación `reshape`, en el que se describen los errores que pueden surgir en momento de ejecución con tensores de entrada o parámetros no válidos.

La operación `reshape` devuelve un tensor con los mismos datos y número de elementos que la entrada, pero con la forma especificada, y los errores que pueden ocurrir son los siguientes:

- Usar más de una dimensión con el valor -1, cuando solo una puede ser inferida automáticamente por PyTorch.

```
t = torch.zeros(4)
torch.reshape(t, (-1,-1))
>>> RuntimeError: only one dimension can be inferred
```

- Especificar una dimensión con un valor negativo distinto de -1.

```
t = torch.zeros(4)
torch.reshape(t, (-2,2))
>>> RuntimeError: invalid shape dimension -2
```

- Indicar una forma cuyo número total de elementos no coincide con el del tensor original.

```
t = torch.zeros(4, 2)
torch.reshape(t, (4, 3))
>>> RuntimeError: shape '[4, 3]' is invalid for input of size 8
```

Luego de identificar y analizar los errores asociados a las 15 operaciones seleccionadas, el siguiente paso fue abstraer las condiciones que los provocan. Estas condiciones, que reflejan cuándo una operación puede fallar en tiempo de ejecución, se transformaron en restricciones que debían cumplirse para evitar dichos errores. Inicialmente, se consideró registrar estas restricciones en lenguaje natural, como en los ejemplos anteriores. Sin embargo, esta opción no permitía una traducción directa a un formato formal compatible con herramientas como los solucionadores SMT, que eran parte del objetivo posterior. Por ello, se decidió representar las restricciones mediante lógica de primer orden, lo que permite una formalización precisa.

Con el objetivo de capturar con precisión las restricciones observadas en cada operación a partir de los errores recolectados, se procedió a formular predicados que describieran tanto las dimensiones de los tensores de entrada como las de salida, así como las relaciones entre ambas. Esta representación debía incorporar el uso de cuantificadores, operadores como productos y elementos de lógica proposicional básica, permitiendo así una formalización rigurosa. A continuación, se presenta un ejemplo utilizando la operación `reshape`, basado en los errores registrados para dicha operación:

La operación `reshape` toma como entrada un tensor de dimensión n, con forma $[x_1, x_2, \ldots, x_n]$, y una tupla $(y_1, y_2, \ldots, y_m)$. Produce como salida un tensor de dimensión m, con forma $[y_1, y_2, \ldots, y_m]$.

Las restricciones asociadas a esta operación, las cuales fueron formuladas a partir del análisis de los errores recolectados, se expresan mediante lógica de primer orden de la siguiente manera:

1. Cada elemento de la tupla de destino debe ser positivo o igual a -1:

$$\forall i \in [1..m] : \quad y_i > 0 \ \vee \ y_i = -1$$

2. El número total de elementos debe conservarse para el tensor de salida, es decir, el producto de las dimensiones de entrada debe ser igual al de salida:

$$\prod_{i=1}^{n} x_i = \prod_{i=1}^{m} y_i$$

La idea de formalizar las restricciones obtenidas a partir de los errores, utilizando lógica de primer orden, fue fundamental para el desarrollo del resto del trabajo, ya que sentó las bases de cómo queríamos representar y capturar el comportamiento de las operaciones.

Posteriormente, se decidió trasladar esta lógica a una gramática formal que permitiera describir las restricciones sobre los tensores en términos de tipos. Esta decisión se fundamenta en que, en la literatura, es común abordar este tipo de problemas mediante gramáticas formales, ya que ofrecen una representación estructurada y precisa .

4.2. Gramática

Tras múltiples iteraciones y refinamientos, se definió una gramática expresiva que cumple con los objetivos planteados al inicio. La gramática final refleja rigurosamente la estructura de los tipos tensoriales y sus restricciones, facilitando una representación clara del comportamiento de las operaciones.

4.3. Consideraciones previas

A continuación, se presentan algunas consideraciones previas tomadas en cuenta al inicio del diseño de esta gramática, destacando los principios conceptuales que guiaron su desarrollo y las decisiones clave que contribuyeron a definir sus límites y capacidades expresivas.

- **Elección de una gramática formal.**
  Se optó por desarrollar una gramática formal como base del sistema, dado que este enfoque, ampliamente respaldado en la literatura, permite modelar de manera precisa las restricciones que se desean imponer dentro del sistema de tipos.

- **Adopción y extensión de gramática previa.**

  El diseño inicial se basó en la propuesta presentada en *Generalizing Shape Analysis with Gradual Typing* [12]. No obstante, dicha propuesta presentaba limitaciones importantes para el análisis requerido: no consideraba las restricciones como parte integral de la gramática y carecía de soporte para elementos clave como el manejo de listas de tensores y operaciones adicionales, incluyendo operaciones unarias (como `ceil` o valor absoluto) y operadores lógicos más complejos dentro de las restricciones. Por estas razones, la gramática fue extendida significativamente para aumentar su expresividad y adaptarse a estas necesidades, permitiendo así un análisis más detallado y preciso de las estructuras y restricciones.

- **Integración de restricciones y operaciones específicas.**

  Siguiendo el enfoque del trabajo PyTea, las restricciones se integran de manera explícita dentro de la gramática. No obstante, en esta propuesta se amplía este concepto incorporando las operaciones necesarias para representar completamente las expresiones propias de PyTorch, incluyendo los tensores de entrada y salida, junto con sus respectivas restricciones. Estas restricciones se formulan mediante lógica de primer orden, lo que permite su traducción directa a herramientas de verificación formal como el SMT solver Z3.

- **Enfoque en operaciones con firmas complejas.**

  Operaciones en PyTorch como `cat`, `sum`, `split`, `unsqueeze`, `max`, `squeeze`, `expand` y `any`, presentan múltiples variantes: una forma simplificada que puede ejecutarse sin parámetros (por ejemplo, sin especificar `dim`) y una forma más compleja que requiere argumentos adicionales. En esta gramática se decidió modelar únicamente las variantes complejas, ya que son las que pueden inducir errores en tiempo de ejecución. Las versiones triviales, al no generar errores y ser fácilmente inferibles, fueron deliberadamente omitidas.

4.4. Sintaxis

La sintaxis de la Figura 4.1 define la estructura del lenguaje, permitiendo representar, operaciones, restricciones sobre tensores y expresiones numéricas. Esta gramática incluye los siguientes elementos clave:

- **Declaraciones** (`decl`), donde las variables se asocian explícitamente a tipos concretos (T), incluyendo tipos básicos como booleanos ($\mathbb{B}$), enteros ($\mathbb{Z}$), tensores ($TT([d_0, ..., d_n])$) y listas.
- **Tipos** (T), donde un tensor se representa mediante la notación $TT([d_0, ..., d_n])$, en la que $d_0, \ldots, d_n$ indican sus dimensiones, que pueden ser variables o valores enteros ($\mathbb{Z}$). Asimismo, al igual que en PyTorch, la gramática asume índices que comienzan en 0, garantizando una representación coherente y consistente de los tensores.
- **Expresiones** (e), tales como las siguiente:

  - Operaciones binarias y unarias (como suma, resta, valor absoluto, `floor` y `ceil`).
  - Manipulación estructural de tensores (por ejemplo, `reshape`, `transpose`, `cat`, `permute`).
  - Generación de tensores (por ejemplo, `zeros`, `ones`, `range`, `arange`).

  Cabe destacar que algunas operaciones de PyTorch pueden aparecer múltiples veces en la gramática con distintas variantes. Esto se debe a que ciertas funciones presentan comportamientos diferentes según los tipos y cantidad de argumentos con que se invocan, por lo que es necesario definir varias firmas para representar con precisión estas diferencias.

- **Restricciones** (C), diseñadas para establecer relaciones entre expresiones mediante operadores relacionales ($=$, $\neq$, $\leq$, $\geq$, etc.), admitiendo la composición lógica mediante conjunciones.
- **Ambiente** (Γ), utilizado para mantener un ambiente de tipado estático, registrando las asociaciones entre variables y sus respectivos tipos.
- **Parámetros explícitos**: Algunas operaciones requieren la provisión explícita de valores enteros (por ejemplo, dimensiones concretas), reflejando las restricciones propias de operaciones que dependen del tamaño o forma del tensor.

$$
\begin{aligned}
& n, m \in \mathbb{Z} \\
\text{(Declaration)} \quad & \mathit{decl} ::= x : T \\
\text{(Type)} \quad & T ::= \mathbb{B} \mid \mathbb{Z} \mid TT([d_0, ..., d_n]) \mid [T] \\
\text{(Dimension)} \quad & d ::= n \mid x \\
\text{(Constraint)} \quad & C ::= e\ op_c\ e \mid C \wedge C \mid C \vee C \mid \top \\
\text{(Operation Constraint)} \quad & op_c ::= \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \\
\text{(Operation Expressions)} \quad & op_e ::= + \mid - \mid * \mid / \\
\text{(Operation Unary)} \quad & op_u ::= |\cdot| \mid \lfloor \cdot \rfloor \mid \lceil \cdot \rceil \\
\text{(Expression)} \quad & e ::= x \\
& \quad \mid\ n \\
& \quad \mid\ [e_1, ..., e_n] \\
& \quad \mid\ e\ op_e\ e \\
& \quad \mid\ op_u\ e \\
& \quad \mid\ \text{range}(e_1, e_2) \\
& \quad \mid\ \text{range}(e_1, e_2, e_3) \\
& \quad \mid\ \text{view}(e_1, e_2) \\
& \quad \mid\ \text{reshape}(e_1, e_2) \\
& \quad \mid\ \text{transpose}(e, m_1, m_2) \\
& \quad \mid\ \text{cat}(e_1, m) \\
& \quad \mid\ \text{sum}(e_1, m) \\
& \quad \mid\ \text{split}(e, m) \\
& \quad \mid\ \text{unsqueeze}(e_1, m) \\
& \quad \mid\ \text{zeros}(e) \\
& \quad \mid\ \text{arange}(e_1, e_2) \\
& \quad \mid\ \text{arange}(e_1, e_2, e_3) \\
& \quad \mid\ \text{ones}(e) \\
& \quad \mid\ \text{max}(e_1, m) \\
& \quad \mid\ \text{squeeze}(e_1, e_2) \\
& \quad \mid\ \text{expand}(e_1, e_2) \\
& \quad \mid\ \text{any}(e_1, m) \\
\text{(Environment)} \quad & \Gamma ::= \emptyset \mid \Gamma, x : TT([d_0, ..., d_n]) \mid \Gamma, x : \mathbb{Z} = d
\end{aligned}
$$

Figura 4.1: Sintaxis de la gramática formal.

4.5. Sistema de tipos

El sistema de tipos definido en esta propuesta tiene como objetivo garantizar la coherencia estructural y semántica de las expresiones. Para ello, se introduce un conjunto de reglas

formales que permiten verificar el correcto uso de tensores, expresiones numéricas y restricciones, asegurando que las operaciones sean aplicadas únicamente cuando sus condiciones de tipado lo permiten. Se comenzará presentando las reglas básicas del sistema, para luego introducir las reglas específicas de tipado orientadas a las operaciones PyTorch.

4.5.1. Reglas Básicas del Sistema de Tipos

Comenzando con las reglas del sistema de tipos que permiten derivar expresiones tenemos las siguientes:

- EX-N: Esta regla establece que cualquier número n puede tiparse como un entero $\mathbb{Z}$ bajo cualquier ambiente Γ, sin generar ninguna restricción adicional (representada por $\top$).
- EX-XZ: Si en el ambiente Γ existe una variable x asociada a un valor entero d, podemos derivar que x tiene tipo $\mathbb{Z}$ bajo el mismo ambiente, sin introducir nuevas restricciones.
- EX-XT: Si en el ambiente Γ una variable x está asociada a un tipo $TT([d_0, ..., d_n])$, podemos derivar que x tiene ese mismo tipo $TT([d_0, ..., d_n])$, sin generar restricciones adicionales.
- EX-U-OP: Si desde el ambiente Γ podemos derivar que una expresión e es de tipo $\mathbb{Z}$ con una cierta condición C, entonces podemos concluir que una operación unaria aplicada sobre e también es de tipo $\mathbb{Z}$, con esa misma condición C.
- EX-B-OP: Si desde el ambiente Γ podemos derivar que dos expresiones e_1 y e_2 son de tipo $\mathbb{Z}$, con sus respectivas condiciones C_1 y C_2, entonces una operación binaria op_e entre e_1 y e_2 será de tipo $\mathbb{Z}$, y su condición será la conjunción $C_1 \wedge C_2$.

$\boxed{\Gamma \vdash_1 e : T; C}$

$$\text{(EX-N)}\frac{}{\Gamma \vdash_1 n : \mathbb{Z}; \top} \qquad \text{(EX-XZ)}\frac{(x : \mathbb{Z} = d) \in \Gamma}{\Gamma \vdash_1 x : \mathbb{Z}; \top}$$

$$\text{(EX-XT)}\frac{x : TT([d_0, ..., d_n]) \in \Gamma}{\Gamma \vdash_1 x : TT([d_0, ..., d_n]); \top} \qquad \text{(EX-U-OP)}\frac{\Gamma \vdash_1 e : \mathbb{Z}; C}{\Gamma \vdash_1 op_u\ e : \mathbb{Z}; C}$$

$$\text{(EX-B-OP)}\frac{\Gamma \vdash_1 e_1 : \mathbb{Z}; C_1 \quad \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2}{\Gamma \vdash_1 e_1\ op_e\ e_2 : \mathbb{Z}; C_1 \wedge C_2}$$

Luego tenemos la regla V-TT que establece que una expresión de tipo $TT([d_0, ..., d_n])$ puede tiparse correctamente bajo un ambiente Γ si cada una de sus dimensiones d_i puede ser derivada como un entero ($\mathbb{Z}$) con su respectiva condición C_i. Adicionalmente, al derivarse se tiene que todas las dimensiones sean estrictamente mayores que cero. En resumen, esta regla garantiza que todos los tamaños declarados en el tipo estructurado sean enteros válidos y positivos.

$\boxed{\Gamma \vdash_4 TT([d_0, ..., d_n]) : C}$

$$\text{(V-TT)}\frac{\forall i.\Gamma \vdash_1 d_i : \mathbb{Z}; C_i}{\Gamma \vdash_4 TT([d_0, ..., d_n]) : \bigwedge_{i=1}^{n}(d_i > 0)}$$

A continuación, se definen las reglas relacionadas con la derivación de ambientes:

- E-ENV: Esta regla establece que el ambiente vacío siempre es válido y satisface la condición trivial $\top$, es decir, sin generar restricciones adicionales.
- E-INT:Si desde el ambiente Γ podemos derivar una restricción C, entonces al extenderlo con una variable x de tipo $\mathbb{Z}$ asociada a un valor constante d, la nueva condición será $C \wedge (x = d)$, indicando que x debe tener ese valor específico.
- E-TT: Si desde el ambiente Γ podemos derivar una restricción C_1, y además podemos derivar con $\vdash_4$ un $TT([d_0, ..., d_n])$ valido (enteros validos y positivos de dimensiones) con una restricción C_2, entonces podemos extender el ambiente con una variable x asociada a dicho $TT([d_0, ..., d_n])$, con una nueva restricción que sera será $C_1 \wedge C_2$.

$\boxed{\vdash_0 \Gamma : C}$

$$\text{(E-ENV)}\frac{}{\vdash_0 \emptyset : \top} \qquad \text{(E-INT)}\frac{\vdash_0 \Gamma : C}{\vdash_0 \Gamma, (x : \mathbb{Z} = d) : C \wedge (x = d)}$$

$$\text{(E-TT)}\frac{\vdash_0 \Gamma : C_1 \quad \Gamma \vdash_4 TT([d_0, ..., d_n]) : C_2}{\vdash_0 \Gamma, x : TT([d_0, ..., d_n]) : C_1 \wedge C_2}$$

A continuación, las reglas del sistema de tipos relacionadas con la derivación de restricciones:

- C-TOP: Esta regla indica que la restricción $\top$ siempre se considera satisfecha bajo cualquier ambiente Γ, sin necesidad de realizar derivaciones adicionales.
- C-OP: Si podemos derivar que dos expresiones e_1 y e_2 de tipo $\mathbb{Z}$ bajo un ambiente Γ, con restricciones C_1 y C_2 respectivamente, podemos derivar bajo ese ambiente la operación de restricciones op_c de esas dos expresiones e_1 $\texttt{op}_c$ e_2.
- C-AND: Si desde un ambiente Γ podemos derivar las restricciones C_1 y C_2 por separado, entonces también podemos derivar las conjuciones $C_1 \wedge C_2$ bajo el mismo ambiente.
- C-OR: Si desde un ambiente Γ podemos derivar C_1 y C_2, entonces también podemos derivar la disyunción $C_1 \vee C_2$ bajo el mismo ambiente.

$\boxed{\Gamma \vdash_2 C}$

$$\text{(C-TOP)}\frac{}{\Gamma \vdash_2 \top} \qquad \text{(C-OP)}\frac{\Gamma \vdash_1 e_1 : \mathbb{Z}; C_1 \quad \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2}{\Gamma \vdash_2 e_1\ op_c\ e_2}$$

$$\text{(C-AND)}\frac{\Gamma \vdash_2 C_1 \quad \Gamma \vdash_2 C_2}{\Gamma \vdash_2 C_1 \wedge C_2} \qquad \text{(C-OR)}\frac{\Gamma \vdash_2 C_1 \quad \Gamma \vdash_2 C_2}{\Gamma \vdash_2 C_1 \vee C_2}$$

Finalmente, la regla principal del sistema de tipos, TOP-LVL, indica lo siguiente: si podemos derivar con $\vdash_0$ que un ambiente Γ da una restricción base C_1, si con $\vdash_1$ una expresión e tiene tipo T bajo ese ambiente con su respectiva restricción C_2, y además tenemos que con $\vdash_2$ Γ puede derivar otra restricción C que se necesita, y la conjunción $C_1 \wedge C_2$ es satisfactible, entonces podemos concluir que e tiene tipo T bajo el ambiente Γ.

Esta regla actúa como el punto de integración entre las distintas partes del sistema: combina las reglas de derivación del entorno, las de expresiones y las de restricciones. En esencia, garantiza que una expresión está bien tipada si el ambiente de ejecución cumple con las restricciones necesarias y si estas restricciones, junto con las generadas durante el tipado de la expresión, pueden satisfacerse simultáneamente.

$\boxed{\Gamma \vdash_3 e : T}$

$$\text{(TOP-LVL)}\frac{\vdash_0 \Gamma : C_1 \quad \Gamma \vdash_1 e : T; C_2 \quad \Gamma \vdash_2 C \quad C_1 \wedge\ C_2 \text{ sat}}{\Gamma \vdash_3 e : T}$$

Esta sección presenta las reglas básicas que sirven como punto de partida para derivar las demás reglas del sistema. En la siguiente sección, continuaremos con el sistema de tipos, pero ahora enfocado en las operaciones específicas de PyTorch.

4.5.2. Extensión del sistema de tipos para PyTorch

A continuación se presenta la extensión del sistema de tipos con operaciones específicas de PyTorch. Para cada operación se incluye su firma (o múltiples firmas, si aplica), un enlace a la documentación oficial de PyTorch 2.7, un breve resumen, una explicación de las restricciones que se generan, y una descripción de la forma del tensor resultante.

1) `range`(e_1`,` e_2`)` **y `range`(e_1`,` e_2`,` e_3`)` (PyTorch Doc)**
Crea un tensor con una secuencia de enteros dentro de un rango especificado. En el caso de OP-RANGE-2, la restricción C_3 asegura que el valor final del rango sea mayor que el inicial, ya que por defecto el parámetro `step` es igual a 1. Además, C_4 especifica que el resultado es el tensor unidimensional con el valor de x.

Para OP-RANGE-3, la restricción C_4 garantiza que el tercer parámetro `step` no sea igual a cero. La restricción C_5 impone que el valor de `step` sea consistente con el orden del rango (por ejemplo, positivo si el rango es ascendente). Finalmente, C_6 indica que el tensor de salida

es unidimensional con valor x.

$$\text{(OP-RANGE-2)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : \mathbb{Z}; C_1 \quad \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2 \\ x \text{ free} \\ C_3 = e_1 \leq e_2 \\ C_4 = x = |e_2 - e_1| + 1 \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4\end{array}}{\Gamma \vdash_3 \texttt{range}(e_1, e_2) : TT([x]); C}$$

$$\text{(OP-RANGE-3)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : \mathbb{Z}; C_1 \quad \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2 \quad \Gamma \vdash_1 e_3 : \mathbb{Z}; C_3 \\ x \text{ free} \\ C_4 = e_3 \neq 0 \\ C_5 = |e_2 - e_1| * e_3 > 0 \\ C_6 = x = \lfloor \frac{e_2 - e_1}{e_3} \rfloor + 1 \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4 \wedge\ C_5 \wedge\ C_6\end{array}}{\Gamma \vdash_3 \texttt{range}(e_1, e_2, e_3) : TT([x]); C}$$

2) `view(`e_1`,` e_2`)` **(PyTorch Doc)**
Transforma la forma del tensor sin alterar sus datos. En la regla OP-VIEW, e_1 es el tensor de entrada con tipo $TT([d_0, ..., d_n])$ y restricción C_1, y e_2 representa la nueva forma deseada. Aunque en términos prácticos suele pasarse como una lista de enteros separados por comas, conceptualmente puede considerarse como un tensor de tipo $TT([d'_1, \ldots, d'_n])$, con su respectiva restricción C_2. La restricción C_3 asegura que la cantidad total de elementos en ambas formas sea igual, es decir, que el producto de las dimensiones originales sea igual al producto de las dimensiones de la nueva forma. La restricción C_4 garantiza que todas las dimensiones en e_2 sean mayor que 0, mientras que C_5 impone que todas las dimensiones de e_1 sean mayor a 0 o -1. El tipo de salida es $TT([d'_0, \ldots, d'_l])$, correspondiente a la nueva forma establecida por e_2.

$$\text{(OP-VIEW)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 e_2 : TT([d'_0, \ldots, d'_m]); C_2 \\ C_3 = \prod_{i=1}^{n} d_i = \prod_{i=1}^{m} d'_i \\ C_4 = \bigwedge_{i=1}^{m} (d'_i > 0) \\ C_5 = \bigwedge_{i=1}^{n} (d_i > 0) \vee (d_i = -1) \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4 \wedge\ C_5 \wedge\ C_6\end{array}}{\Gamma \vdash_3 \texttt{view}(e_1, e_2) : TT([d'_0, ..., d'_n]); C}$$

3) `reshape(`e_1`,` e_2`)` **(PyTorch Doc)**
Similar a view, modifica la forma de un tensor sin alterar la cantidad total de elementos que contiene. En la regla OP-RESHAPE, e_1 es un tensor de entrada con forma $[d_0, \ldots, d_n]$, y e_2 es un tensor que representa la nueva forma deseada $[d'_0, \ldots, d'_m]$ cada uno con sus respectivas dimensiones C_1 y C_2. La restricción C_3 garantiza que el número total de elementos se conserve entre ambas formas. Luego, C_4 garantiza que todas las dimensiones de salida de e_2 sean positivas y C_5 garantiza que las dimesiones del tensor de entrada sean mayor a 0 o -1.

Finalmente el tensor de salida tiene la forma del parametro e_2.

$$\text{(OP-RESHAPE)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \quad \Gamma \vdash_1 e_2 : TT([d'_0, ..., d'_n]); C_2 \\ C_3 = \prod_{i=1}^{n} d_i = \prod_{i=1}^{m} d'_i \\ C_4 = \bigwedge_{i=1}^{m} (d'_i > 0) \\ C_5 = \bigwedge_{i=1}^{n} (d_i > 0) \vee (d_i = -1) \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4 \wedge\ C_5\end{array}}{\Gamma \vdash_3 \texttt{reshape}(e_1, e_2) : TT([d'_0, ..., d'_n]); C}$$

4) `transpose(`e_1`,` m_1`,` m_2`)` **(PyTorch Doc)**
Intercambia dos dimensiones específicas de un tensor. En la regla OP-TRANSPOSE, e_1 es un tensor de entrada con forma $[d_0, \ldots, d_n]$, y m_1, m_2 son los índices enteros de las dimensiones a intercambiar. Las restricciones C_2 y C_3 corresponden a asegurar que estén dentro del rango del tensor de tipo e_1. C_4 se utiliza para establecer el cambio de dimension. Finalmente el tensor de salida es el mismo tensor con las dimensiones m_1 y m_2 intercambiadas.

$$\text{(OP-TRANSPOSE)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m_1 : \mathbb{Z}; \top \quad \Gamma \vdash_1 m_2 : \mathbb{Z}; \top \\ x \text{ free} \\ x : TT([d'_0, \ldots, d'_n]) \\ C_2 = -n \leq m_1 \wedge m_1 < n - 1 \\ C_3 = -n \leq m_2 \wedge m_2 < n - 1 \\ C_4 = (d'_{m_1} = d_{m_2}) \wedge (d'_{m_2} = d_{m_1}) \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4\end{array}}{\Gamma \vdash_3 \texttt{tranpose}(e_1, m_1, m_2) : TT([d'_0, ..., d'_n]); C}$$

5) `cat(`e_1`,` m`)` **(PyTorch Doc)**
Concatena una lista de tensores a lo largo de una dimensión específica. En la regla OP-CAT, e_1 es una lista de tensores del mismo tipo, y m es un entero que indica la dimensión en la que se desea realizar la concatenación. Las restricciones C_1 y C_2 se encargan de verificar que todos los tensores en la lista tienen tipo $TT([d_0, ..., d_n])$, y que sus dimensiones coinciden en todas las posiciones excepto en la m-ésima. En esta notación, $d_{i,j}$ representa la longitud de la dimensión i del tensor j-ésimo en la lista, además sea l el último tensor de la lista. La restricción C_3 define que el nuevo tamaño en la dimensión m (denotado por x) es la suma de todas las longitudes en esa dimensión de los tensores originales. A partir de esta restricción, se deduce que el tipo del tensor resultante es un tensor concatenado con el valor x en la dimensión m.

$$\text{(OP-CAT)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : [TT(d_{0,1}, \ldots, d_{n,1}), \ldots, TT(d_{0,l}, \ldots, d_{n,l})]; C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ x \text{ free} \\ C_2 = \bigwedge_{j=1}^{l} \bigwedge_{\substack{i=0 \\ i \neq m}}^{n} d_{i,1} = d_{i,j} \\ C_3 = x = \sum_{j=1}^{l} d_{m,j} \\ C = C_1 \wedge\ C_2 \wedge\ C_3\end{array}}{\Gamma \vdash_3 \texttt{cat}(e_1, m) : TT([d_0, \ldots, d_{m-1}, x, d_{m+1}, \ldots, d_n]); C}$$

6) `sum(`e_1`,` m`)` **(PyTorch Doc)**
Devuelve la suma de cada fila del tensor de entrada en la dimensión especificada. En la regla OP-SUM, e_1 tiene un tipo $TT([d_0, ..., d_n])$ con su restricción asociada C_1, y m es de tipo entero con su respectiva restricción $\top$. La restricción C_2 asegura que la dimensión m esté dentro del rango válido de dimensiones del tensor de entrada. A partir de estas restricciones, se concluye que el tipo del tensor resultante es $TT([d_0, \ldots, d_{e_2-1}, d_{e_2+1}, \ldots, d_n])$, que corresponde a el mismo tensor de entrada pero con la dimensión de m eliminada.

$$\text{(OP-SUM)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ C_2 = -n \leq m \wedge m < n - 1 \\ C = C_1 \wedge\ C_2\end{array}}{\Gamma \vdash_3 sum(e_1, m) : TT([d_0, \ldots, d_{m-1}, d_{m+1}, \ldots, d_n]); C}$$

7) `split(`e_1`,` m`)` **(PyTorch Doc)**
Divide un tensor a lo largo de una dimensión en múltiples sub-tensores del mismo tamaño. La regla OP-SPLIT modela el caso simple de esta operación, en el que el tensor se divide en partes iguales. En esta regla, e_1 representa el tensor de entrada con tipo $TT([d_0, ..., d_n])$ y restricción asociada C_1. El segundo argumento m es un entero que indica el número de partes en las que se dividirá el tensor, y se tipa con restricción trivial $\top$. La restricción C_2 garantiza que el valor de m sea estrictamente mayor que cero. A partir de estas restricciones, se deduce que el tipo del resultado es una lista de tensores de tamaño m con tipo $TT([d_0, ..., d_n])$.

$$\text{(OP-SPLIT)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ C_2 = m > 0 \\ C = C_1 \wedge\ C_2\end{array}}{\Gamma \vdash_3 \texttt{split}(e_1, m) : [TT(d_{0,1}, \ldots, d_{n,1}), \ldots, TT(d_{0,m}, \ldots, d_{n,m})]]; C}$$

8) `unsqueeze(`e_1`,` m`)` **(PyTorch Doc)**
Inserta una nueva dimensión de tamaño uno en la posición especificada del tensor. En la regla OP-UNSQUEEZE, e_1 es el tensor de entrada con tipo $TT([d_0, ..., d_n])$ y restricción asociada C_1. El segundo argumento m representa la posición donde se desea insertar la nueva dimensión. La restricción C_2 garantiza que el índice m sea válido, cumpliendo que $-n - 1 \leq m < n + 1$. Finalmente el tensor de salida es un tensor con la nueva dimensión unitaria insertada en la posición m, es decir, $TT([d_0, \ldots, d_m, 1, d_{m+1}, \ldots, d_n])$.

$$\text{(OP-UNSQUEEZE)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ C_2 = -n - 1 \leq m \wedge m < n + 1 \\ C = C_1 \wedge\ C_2\end{array}}{\Gamma \vdash_3 \texttt{unsqueeze}(e_1, m) : TT([d_0, \ldots, d_{m-1}, 1, d_{m+1}, \ldots, d_n]); C}$$

9) `zeros(e)` (PyTorch Doc)
Crea un tensor lleno de ceros a partir de una expresión entera que representa la longitud deseada. En la regla OP-ZEROS-INT, e es una expresión de tipo entero con restricción C_1, y la restricción C_2 garantiza que dicho valor sea estrictamente positivo. C_3 indica que el valor de la nueva dimensión del tensor resultante es e utilizando una variable x. Finalmente, el tipo del tensor resultante es $TT([x])$, es decir, un tensor unidimensional de longitud x.

En la regla OP-ZEROS-TT, se permite crear un tensor de ceros usando directamente una forma de un tensor e que ya representa una forma válida, es decir, tiene tipo $TT([d_0, ..., d_n])$ con restricción C_1. En este caso, el tipo del tensor de salida es simplemente el mismo: $TT([d_0, ..., d_n])$.

$$\text{(OP-ZEROS-INT)}\frac{\begin{array}{c}\Gamma \vdash_1 e : \mathbb{Z}; C_1 \\ x \text{ free} \\ C_2 = 0 < e \\ C_3 = x = e \\ C = C_1 \wedge\ C_2 \wedge\ C_3\end{array}}{\Gamma \vdash_3 \texttt{zeros}(e) : TT([x]); C}$$

$$\text{(OP-ZEROS-TT)}\frac{\Gamma \vdash_1 e : TT([d_0, ..., d_n]); C}{\Gamma \vdash_3 \texttt{zeros}(e) : TT([d_0, ..., d_n]); C}$$

10) `arange`(e_1, e_2) y `arange`(e_1, e_2, e_3) (PyTorch Doc)
Crea un tensor unidimensional con una secuencia de enteros dentro de un rango determinado. Las reglas OP-ARANGE-2 y OP-ARANGE-3 son funcionalmente equivalentes a las reglas OP-RANGE-2 y OP-RANGE-3, respectivamente, con algunas diferencias sutiles. En OP-RANGE-2, la restricción C_4 incluye una suma adicional de 1 en la longitud del rango, lo cual no ocurre en C_4 de OP-ARANGE-2. Asimismo, en OP-RANGE-3, la restricción C_6 utiliza la operación unaria `ceil` para calcular la longitud, mientras que OP-ARANGE-3 emplea `floor` y omite la suma del valor 1.

$$\text{(OP-ARANGE-2)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : \mathbb{Z}; C_1 \quad \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2 \\ x \text{ free} \\ C_3 = e_1 \leq e_2 \\ C_4 = x = |e_2 - e_1| \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4\end{array}}{\Gamma \vdash_3 \texttt{arange}(e_1, e_2) : TT([x]); C}$$

$$\text{(OP-ARANGE-3)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : \mathbb{Z}; C_1 \quad \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2 \quad \Gamma \vdash_1 e_3 : \mathbb{Z}; C_3 \\ x \text{ free} \\ C_4 = e_3 \neq 0 \\ C_5 = |e_2 - e_1| * e_3 > 0 \\ C_6 = x = \lceil \frac{e_2 - e_1}{e_3} \rceil \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4 \wedge\ C_5 \wedge\ C_6\end{array}}{\Gamma \vdash_3 \texttt{arange}(e_1, e_2, e_3) : TT([x]); C}$$

11) `ones`(e) (PyTorch Doc)
Crea un tensor lleno de unos a partir de una expresión que indica su tamaño. En la regla

OP-ONES-INT, la expresión e es de tipo entero, con una restricción C_1, y la restricción C_2 asegura que el valor sea estrictamente positivo. C_3 indica que el valor de la nueva dimensión del tensor resultante es e utilizando una variable x. Finalmente, el tipo del tensor resultante es $TT([x])$, es decir, un tensor unidimensional de longitud x.

En la regla OP-ONES-TT, se permite generar un tensor de unos utilizando una expresión e que ya tiene tipo tensorial con forma conocida, tipada como $TT([d_0, ..., d_n])$ bajo la restricción C_1. El tensor de salida conserva esa misma forma.

$$\text{(OP-ONES-INT)}\frac{\begin{array}{c}\Gamma \vdash_1 e : \mathbb{Z}; C_1 \\ x \text{ free} \\ C_2 = 0 < e \\ C_3 = x = e \\ C = C_1 \wedge\ C_2 \wedge\ C_3\end{array}}{\Gamma \vdash_3 \texttt{ones}(e) : TT([x]); C}$$

$$\text{(OP-ONES-TT)}\frac{\Gamma \vdash_1 e : TT([d_0, ..., d_n]); C}{\Gamma \vdash_3 \texttt{ones}(e) : TT([d_0, ..., d_n]); C}$$

12) `max(`e_1`,` m`)` **(PyTorch Doc)**
Devuelve el máximo valor de todo los elementos del tensor en un nuevo tensor, utilizando una dimensión especifica. En la regla OP-MAX, e_1 es un tensor con tipo $TT([d_0, ..., d_n])$ bajo la restricción C_1, y m es una expresión entera que indica la dimensión sobre la cual se realiza la operación, con su restricción correspondiente $\top$. C_3 garantiza que dicha dimensión se encuentre dentro del rango válido. El resultado tiene tipo $TT([d_0, \ldots, d_{m-1}, d_{m+1}, \ldots, d_n])$ que corresponde al mismo tensor con la dimensión m eliminada.

$$\text{(OP-MAX)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ C_3 = -n \leq m \wedge m < n - 1 \\ C = C_1 \wedge\ C_2 \wedge\ C_3\end{array}}{\Gamma \vdash_3 \texttt{max}(e_1, e_2) : TT([d_0, \ldots, d_{m-1}, d_{m+1}, \ldots, d_n]); C}$$

13) `squeeze(`e_1`,` e_2`)` **(PyTorch Doc)**
Elimina dimensiones de tamaño uno dentro de un tensor en las posiciones especificadas. En la regla OP-SQUEEZE, e_1 es un tensor de entrada con tipo $TT([d_0, ..., d_n])$ y restricción C_1, mientras que e_2 es una expresión entera que indica en que dimensión se aplicara la operación. Las restricciones C_2 asegura que la expresión e_2 está en un rango valido del tensor de entrada. La restricción C_3 y C_4 aseguran que el tensor de salida no tenga elementos con dimensiones de tamaño 1 y que su tamaño sea menor o igual al del tensor de entrada. El tipo del tensor resultante corresponde al mismo tipo de entrada, pero potencialmente con menos

dimensiones, dependiendo de si las dimensiones evaluadas son de tamaño uno.

$$\text{(OP-SQUEEZE-INT)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 e_2 : \mathbb{Z}; C_2 \\ x \text{ free} \\ x : TT([d'_0, \ldots, d'_n]) \\ C_2 = -n \leq e_2 \wedge e_2 < n - 1 \\ C_3 = \bigwedge_{i=0}^{n'} (d'_i \neq 1) \\ C_4 = n' \leq n \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4\end{array}}{\Gamma \vdash_3 \texttt{squeeze}(e_1, e_2) : TT([d'_0, \ldots, d'_n]); C}$$

14) `expand(`e_1`,` e_2`)` **(PyTorch Doc)**
Expande un tensor a una nueva forma compatible, duplicando elementos si es necesario. En la regla OP-EXPAND, e_1 es el tensor original con tipo $TT([d_0, ..., d_n])$ y restricción C_1, mientras que e_2 representa la forma objetivo, también de tipo tensorial con restricción C_2. La restricción C_3 asegura que la forma objetivo tenga una cantidad de dimensiones mayor o igual a la del tensor original, lo cual es necesario para que la expansión sea válida. El tipo de salida es $TT([d'_0, \ldots, d'_n])$, es decir, el tensor expandido a la nueva forma solicitada.

$$\text{(OP-EXPAND)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \quad \Gamma \vdash_1 e_2 : TT([d'_0, ..., d'_n]); C_2 \\ C_3 = n \leq n' \\ C = C_1 \wedge\ C_2 \wedge\ C_3\end{array}}{\Gamma \vdash_3 \texttt{expand}(e_1, e_2) : TT([d'_0, ..., d'_n]); C}$$

15) `any(`e_1`,` m`)` **(PyTorch Doc)**
Evalúa si al menos un elemento en el tensor es evalua a `True` largo de una dimensión específica. En la regla OP-ANY, e_1 es el tensor de entrada con tipo $TT([d_0, ..., d_n])$ y restricción C_1, y m es una expresión entera que indica la dimensión a reducir. C_3 garantiza que la dimensión indicada esté dentro del rango válido del tensor. El tipo de salida es $TT([d_0, \ldots, d_{m-1}, d_{m+1}, \ldots, d_n])$.

$$\text{(OP-ANY)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ C_3 = -n \leq m \wedge m < n - 1 \\ C = C_1 \wedge\ C_2 \wedge\ C_3\end{array}}{\Gamma \vdash_3 \texttt{any}(e_1, e_2) : TT([d_0, \ldots, d_{m-1}, d_{m+1}, \ldots, d_n]); C}$$

4.6. Limitaciones de la gramática formal

La gramática presentada en este trabajo es lo suficientemente expresiva para capturar las condiciones necesarias que permiten que las funciones elegidas de PyTorch se ejecuten sin errores en tiempo de ejecución. Gracias a su diseño, es posible representar para cada operación los tensores de entrada, de salida, sus parámetros, y sus respectivas restricciones que deben cumplir para poder ejecutarse. Sin embargo, a pesar de estas capacidades, existen ciertas limitaciones que vale la pena destacar, ya que afectan el alcance y la precisión del análisis en casos particulares.

- **Dependencia de valores numéricos concretos.**
  Operaciones como `any`, `expand` y `unsqueeze` requieren que ciertos parámetros, particularmente aquellos que hacen referencia a dimensiones específicas del tensor, sean conocidos de forma numérica explícita. Cuando estos valores se expresan mediante una expresión, la gramática no es capaz de inferir información concreta sobre su resultado antes de la ejecución. Esto limita su capacidad para formular restricciones sobre dichas expresiones.

- **Restricciones en la representación de firmas complejas.**
  La operación `split` cuenta con múltiples variantes en PyTorch, incluyendo una versión avanzada que permite dividir tensores en segmentos de tamaños arbitrarios especificados por una lista. No obstante, en este trabajo se decidió modelar únicamente la variante más simple de `split`, aquella que divide el tensor en partes iguales. Esta decisión se debió a que la notación requerida para expresar la versión compleja excedía las capacidades actuales de la gramática.

- **Empleo de variables libres como recurso auxiliar.**
  En determinadas operaciones fue necesario incorporar variables libres dentro de las restricciones para completar adecuadamente la definición de la operación. Esta decisión se tomó porque, en algunos casos, la gramática no permite representar de forma directa ciertas relaciones entre dimensiones o parámetros, lo que hace necesario el uso de estas variables auxiliares para expresar correctamente dichas dependencias.

Capítulo 5

Evaluación

En esta sección se presentan dos ejemplos de aplicación que ilustran el uso práctico de la gramática desarrollada, utilizando como caso de estudio las operaciones `unsqueeze` y `reshape`. Primero se comenzó con una operación sencilla como `unsqueeze`, para luego avanzar hacia un caso más complejo como `reshape`. Ambos ejemplos fueron implementados en Python, haciendo uso del módulo oficial de Z3. El código fuente correspondiente se encuentra disponible en el repositorio de GitHub: z3-memoria.

El objetivo principal de estos ejemplos es demostrar cómo las restricciones y estructuras definidas en la gramática pueden trasladarse de manera directa a una herramienta práctica que permita realizar comprobaciones antes de la ejecución de programas.

De este modo, se busca validar que la gramática no solo sea expresiva en términos formales, sino también útil como base para la construcción de sistemas de verificación estática. Los ejemplos ilustran cómo se invocan las operaciones `unsqueeze` y `reshape` con sus respectivos parámetros de entrada, a partir de los cuales se generan las restricciones que posteriormente se evalúan mediante el SMT Solver Z3 para determinar su satisfacibilidad.

Cabe señalar que este desarrollo no constituye una herramienta de análisis completa, sino una prueba de concepto que permite validar empíricamente la expresividad y utilidad de la gramática. Las verificaciones realizadas se basan directamente en las reglas formales definidas para `reshape`, evaluando si las restricciones se satisfacen correctamente bajo diferentes configuraciones de tensores de entrada y salida.

5.1. Ejemplo Unsqueeze

Para iniciar, se realizó una evaluación sencilla de la operación `unsqueeze`. Este ejemplo simula la invocación de dicha operación utilizando un tensor de entrada y un parámetro

entero, siguiendo la definición establecida en la gramática presentada:

$$\text{(OP-UNSQUEEZE)}\frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \\ \Gamma \vdash_1 m : \mathbb{Z}; \top \\ C_2 = -n - 1 \leq m \wedge m < n + 1 \\ C = C_1 \wedge\ C_2\end{array}}{\Gamma \vdash_3 \texttt{unsqueeze}(e_1, m) : TT([d_0, \ldots, d_{m-1}, 1, d_{m+1}, \ldots, d_n]); C}$$

Para verificar la satisfacibilidad de estas operaciones se implementó una solución utilizando un solver y variables simbólicas. El procedimiento consistió en:

1. Crear un solver.
2. Definir las variables simbólicas que representarán las dimensiones a verificar.
3. Introducir las restricciones correspondientes.

En este contexto, $e1$ corresponde a una lista de enteros y m al parámetro de la operación `unsqueeze`.

```
def is_unsqueeze_valid(e1, m: int) -> bool:
    # Crear solver Z3
    s = Solver()

    # Variables simbolicas para las dimensiones
    n = len(e1)
    e1_dims = [Int(f"d_{i}") for i in range(n)]
```

Luego se añaden las restricciones necesarias. En este caso, asumiremos la restricción $C1$ por razones prácticas:

```
    # Restriccion C2
    constraint_c2_leq = -n <= m
    constraint_c2_gt: bool = m < (n + 1)
    constraint_c2 = And(constraint_c2_leq, constraint_c2_gt)
    s.add(constraint_c2)
```

Finalmente, se verifica si el resulta satisfacible en función de las restricciones y los parámetros de entrada

```
    return s.check() == sat
```

Una vez implementada la función encargada de verificar la satisfacibilidad de las operaciones, el siguiente paso consistió en evaluarla con distintos parámetros. El primer caso corresponde a una operación válida, la cual el solver resuelve correctamente:

```
Ejemplo 1: unsqueeze([4, 2, 3], 3)
C1: True
C2: -3 <= 2: True
C2: 2 < 4: True
VALID operation.
```

El segundo caso muestra un error en la restricción $C2$, ya que el parámetro m es menor que el valor mínimo permitido (dependiente de la longitud de $e1$):

```
Ejemplo 2: unsqueeze([1, 2, 3], -200)
C1: True
C2: -3 <= -200: False
C2: -200 < 4: True
INVALID operation.
```

Finalmente, el tercer caso presenta un error en la restricción $C2$, pero esta vez debido a que m excede el límite superior. A continuación, se muestra el código y su salida:

```
Ejemplo 3: unsqueeze([1, 2, 3], 10)
C1: True
C2: -3 <= 10: True
C2: 10 < 4: False
INVALID operation.
```

5.2. Ejemplo Reshape

Después del ejemplo sencillo con `unsqueeze`, se abordó un caso más complejo que involucra dos tensores como parámetros: la operación `reshape`. La heurística utilizada fue similar a la de `unsqueeze`: partir de la definición formal en la gramática, construir una función que incorporara las restricciones correspondientes y, posteriormente, evaluar los casos problemáticos. Utilizando la definición de la gramática de `reshape`:

$$\text{(OP-RESHAPE)}\ \frac{\begin{array}{c}\Gamma \vdash_1 e_1 : TT([d_0, ..., d_n]); C_1 \quad \Gamma \vdash_1 e_2 : TT([d'_0, ..., d'_n]); C_2 \\ C_3 = \prod_{i=1}^{n} d_i = \prod_{i=1}^{m} d'_i \\ C_4 = \bigwedge_{i=1}^{m} (d'_i > 0) \\ C_5 = \bigwedge_{i=1}^{n} (d_i > 0) \vee (d_i = -1) \\ C = C_1 \wedge\ C_2 \wedge\ C_3 \wedge\ C_4 \wedge\ C_5\end{array}}{\Gamma \vdash_3 \texttt{reshape}(e_1, e_2) : TT([d'_0, ..., d'_n]); C}$$

Al igual que en el caso anterior, se comenzó creando un solver y definiendo variables simbólicas para representar cada una de las dimensiones de los tensores $e1$ y $e2$. Al igual que el ejemplo anterior, por razones prácticas se asumen satisfechas las restricciones $C1$ y $C2$.

```python
def is_reshape_valid(e1, e2) -> bool:
    # Crear solver Z3
    s = Solver()

    # Variables simbolicas para las dimensiones
    e1_dims = [Int(f"d_{i}") for i in range(len(e1))]
    e2_dims = [Int(f"d'_{i}") for i in range(len(e2))]
```

Posteriormente, se añadieron las restricciones necesarias. En primer lugar, $C3$, que establece que el producto de las dimensiones del tensor de entrada debe ser igual al producto de las dimensiones del tensor de salida:

```
    # Restricciones C3: Producto de dimensiones debe ser igual
    product_e1 = 1
    for i, dim in enumerate(e1):
        product_e1 *= dim

    product_e2 = 1
    for i, dim in enumerate(e2):
            product_e2 *= dim
    # Verifica que el producto de las dimensiones del tensor de entrada sea igual al
    producto de las dimensiones objetivo
    C_3 = product_e1 == product_e2
    s.add(C3)
```

A continuación, se incorporó la restricción $C4$, que asegura que todas las dimensiones de salida sean estrictamente positivas:

```
    # Restricciones C4: Las dimensiones de salida tienen que ser todas positivas
    constraints_c4 = []
    for i, dim in enumerate(e2):
        constraint = e2_dims[i] > 0
        constraints_c4.append(constraint)
        s.add(constraint)
```

Finalmente, se definió la restricción $C5$, la cual indica que las dimensiones de entrada deben ser estrictamente positivas o, en su defecto, tomar el valor -1:

```
    # Restriccion C5: Las dimensiones de entradas pueden ser mayor a 0 o -1
    constraints_c5 = []
    for i, dim in enumerate(e1):
        # Usar Or de Z3: e1_dims[i] > 0 OR e1_dims[i] == -1
        constraint = Or(e1_dims[i] > 0, e1_dims[i] == -1)
        constraints_c5.append(constraint)
        s.add(constraint)
```

Una vez implementada la función que verifica la satisfacibilidad de la operación `reshape`, se procedió a evaluarla con distintos ejemplos. El primer caso corresponde a una operación válida:

```
Ejemplo 1: reshape([4, 2, 3], [4, 6])
C1: True
C2: True
C3: 24 == 24
C4: True ^ True
C5: Or(True, False) ^ Or(True, False) ^ Or(True, False)
VALID operation.
```

El segundo caso muestra un error, ya que el producto de las dimensiones de entrada y salida no coincide:

```
Ejemplo 2: reshape([4, 2, 3], [4, 7])
C1: True
C2: True
C3: 24 == 28
```

```
C4: True ^ True
C5: Or(True, False) ^ Or(True, False) ^ Or(True, False)
INVALID operation.
```

Finalmente, el tercer caso presenta un error debido a la presencia de valores negativos en las dimensiones de salida:

```
Ejemplo 3: reshape([6, 4], [-3, -8])
C1: True
C2: True
C3: 24 == 24
C4: False ^ False
C5: Or(True, False) ^ Or(True, False)
INVALID operation.
```

En conclusión, aunque estos ejemplos son ilustrativos y contienen ciertas simplificaciones, la implementación de las operaciones `unsqueeze` y `reshape` demuestra que la gramática puede traducirse de manera efectiva a un entorno práctico. Esto evidencia su potencial como base para el desarrollo de sistemas de verificación estática en contextos reales.

Capítulo 6

Conclusión

El trabajo desarrollado en esta memoria constituye un primer avance hacia la detección temprana de errores relacionados con la compatibilidad de dimensiones en operaciones tensoriales dentro de programas basados en PyTorch. En el contexto actual, donde el aprendizaje profundo y el manejo de tensores son fundamentales para una amplia variedad de aplicaciones prácticas, el control adecuado de las dimensiones en las operaciones tensoriales representa un desafío clave.

Como parte de este esfuerzo, se utilizó la herramienta TEOSC para analizar un total de 4.427 archivos distribuidos en 11 repositorios distintos. Este análisis permitió identificar un conjunto representativo de 15 operaciones esenciales que constituyen el núcleo de la manipulación tensorial en PyTorch. La selección de estas operaciones clave sirvió como base para iniciar el proceso de formalización desarrollado en este trabajo.

Posteriormente, se realizó una revisión exhaustiva de la literatura y de las herramientas existentes orientadas al análisis de dimensiones tensoriales. Este análisis permitió identificar tanto las limitaciones como los aportes de los enfoques previos.

A continuación, en base a la identificación de las operaciones clave y al análisis del estado actual del problema, se procedió a la obtención de las restricciones dimensionales asociadas a cada una de estas operaciones. Para ello, se adoptó un enfoque más empírico, basado en el análisis práctico del comportamiento de las operaciones. Las restricciones obtenidas fueron formalizadas mediante expresiones en lógica booleana de primer orden. Esta formalización constituyó la base técnica sobre la cual se estructuró la propuesta de solución presentada en esta memoria.

Finalmente, como resultado de este proceso, se desarrolló una gramática formal que representa un aporte inicial hacia futuras implementaciones. Esta gramática permite formalizar y estructurar el trabajo realizado, facilitando su lectura y posterior implementación. Para validar su aplicabilidad, se desarrolló una prueba de concepto utilizando Python y el solver Z3, centrada en la operación `reshape`. Aunque se trata de un ejemplo ilustrativo con ciertas simplificaciones, esta implementación demuestra que la gramática puede traducirse fácilmente a un entorno práctico, evidenciando así su potencial como base para sistemas de verificación estática en contextos reales.

En cuanto a líneas de trabajo futuro, esta gramática podría enriquecerse incorporando un enfoque de gradual typing, ampliando su cobertura y agregando soporte para operaciones más complejas. Idealmente, sería posible evolucionar desde un enfoque puramente técnico hacia aplicaciones más prácticas, utilizando la gramática como base estructural.

El aporte presentado permite establecer las restricciones fundamentales dentro de un modelo que favorece su integración con herramientas existentes. Además, el enfoque basado en tipos y la forma en que está estructurada la gramática facilitan su traducción directa a un SMT solver, lo que contribuye a su aplicabilidad en contextos prácticos.

Bibliografía

[1] Martín Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015. `https://www.tensorflow.org/`.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. TACAS*, pages 337–340. Springer, 2008.

[3] Jacob Devlin, Ming-Wei Chang, and Kenton Lee. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proc. NAACL-HLT*, pages 4171–4186, 2019.

[4] Julian Dolby, Avraham Shinnar, and Allison Allain. Ariadne: Analysis for machine learning programs. In *Proc. MAPL*, pages 1–10, 2020.

[5] Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. Gradual tensor shape checking, 2022.

[6] Kaiming He, Georgia Gkioxari, and Piotr Dollár. Mask r-cnn. In *Proc. ICCV*, pages 2980–2988, 2017.

[7] Ho Young Jhoo, Sehoon Kim, and Woosung Song. A static analyzer for detecting tensor shape errors in deep neural network training code. In *Proc. ICSE Companion*, pages 316–317, 2022.

[8] Glenn Jocher. Yolov5: Real-time object detection, 2020. GitHub repository.

[9] Alexander Kirillov, Eric Mintun, and Nikhila Ravi. Segment anything, 2023.

[10] Sifis Lagouvardos, Julian Dolby, and Neville Grech. Static analysis of shape in tensorflow programs. In *Proc. ECOOP*, pages 15:1–15:29, 2020.

[11] Junnan Li, Dongxu Li, and Caiming Xiong. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation, 2022.

[12] Zeina Migeed, James Reed, Jason Ansel, and Jens Palsberg. Generalizing shape analysis with gradual types, 2023.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019.

[14] Alec Radford, Jong Wook Kim, and Chris Hallacy. Learning transferable visual models from natural language supervision. In *Proc. ICML*, pages 8748–8763, 2021.

[15] Alec Radford, Jong Wook Kim, and Tao Xu. Robust speech recognition via large-scale weak supervision, 2022.

[16] Robin Rombach, Andreas Blattmann, and Dominik Lorenz. High-resolution image synthesis with latent diffusion models. In *Proc. CVPR*, pages 10684–10695, 2022.

[17] Hugo Touvron, Thibaut Lavril, and Gautier Izacard. Llama: Open and efficient foundation language models, 2023.

[18] Thomas Wolf, Lysandre Debut, and Victor Sanh. Transformers: State-of-the-art natural language processing. In *Proc. EMNLP Demos*, pages 38–45, 2020.

[19] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2, 2019. `https://github.com/facebookresearch/detectron2`.