UNIVERSIDAD DE LOS ANDES

PROYECTO DE GRADO

Presentado ante la ilustre Universidad de Los Andes como requisito final para obtener el Título de Ingeniero de Sistemas

BRAIN-CEMISID VERSIÓN 2.1 - CONSTRUCCIÓN DE UN KERNEL CEREBRO ARTIFICIAL EN PYTHON: CREACIÓN DE UN SOPORTE INTERMEDIO PARA EL PROCESAMIENTO EN PARALELO

Por

Br. Mudafar El Halabi A.

Tutor: Dr. Gerard Páez M.

Mayo 2017

Resumen: El proyecto Brain-CEMISID es fundado en el Centro de Estudios en Microcomputación y Sistemas distribuidos en la Universidad de Los Andes, el cual busca crear un cerebro artificial basado principalmente en redes neuronales de distintos tipos, las cuales permiten emular ciertos comportamientos del cerebro humano tales como el reconocimiento de estímulos visuales y auditivos, adición por memoria, representación de los conceptos de cantidad y orden, lectura comprensiva de palabras y estado mental de la intención. El proyecto está implementado en el lenguaje de programación 'Python' y sigue un estilo de programación estricto que permite modularidad, extensibilidad, reusabilidad, portabilidad y documentación.

Sin embargo todo el calculo requerido por las redes neuronales y sus módulos relacionados se ejecutan de forma secuencial, lo que puede presentar un bajo rendimiento para el procesamiento y la toma de decisión de entradas de datos grandes. Se propone como solución la creación de un soporte intermedio para ejecutar el proceso del calculo requerido por la redes neuronales de forma paralela. Además realizar un proceso de reingeniería de software aplicando el paradigma de la programación paralela.

Palabras claves: Cerebro Artificial, Programación Paralela, Python, Red Neuronal, Modelado Computacional.

ÍNDICE

Agradecimientos

Esté es mi último paso para cumplir una de mis metas, terminar mis estudios de Ingeniería de Sistemas. Ésta carrera me permitió mucho más de lo que inicialmente propuse, vivir excelentes experiencias y conocer amigos y compañeros inolvidables.

Agradezco a la Universidad de Los Andes que me dio la bienvenida al mundo como tal, y la oportunidad que me ha brindado.

Agradezco a mi tutor el Profesor Gerard Páez Monzón, no sólo por la tutoría, sino por cambiar mi visión hacia el mundo y enfrentar la complejidad.

Agradezco a todos los profesores que dejaron marcas notables durante mi carrera, especialmente a : Rodolfo Sumoza, Richard Espinosa,

Agradezco a mi familia, por su apoyo durante toda la carrera.

Agradezco a todos mis amigos.

Capítulo 1

Introducción

1.1 Antecedentes

En las ultimas décadas se han realizado numerosos esfuerzos para emular las capacidades y comportamiento del cerebro una de las maquinas mas maravillosas de la naturaleza, todo esto mediante la realización de varios proyectos que pretenden lograr este desafío. Uno de ellos es el Blue Brain [Var12] dirigido por el Dr. Henry Makran en conjunto con IBM esta idea comenzó en el año 2005 y hasta la fecha ha logrado modelar la mitad del cerebro de un ratón, la idea es realizar una copia fidedigna del cerebro humano.

Por otra parte DeepMind [Gib15] una compañía de inteligencia artificial que fue adquirida por Google en el 2014, realizo una red neuronal que permitía aprender a jugar una cierta cantidad de vídeos juegos de árcade de los años 70 hasta tal punto de llegar a derrotar humanos profesionales.

EL BRAIN-CEMISID es una idea que surge con el propósito de mostrar de que es capaz de realizar el hombre con tan solo los conocimientos y la imaginación, este modelo cerebral esta conformado por una red neuronal [And12] que no es mas que un conjunto de neuronas interconectadas las cuales almacenaran los estímulos percibidos del medio circundante y unas esferas [Bru15], [Gra15], [Mon14] y [Muc15] en las cuales actuaran una cierta cantidad de neuronas que realizaran una tarea en especifico, las cuales juntas lograran emular la percepción de la vista y el oído.

El trabajo en conjunto del BRAIN-CEMISID es una gran oportunidad para tener un breve acercamiento al desarrollo inicial de un pensamiento complejo. La complejidad

de dicho trabajo permite de una forma u otra enfrentarse a la imaginación y la interdisciplinaridad para apropiárselas y reconstruirlas, como eje principal en el desarrollo de una solución.1*

El BRAIN-CEMISID es un proyecto que se ha llevado a cabo durante unos años dando como resultado la creación de varias etapas completadas:

[And12] inicio el Proyecto con la creación de una red neuronal en VLSI, la cual más adelante fue reinterpretada y adaptada al ambiente CUDA de programación paralela, de la mano de [Ran12] cuyo trabajo se tituló "Motor de Neuronas programado en paralelo bajo el ambiente CUDA".

[Mon14] en su trabajo basado en la creación del Estado Sensorial, desarrolla la primera versión del BRAIN-CEMISID, durante el desarrollo del ES se enfoca en la creación de las 3 primeras esferas (Sensorial, Perceptora y Vectorizada), la cuales en conjunto tendrán la capacidad de reconocer caracteres numéricos, mediante la vectorización de imágenes captadas por el sentido de la vista y formaran el aprendizaje de la red neuronal.

[Muc15] continua con el desarrollo del BRAIN-CEMISID, creando la capa del Estado Cerebral o Mundo Cerebral, se resolvieron ambigüedades de la interfaz de comunicación (Esfera Analıtica), se hicieron nuevas redes neuronales para asociar los sentidos de la (vista y el oıdo) en un principio, independientes y asociar sus estímulos (Esfera Relacional) además, de lograr la reacción de la capa operacional que da una reacción ante una serie de estímulos (capa cultural).

[Gra15] su trabajo se basa en la creación de la Esfera de Capacidades, la cual crea una estructura neuronal especifica llamadas Banco Neuronal Geométrico, las cuales tienen capacidades de realizar cálculos geométricos y algebraicos. El objetivo fue implementar en nuestro cerebro artificial conceptos matemáticos básicos que le otorgaran al cerebro la capacidad de suma.

[Bru15] la creación de la capa operacional comprendida por una "esfera perceptora" que genera una analogía del funcionamiento de algunos sentidos del ser humano, específicamente los sentidos de la Vista y el Oıd[And12] para obtener la capacidad de reconocimiento de caracteres, que permita la realización de una lectura comprensiva y se haga una asociación e interpretación con conocimientos previos.

[B16] ensamblar las 6 esferas las cuales traerá como resultado un proceso que simula la vista y el oído, todo lo que se oye y se ve será mostrado al igual que la compresión de palabras y capacidad de cálculo. Con la finalidad de observar lo que está pensando el cerebro

[M16] reestructura global del cerebro y implementación en Python (versión 2.0), además de agregar una nueva esfera intencional, que permite al cerebro tomar decisión de forma autónoma (con libertad) dentro de un conjunto de opciones disponibles.

1.2 Planteamiento del problema

Actualmente el Centro de Estudios en Micro-computación y Sistemas distribuidos CEMISID, se encuentra trabajando en la elaboración de un cerebro artificial, el cual busca emular una de las maquinas más impresionantes creada por la naturaleza como lo es el cerebro humano. Este cerebro cuenta con 7 Esferas concéntricas (Sensorial, Analítica, Cultural, Relacional, de Capacidades, Perceptora e Intencional), así como simular la vista y el oído con la finalidad de observar lo que está pensando el cerebro. El último trabajo BrainCEMISID 2.0, permitió garantizar la mantenibilidad,flexibilidad, extensibilidad, escalabilidad y modularidad, pero desecho la capacidad de procesamiento de las neuronas en paralelo, lo que presenta una baja de rendimiento notable para procesar entradas de datos grandes.

Por ésta razón se propone construir un soporte intermedio para el procesamiento en paralelo, y para mantener la portabilidad del núcleo se propone una flexibilidad de ejecución secuencial si el ambiente de ejecución no cuenta con los recursos necesarios.

1.3 Justificación

A lo largo de la evolución del desarrollo del proyecto BrainCEMISID, se ha enfrentado la complejidad de distintas áreas; también ha permitido ofrecer una gran base para futuros desarrollos y/o investigaciones en distintas área, tales como la computación, la neurociencia, robótica y muchas otras.

Igualmente enfrentar la complejidad del paradigma de programación paralela permite adquirir una comprensión mejor de como el cerebro procesa los datos en sus neuronas, así como usar la creatividad para lograrlo por la ausencia de métodos genéricos para llevar a cabo dicha tarea.

1.4 Objetivos

1.4.1 Objetivo General

Desarrollar un "soporte intermedio" o *middleware* en inglés, que permita mejorar substancialmente el rendimiento del cerebro mediante el procesamiento en paralelo, e implementarlo usando el paradigma de la programación paralela de Python.

1.4.2 Objetivos Espécificos

Realizar un análisis del funcionamiento de cada una de las esferas creadas en las tesis anteriores, con el fin de recrearlas, pero está vez siguiendo el paradigma de programación paralela de python.

Reingeniería de las esferas para procesar sus operaciones neuronales de forma paralela.

Hacer pruebas que permitan verificar la correctitud del funcionamiento del cerebro bajo la programación paralela.

1.5 Metodología

La metodología utilizada para el desarrollo e implementación de este trabajo será el modelo en espiral.

Dicho modelo se puede aplicar a través del ciclo de vida completo de una aplicación, desde el desarrollo del concepto hasta el mantenimiento. El modelo espiral que Boehm [Boe88] propuso originalmente, es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de la construcción de prototipos con los aspectos controlados y sistemáticos del modelo en cascada. Proporciona el material para el desarrollo rápido de versiones incrementales de software.

En [Som05], se expone que cada ciclo del espiral se divide en cuatro regiones bien definidas:

Definición de Objetivos: En esta región se definen los objetivos específicos. Se identifican los riesgos, las restricciones del producto y del proceso. Por último se traza un plan detallado de gestión.

Evaluación y Reducción de Riesgos: Se lleva a cabo un análisis detallado para cada uno de los riesgos y se definen los pasos para reducir dichos riesgos.

Desarrollo y Validación: Se elige un modelo para el desarrollo del sistema dependiendo de los resultados que arroje la evaluación de riesgos.

Planificación: Se evalúa el proyecto hasta el trabajo hecho en el ciclo actual. En caso de necesitar otro ciclo se planifica dicho ciclo.

En resumen usar esta metodología evolutiva permitirá refinar el modelo conceptual en cada iteración y que cada versión parcial de software se acerque cada vez más a la versión estable de forma incremental, donde se evalúan todas las alternativas y se eligen aquellas que minimizan el riesgo y no comprometa la culminación del proyecto.

1.6 Estructura del Documento

La organización del documento es como sigue:

El capítulo 1 presenta la introducción al proyecto de grado. En él se estudian los antecedentes, plantea el problema, justifica el proyecto, establecen los objetivos y aclara la metodología seguida.

El capítulo 2 contiene el marco teórico. Donde se describe brevemente la computación secuencial y la paralela, luego se habla de los sistemas paralelos y su clasificación clásica. Se dan a conocer las características más importantes de la programación paralela y por último se definen los conceptos de: proceso, hilo(virtual) y IOPS. El capítulo 3 primero explica el proceso de foma detallada de la creación del soporte intermedio para el procesamiento en paralelo. luego exponen cada uno de los módulos dóciles para la ejecución en paralelo identificados en sub-secciones donde se explica brevemente el segmento de código hallado, y por último describe el proceso de modificación hecho para cada módulo.

Finalmente, en el capítulo 4 se dan las conclusiones y recomendaciones derivadas del trabajo realizado.

Capítulo 2

Marco Teórico

2.1 Computador Secuencial (clásico)

Tradicionalmente los programas se han ejecutado, y aún actualmente en su gran mayoría, de manera secuencial. En otras palabras, se procesan en computadoras secuenciales conocidas como máquinas Von Neumann. Un programa o algoritmo se ejecuta sobre un procesador a la vez a pesar de componerse el programa de varias partes independientes que luego se integran por su secuencia para completar la ejecución.

2.2 Computador Paralelo

Desde el punto de vista del hardware, casi todos los computadores de la actualidad son paralelos, ya que cuentan con múltiples núcleos o unidades de ejecución.

2.3 Sistemas Paralelos

Según la taxonomía clásica de Flynn [Fly72] los sistemas paralelos se clasifican según el flujo de datos e instrucciones.

- SISD(Single Instruction Single Data) corresponde al caso secuencial. Se tiene un único flujo de instrucciones que se tratan consecutivamente, y se trabaja sobre un único conjunto de datos.
- SIMD(Single Instruction Múltiple Data) donde se considera un único flujo de instrucciones que se tratan consecutivamente, y trabaja sobre un único conjunto de

datos. Es un modelo paralelo en el que trabajan varios elementos de proceso, ejecutando en cada momento la misma instrucción pero sobre un conjunto de datos distintos. Un buen ejemplo son los procesadores gráficos de las compañias Nvidia y ATI (comprada por AMD).

- MISD(Múltiple Instructions Single Data) se ejecutan varios flujos de instrucciones al mismo tiempo actuando sobre un único conjunto de datos. Ninguna de las arquitecturas existentes implementa este modelo.

- MIMD(Multiple Instructions Múltiple Data) es el modelo a seguir por la mayoría de los sistemas en paralelo actuales, y más aún los de propósito general. En este modelo se tienen varias unidades de proceso, cada una con un conjunto de datos asociados distintos y ejecutando un flujo de instrucciones también distinto. Un ejemplo sería el caso donde se tienen varios núcleos (Cores) que comparten memoria y varios hilos (threads) asignados a los cores, los hilos trabajan de manera independiente aunque ejecutan el mismo código, lo que da como consecuencia que hilos distintos vayan por instrucciones del código y además pueden acceder a zonas distintas de datos aunque comparten memoria.

2.4 Programación Paralela

Según Almeida, Giménez, Mantas y Vidal [**Reference16**] Es un método para ejecutar varias instrucciones de manera simultánea, comúnmente el trabajo se distribuye en varios procesadores que trabajan en conjunto para resolver el problema, cada procesador trabaja en una porción de problema y de igual manera los procesos pueden intercambiar datos, a través de las direcciones de memoria compartida o mediante una red de interconexión.

2.4.1 Características

- El paralelismo a nivel de instrucción consiste en poder ejecutar varias instrucciones al mismo tiempo utilizando diferentes técnicas como la segmentación encauzada o el uso de varias unidades funcionales, y se pueden combinar las técnicas entre sı.

- La memoria se divide en bloques, lo que da la posibilidad de estar accediendo a bloques distintos en el mismo momento, posiblemente en un bloque leyendo y en otro escribiendo.

- La memoria esta organizada jerárquicamente, con distintas velocidades de acceso según el nivel donde se encuentra. Típicamente el acceso más rápido es a los

registros, luego en el siguiente nivel se encuentran las memorias cache (que a su vez puede tener varios niveles), a continuación la memoria principal y por último la memoria secundaria.

- La especulación consiste en ejecutar instrucciones distintas simultáneamente, de manera que cuando sea necesario utilizar el resultado de estas no haya que esperar a que se obtengan. Un caso típico es tener una sentencia if con un else, y disponer de 2 unidades en las que se pueden evaluar al mismo tiempo las sentencias de los bloques (if y else) de manera que cuando se evalúa la condición se toma el resultado del bloque correspondiente. También se puede ejecutar sólo el bloque que parece más probable que haya que ejecutar, y cuando se llega al if seguramente tendremos ya disponible el resultado requerido.

- La ejecución fuera de orden consiste en detectar dentro del código instrucciones que no dependan entre sí para ser ejecutadas simultáneamente o en un orden distinto al que aparecen.

2.5 Conceptos

Según Coulouris, George (2012) [Cou12] los siguientes conceptos se pueden definir como sigue:

2.5.1 Proceso

la noción tradicional de sistemas operativos de un proceso es aquel que ejecuta una sóla actividad, pero con el requerimiento de los sistemas distribuidos la noción del mismo fue mejorada para poder ejecutar varias actividades. Hoy día un proceso consiste de un ambiente de ejecución en conjunto con uno o mas hilos (virtuales).

2.5.2 Hilo (virtuales)

También llamado thread en inglés, es la abstracción del sistema operativo de una actividad, el término se deriva de la frase "hilo de ejecución". Un ambiente de ejecución es la unidad de la administración de recursos. Los hilos pueden ser creados o destruidos dinámicamente en tiempo de ejecución.

2.5.3 IOPS

De sus siglas en ingles Operaciones de entrada/salida por segundo, es una medida de rendimiento usada para caracterizar los dispositivos de almacenamientos secundarios, como los son los discos duros, unidades de estado sólido.

Capítulo 3

Creación de un soporte intermedio para el procesamiento en paralelo

En este capítulo se describen los pasos necesarios para la implementación eficaz de la programación paralela en el núcleo del BrainCEMISID.

3.1 Detección del sistema paralelo

El objetivo principal es la determinación de cantidad de procesadores disponibles en el sistema operativo donde será ejecutado el núcleo del BrainCEMISID, sean físicos o bien virtuales -también llamadas núcleos o hilos reales- con el fin de calibrar el número de conjunto de hilos -recursos inicializados listos para usar – o thread pool en inglés, el cual permitirá lograr una ejecución de las distintas partes del núcleo en paralelo, de forma optimizada para los recursos disponibles en el computador donde es ejecutado.

3.1.1 Compatibilidad

La detección es compatible con los siguientes sistemas operativos: Unix, Linux, Windows, BSD y Solaris, además es compatible con las siguientes bibliotecas: Multiprocessing, Cpuset, Psutil, Posix y jython, con el fin de ofrecer una mayor portabilidad y confiabilidad a la herramienta.

3.1.2 Algoritmo

Primero se determina el sistema operativo donde es ejecutado el núcleo del BrainCEMISID, luego se inicia una comunicación directa con las herramientas integradas en dicho sistema

operativo, solicitando el número de procesadores disponibles en el sistema, si la solicitud no es respondida se intentará usar las bibliotecas compatibles con el sistema una por una haciendo la solicitud de nuevo, finalmente si ninguna biblioteca facilita la información requerida, se devuelve una excepción que describe el hecho de no poder determinar el número de procesadores.

3.1.3 Determinación de números de hilos

El número de hilos (virtuales) a usar inicialmente es igual a la cantidad de procesadores presentes en el sistema de ejecución, luego para las operaciones de entrada/salida con las unidades de almacenamiento secundario se incrementa la cantidad de hilos según el siguiente algoritmo:

```
sea n_es  el número total de operaciones de entrada/salida de un bloque de código dado,
y sea n_p el número total de procesadores detectados del sistema,
y sea n_h el número total de hilos virtuales

Si  n_es  <=  n_p, entonces  n_h  será igual a n_p
Si  n_es  > n_p, entonces n_h será igual a n_es y 70  como umbral.
```


70 es la media experimental de operaciones de entrada/salida por segundo (IOPS) para unidades de almacenamiento secundarias tradicionales o magnéticas [Ian12]

Por ejemplo: si se tienen 8 operaciones de entrada/salida de almacenamiento secundario en un bloque de código dado y 4 procesadores presentes en el sistema, entonces el número de hilos a usar es 8, el equivalente a sólo dos operaciones secuenciales; éste resultado se puede obtener fácilmente al dividir la cantidad de hilos entre el número de procesadores: $8/4 = 2$

3.2 Identificación de módulos dóciles para la ejecución en paralelo

Este paso es muy crítico e importante para la posterior reingeniería de cada módulo, aunque existan criterios generales para la identificación de secciones de código para ser ejecutados en paralelo, éstos no son suficientes y tampoco correctos para todos los casos,

por lo que se debe tener experiencia y usar la intuición para lograr una ejecución en paralelo eficaz y correcta.

Para llevar acabo ésta tarea primero se analizan las secciones de código especialmente los ciclos de repetición, luego se busca la posibilidad del manejo de datos y/o operaciones de entrada/salida de almacenamiento de forma paralela.

El las siguientes subsecciones se muestran los resultados de los estudios de cada módulo.

3.2.1 KernelBrainCemisid

3.2.1.1 Constructor __init__

En el constructor de la clase, se hace la carga inicial de la memoria persistente de las destinas redes neuronales del núcleo, las cuales son operaciones de entrada/salida de almacenamiento secundario.

Dicha carga de cada clase o módulo es totalmente independiente de las otras, por lo que es posible llevar a cabo el proceso de forma paralela sin enfrentar problemas de carreras en la data cargada.

3.2.1.2 _bip_addition

En el método privado bip de adición del protocolo 'Bum-Bip-Check-Clack', la carga del conocimiento de los sensores visuales y de oídos se hace de forma secuencial (ciclo de repetición), y el resultado de cada conocimiento se guarda en un arreglo.

3.2.1.3 sight_recognize

En el método publico del reconocimiento de vista, específicamente al encontrar un estado de confusión -esto es se reconocen dos o mas figuras parecidas- la obtención de las relaciones de vista se hace de forma secuencial para cada identificador del conjunto de neuronas reconocidas.

3.2.2 RelNetwork

3.2.2.1 Constructor __init__

En el constructor de la clase, la lista de neuronas se genera de forma secuencial, la longitud de la lista lo define el parámetro de entrada del constructor 'neuron_count'.

3.2.2.2 learn

Es un método público que permite adquirir nuevo conocimiento listo para aprender, de forma análoga a 3.2.2.1 la lista se genera de forma secuencial, sólo que la cantidad de neuronas a generar lo define la longitud de la lista de neuronas.

3.2.2.3 get_hearing_rels get_sight_rels

Éstos dos métodos son muy similares, obtienen las relaciones de oído y vista respectivamente, en ambos métodos se hace de forma secuencial; pero se verifica primero que se reconoce el sonido o la vista respectivamente y luego se obtiene y se agrega el conocimiento a la lista correspondiente.

3.2.3 UncosciousFilteringblock

3.2.3.1 _filter_biology

Es un método privado, permite seleccionar la mejor entrada para la parte biológica. Se calcula de forma secuencia cada entrada en búsqueda de la distancia mínima entre la suma del estado interno biológico con el estado biológico de cada entrada, dividida por el estado biológico deseado.

3.2.3.2 _filter_culture

En éste método privado se presenta una lógica de programación análoga a 3.2.3.1, a diferencia que no se calcula una distancia, sino se busca la memoria con el estado cultural máximo.

3.2.3.3 _filter_feelings

Éste método privado es análoga a 3.2.3.2, a diferencia que se busca la memoria con el mejor sentimiento (máximo).

3.2.4 CosciousDecisionsblock

3.2.4.1 Constructor __init__

En el constructor de la clase, se prepara un conjunto de entrenamiento, el cual usa valores aleatorios en tres etapas para generar de forma secuencial el valor de elección de estado biológico, cultural y sentimental y el de predicción.

3.2.5 CulturalNetwork

3.2.5.1 Constructor __init__

En el constructor de la clase la creación de grupos culturales es secuencial.

3.2.5.2 resize

Éste método se encarga de llenar la lista de neuronas con memorias, de forma análoga a 3.2.5.1 los grupos culturales se crean de forma secuencial.

3.2.6 DecisionsBlock

3.2.6.1 get_output_memory

En éste método público se obtienen los grupos culturales y la memoria de la cual la decisión se puede inferir, justamente el calculo de las entradas conscientes para inferir éstas últimas se hace de forma secuencial.

3.2.7 EpisodicMemory

3.2.7.1 retrieve_exact_memory

Éste método publico retorna la memoria exacta, excepto para el último elemento del disparador, éstos son procesados de forma secuencial.

3.3 Reingeniería de módulos usando código paralelo

Para hacer la reingeniería, se usa la biblioteca multiprocessing de python por su versatilidad y amplia compatibilidad con los distintos sistemas operativos y plataformas de computación tanto para sistemas paralelos como para sistemas clásicos o secuenciales; también por ofrecer los conjuntos de hilos preparados o thread's pool en ingles, que pueden ejecutarse eficientemente de forma asíncrona, además reducen el número de hilos de la aplicación y ofrecen una forma para administrar los hilos.

Por limitaciones de la bibilioteca oficial de python, se usa una bifurcación (o fork en inglés) de multiprocessing llamada Pathos Framework que ofrece mediante el uso de dill (utilidad para serializar cualquier parte de pyhton) la habilidad de ejecutar en paralelo cualquier sección de código dentro de una clase así como a las funciones que accedan a parámetros internos de una clase (self en python), donde en ambos casos la oficial no cuenta con dicha habilidad.

Fases de reingeniería de cada módulo: 1. Detección del sistema para determinar número de procesadores disponibles 2. Para operaciones de entrada/salida se usa el algoritmo de la sección 3.1.3 para determinar la cantidad de hilos a usar en el conjunto de hilos preparados. 3. Reestructuración y/o re-escritura del código si es necesario, para hacerlo ejecutar de forma paralela. 4. Verificación de la correctitud de la modificación, haciendo varias consultas al Núcleo BrainCemisid.

En las siguientes subsecciones se muestran los detalles del análisis de código para cada método de cada clase.

3.3.1 KernelBrainCemisid

3.3.1.1 Constructor __init__

Cada carga de memoria se reescribe en forma de función lambda para ser usada como parámetro para el método 'apply_async' de la clase Pool de la biblioteca 'pathos'. 'apply_async' permite una ejecución concurrente no bloqueante.

Nota: En esta tarea no se puede ejecutar las instrucciones en paralelo de forma tradicional, donde la variante es la data de entrada, en cambio en este caso la variante es una carga de archivos de memorias persistentes de clases distintas, por lo que el método 'apply_async' es el apropiado para evitar la carga secuencial de ese tipo de tareas.

3.3.1.2 _bip_addition

Este método hace llamada al método get_hearing_rels 3.2.2.3, que ya se modificó para ejecutarse en paralelo, y por limitaciones de la biblioteca multiprocessing de Python, no se puede ejecutar un proceso en paralelo que a su vez tiene sub-procesos que se ejecuten en paralelo.

Debido a ésta limitación se debe elegir a cual de los dos métodos ejecutar en paralelo, y el criterio seleccionado es ejecutar el proceso con mayor peso computacional en paralelo.

3.3.1.3 sight_recognize

En éste método ocurre lo mismo que en 3.3.1.2, por lo que se elige al sub-método get_sight_rels 3.2.2.3 para ser ejecutado en paralelo, de forma análoga por ser el método con mayor peso computacional.

3.3.2 RelNetwork

3.3.2.1 Constructor __init__

En éste método se encapsula el código para generar cada relación neuronal en una función lambda, que recibe el contador como parámetro.

Ésta función será ejecutada en paralelo usando el método 'map' de 'multiprocessing', y dará como resultado una lista de neuronas.

3.3.2.2 learn

Éste método es análogo a 3.3.2.1, a diferencia que la cantidad de neuronas a preparar depende de la lista de neuronas.

3.3.2.3 get_hearing_rels get_sight_rels

En ambos métodos se presenta un caso especial, donde se requiere hacer una verificación antes de agregar el valor a la lista, pero por limitaciones del método 'map' siempre se debe retornar un valor para cada entrada, lo que supone un problema para la consistencia de datos para ambas listas de relaciones a retornar, la solución aplicada se describe mediante el siguiente algoritmo de dos fases:

I. Si se reconoce el identificador se retorna el conocimiento, de lo contrario se retorna un valor nulo 'None' II. Se filtran los posibles valores nulos de la lista de relaciones correspondiente.

Nota: el paso II. Es importante para mantener la consistencia de los valores de las variables y su procesamiento.

Se encapsula el código responsable de obtener la relación por oído y por vista respectivamente en una función lambda, que retorna el conocimiento relacionada si la neurona correspondiente reconoce el identificador de oído o vista respectivamente, de lo contrario se retorna un valor nulo.

Cada función lambda será ejecutada en paralelo usando el método 'map' de 'multiprocessing', que dará como resultado una lista de relaciones de oído y vista respectivamente, por último ambas listas se filtran para eliminar los valores nulos.

3.3.3 UncosciousFilteringblock

3.3.3.1 _filter_biology

En esté método se presenta una estructura de programación clásica, donde se calcula un valor inicial, luego se itera sobre el resto de entradas haciendo el mismo cálculo buscando el mínimo o el máximo según sea el caso. Al ejecutar en paralelo la misma estructura se puede presentar la condición de carrera, que lo más probable dará un resultado errado; por esto se re-estructuro esa sección como sigue, con el objetivo de evitar la ocurrencia de dicha condición.

El proceso iterativo se encapsula en una función privada que recibe un sólo parámetro, en este caso una memoria; se obtiene el conocimiento desde la memoria, luego se calcula la distancia, finalmente se retorna una dupla con la distancia calculada y la memoria. Ésta función será ejecutada en paralelo usando el método 'map' de 'multiprocessing', con la lista de todas las memorias, al terminar se calcula la distancia mínima , y se retorna la mejor biología.

3.3.3.2 _filter_culture

Éste método es análogo a 3.3.3.1, a diferencia que en vez de calcular la distancia se obtiene directamente el valor de cultura usando get_culture() para la memoria dada, por último se calcula la máxima cultura desde la dupla.

3.3.3.3 _filter_feelings

Éste método es idéntico a 3.3.3.2, sólo que se calcula el sentimiento en vez de la cultura, pero a nivel de código sólo cambian los nombres de las variables y los métodos, por lo que se aplica lo mismo.

3.3.4 CosciousDecisionsblock

3.3.4.1 Constructor __init__

Primero se encapsula el código de la generación del conjunto de entrenamiento, en una función privada, que recibe un sólo parámetro, que permite limitar la cantidad a generar. Luego la función es ejecutada con 'Pool' de 'multiprocessing', para generar el conjunto de entrenamiento en paralelo, por último se asigna el resultado de la ejecución de 'Pool' a la misma variable 'training_set'.

3.3.5 CulturalNetwork

3.3.5.1 Constructor __init__

En éste constructor, se encapsula el código de generación de grupo cultural en una función lambda, que tiene como único parámetro a 'index', para contar la cantidad a generar, luego la función es ejecutada en 'Pool' de 'multiprocessing' para obtener una lista de grupos culturales de forma paralela.

3.3.5.2 resize

De forma análoga a 3.3.5.1, se obtiene la lista de los grupos culturales de forma paralela.

3.3.6 DecisionsBlock

3.3.6.1 get_output_memory

Primero se encapsula el código en una función lambda, con la memoria como el único parámetro de entrada, para obtener el conocimiento relacionado a dicha memoria.

Luego ésta función es ejecutada en un 'Pool' de 'multiprocessing' para obtener una lista de entradas conscientes que a su vez serán usadas para obtener la salida consciente.

3.3.7 EpisodicMemory

3.3.7.1 retrieve_exact_memory

En esté método se presenta otra estructura de programación poco común, donde se itera sobre un rango procesando el método 'bip' del protocolo'bbcc' en todas las iteraciones excepto la último, donde ahora se usa el método 'check' del mismo protocolo con el último disparador para obtener la memoria exacta.

Para la ejecución de éste código en paralelo, primero se separan el proceso en dos tareas:

- la primera procesa los 'bip' de todos los disparadores excepto el último, ésta tarea se encapsula en una función lambda, con el indice del disparador como parámetro único de entrada, la cual es ejecutada en un 'Pool' de 'multiprocessing'.
- la segunda se encarga de retornar la memoria exacta del último disparador.

Capítulo 4

Conclusiones y recomendaciones

4.1 Conclusiones

Desde el inicio del proyecto Brain-CEMISID se ha enfrentado la complejidad de distintas paradigmas, en este proyecto de grado cómo los otros se presentaban dos desafíos; el primero era mejorar el rendimiento del núcleo mediante la reingeniería de software, pero ésta vez usando un paradigma de programación paralela. El segundo, mantener la portabilidad que ofrece el núcleo, así como no alterar las estructuras de códigos presentes. Los desafíos no cuentan con manuales o pasos a seguir, por lo que se debe usar la creatividad, hacer profundos análisis de código y usar pruebas de ensayo y error para poder enfrentar dichos desafíos. También fue requerido lidiar con las limitaciones de Python para ejecutar ciertas tareas en paralelo. Ahora el núcleo cuenta con un módulo especifico para la detección del sistema paralelo, donde de forma automatizada se puede determinar los parámetros más importantes de dicho sistema, como lo son los números de procesadores y de hilos reales o thread en inglés. El mismo modulo también ofrece generar más hilos para operaciones bloqueantes como los son las llamadas a dispositivos de almacenamiento secundario. La nueva versión es perfectamente escalable, detecta y decide de forma automática la cantidad de hilos y/o procesadores presentes en el sistema a usar, ofreciendo una mejora substancial en el rendimiento para futuras aplicaciones donde las entradas de datos pueden ser vectores de tamaños considerablemente más grandes. Se mantiene el estilo estricto para el código y la documentación propuesto en el trabajo anterior, por lo que los módulos se pueden usar para diferentes aplicaciones y/o extender sus funcionalidades. El nuevo Brain-CEMISID a pesar de ejecutar sus tareas en paralelo, se mantiene portable y se puede ejecutar en cualquier computador con intérprete de python. Se sigue la documentación y el diseño estricto propuesto por la

versión anterior, permitiendo una mayor comprensión y una adopción más rápida de los módulos; con el fin de acelerar cualquier proceso de desarrollo del núcleo.

4.2 Recomendaciones

Agregar a la interfaz la habilidad de cargar imágenes desde archivos, así como la habilidad de guardar (exportar) la figura dibujada en la malla vectorial de la vista y del oído en un archivo, para su posterior uso en futuras operaciones. La entrada se puede extender para usar una cámara conectada al computador, donde se pueden capturar fotos y pasarlas directamente a la malla de vista.

Implementar un módulo que transforma un sonido grabado en formato digital (por ejemplo wav), a un flujo crudo de bits (sin formato), luego habilitar una entrada para el sensor de oído que permita cargar dicho flujo en el la malla vectorial actual. También se debe poder reproducir el sonido guardado en la malla.

Hacer una interfaz de comandos o CLI en ingés para extender el manejo de las funciones internas del cerebro desde una consola interprete de comandos (por ejemplo CMD), con el objetivo de permitir a cualquier usuario poder interactuar con las distintas redes neuronales y/o funciones del núcleo sin tener que saber programar (en pyhton en este caso).

Usar el núcleo del cerebro como un módulo en un robot experimental, que pueda percibir su entorno -esto es equipado con una cámara y un micrófono- con el objetivo de ofrecerle una toma de decisión libre basada en el estado interno del núcleo.

Bibliografía

[Fly72] M. Flynn. “Some computer organizations and their effectiveness. IEEE Trans. Comput., C-21:948.” In: (1972).

[Boe88] B.W. Boehm. “Software Risk Management Tutorial, Computer Society”. In: (Apr. 1988).

[Som05] I. Sommerville. “Ingenieria de Software. Pearson Educación S.A, Madrid, España, 7 edition.” In: (2005).

[And12] L. Andrade. “Creación de una Red Neuronal en VLSI. Proyecto de Grado, Universidad de Los Andes.” In: (2012).

[Cou12] George Coulouris. “Distributed Systems Concepts and Design.” In: (2012).

[Ian12] Ianatkin. “Getting The Hang Of IOPS v1.3”. In: (2012). URL: https://www.symantec.com/connect/articles/getting-hang-iops-v13#a08.

[Ran12] C. Rangel. “Motor de Neuronas programado en paralelo Bajo el ambiente CUDA. Proyecto de Grado, Universidad de Los Andes.” In: (2012).

[Var12] Varios. “Blue brain project”. In: (2012). URL: http://blue-brain-projhttp://www.artificialbrains.com/ect.

[Mon14] J. Monsalve. “BRAIN-CEMISID versión 1.0. Diseño de un núcleo operacional para la construcción de un kernel-cerebro artificial implementado en programación paralela en el ambiente CUDA: Creación del Estado Sensorial. Proyecto de Grado, Universidad de Los Andes.” In: (2014).

[Bru15] R. Bruzual. “BRAIN-CEMISID version 1.4 Diseño de un núcleo operacional para la construcción de un kernel-cerebro artificial implementado en programación paralela en el ambiente CUDA: Creación de la Lectura Comprensiva de Palabras. Proyecto de Grado, Universidad de Los Andes.” In: (2015).

[Gib15] lessons E. Gibney. “Game-playing neuroscience. Software holds”. In: (2015). URL: http://www.nature.com/news/game-playing-software-holds-lessons-for-neuroscience-1.16979.

[Gra15] R. Graterol. “BRAIN-CEMISID versión JJ. versión R. Diseño de un núcleo operacional para la construcción de un kernel-cerebro artificial implementado en programación paralela en el ambiente CUDA: Creacion de la Esfera de Capacidades. Proyecto de Grado, Universidad de Los Andes”. In: (2015).

[Muc15] J. Muchacho. “BRAIN-CEMISID versión JJ. Diseño de un núcleo opera- cional para la construcción de un kernel-cerebro artificial implementado en programación paralela en el ambiente CUDA: Creación del Mundo Cerebral. Proyecto de Grado, Universidad de Los Andes.” In: (2015).

[B16] Sosa B. “BRAIN-CEMISID versión 1.5 Construcción de un kernel cerebro artificial implementado en programación paralela en el ambiente CUDA: Ensamblaje de las Esferas Sensorial, Analıtica, Cultural, Relacional, de Capacidades y Perceptora”. In: (2016).

[M16] Fernández M. “Brain-CEMISID Versión 2.0 - Construcción de un Kernel Cerebro Artificial en Python: Reingenieria de Software y Creación del Estado Mental de la Intencióna”. In: (2016).