Documentación RNTA.jl

Introducción al Proyecto

RNTA.jl (Red Neuronal Tensorial Adaptativa) es un proyecto ambicioso en Julia que implementa un nuevo paradigma de deep learning basado en representaciones tensoriales tridimensionales y dinámicas inspiradas en estructuras cerebrales. A diferencia de las redes neuronales tradicionales, RNTA utiliza campos tensoriales volumétricos y mecanismos de atención espacial para crear modelos con mayor capacidad de razonamiento y adaptación.

Objetivo del Proyecto

El objetivo principal de RNTA.jl es desarrollar una biblioteca que permita construir la inteligencia artificial más poderosa del mundo mediante:

1. Una arquitectura neuronal volumétrica (3D) que supera las limitaciones de las redes neuronales tradicionales basadas en capas planas
2. Mecanismos de adaptación dinámica inspirados en la neuroplasticidad cerebral
3. Sistemas de deliberación interna y diálogo que permiten un razonamiento más profundo
4. Representaciones tensoriales ricas que capturan relaciones semánticas complejas
5. Optimización automática de la arquitectura según las tareas y datos

Aplicaciones Potenciales

Las capacidades de RNTA.jl permitirían revolucionar múltiples campos:

- **Procesamiento de lenguaje natural**: Sistemas con mayor comprensión contextual y capacidad de razonamiento
- **Razonamiento científico**: Modelos capaces de entender y aplicar teorías científicas complejas
- **Planificación estratégica**: Sistemas que pueden evaluar múltiples escenarios y consecuencias
- **Creación artística**: Generación de contenido con coherencia estructural y profundidad conceptual
- **Investigación científica**: Asistentes con capacidad para plantear hipótesis y diseñar experimentos
- **Medicina personalizada**: Análisis multidimensional de datos médicos para diagnósticos precisos

Documentación de Módulos y Archivos

acceleration/CUDATensors.jl

- **Propósito**: Implementa operaciones tensoriales aceleradas por CUDA para RNTA, permitiendo aprovechar GPUs NVIDIA para cálculos intensivos.
- **Funciones**:
  - `init_cuda_tensors()`: Inicializa el sistema de aceleración CUDA para RNTA.
  - `use_cuda_tensors(active=true)`: Activa o desactiva el uso de CUDA para operaciones tensoriales.
  - `is_cuda_active()`: Verifica si la aceleración CUDA está activa.
  - `to_cuda(tensor)`: Transfiere un tensor a la GPU si CUDA está activo.
  - `to_host(tensor)`: Transfiere un tensor de la GPU a la CPU.
  - `cuda_tensor_convolution(input, kernel; stride=(1,1,1), padding=0)`: Implementación acelerada por CUDA de la convolución tensorial 3D.
  - `cuda_zero_pad(tensor, padding)`: Implementación acelerada por CUDA para añadir padding de ceros a un tensor.
  - `cuda_adaptive_pooling(input, output_size; mode=:max)`: Implementación acelerada por CUDA del pooling adaptativo.
  - `cuda_volumetric_activation(tensor; type=:adaptive_tanh, parameters=nothing)`: Implementación acelerada por CUDA de las activaciones volumétricas.
  - `cuda_adaptive_tanh(tensor, slope_factor)`: Implementación CUDA de la activación adaptive_tanh.
  - `cuda_tensor_relu(tensor, alpha, sine_factor)`: Implementación CUDA de la activación tensor_relu.
  - `cuda_spatial_attention_transform(input, attention_map)`: Implementación acelerada por CUDA de la transformación atencional.

acceleration/HardwareAdaptation.jl

- **Propósito**: Proporciona funcionalidades para adaptar y optimizar la ejecución de RNTA a diferentes tipos de hardware (CPU, GPU, etc.).
- **Funciones**:
  - `detect_hardware()`: Detecta y caracteriza el hardware disponible en el sistema.
  - `optimize_for_hardware(brain_space, profile)`: Optimiza la configuración del espacio cerebral para el hardware disponible.
  - `configure_memory_for_gpu(gpu_memory_gb)`: Configura parámetros de memoria óptimos para GPU.

- `configure_memory_for_cpu(cpu_memory_gb)`: Configura parámetros de memoria óptimos para CPU.
- `configure_precision_for_hardware(profile)`: Configura la precisión óptima basada en el hardware.
- `select_parallelism_strategy(brain_space, profile)`: Selecciona la estrategia de paralelismo óptima para el hardware y modelo.
- `estimate_model_size(brain_space)`: Estima el tamaño del modelo en GB.
- `determine_optimal_batch_size(brain_space, profile)`: Determina el tamaño de batch óptimo para el modelo y hardware.
- `apply_hardware_optimizations(brain_space, profile, precision, parallelism)`: Aplica optimizaciones específicas del hardware al modelo.
- `set_compute_precision(brain_space, precision)`: Configura la precisión de cómputo para un espacio cerebral.
- `configure_parallelism(brain_space, parallelism)`: Configura estrategia de paralelismo para un espacio cerebral.
- `optimize_for_cuda(brain_space, profile)`: Aplica optimizaciones específicas para GPUs NVIDIA.
- `optimize_for_cpu(brain_space, profile)`: Aplica optimizaciones específicas para CPUs.
- `adapt_execution_plan(brain_space, profile)`: Crea un plan de ejecución optimizado para el hardware disponible.
- `map_operators_to_devices(plan, brain_space, profile)`: Asigna operadores a dispositivos específicos.
- `create_communication_plan(brain_space, profile, strategy)`: Crea un plan de comunicación para operaciones multi-dispositivo.
- `create_kernel_optimizations(profile)`: Crea optimizaciones de kernel específicas para el hardware.
- `benchmark_operations(brain_space, profile)`: Realiza benchmarks de operaciones clave para afinar el rendimiento.
- `get_optimal_batch_size(brain_space, profile, max_memory_usage_fraction=0.7)`: Determina el tamaño de batch óptimo para maximizar uso de hardware sin OOM.
- `enable_mixed_precision(brain_space)`: Habilita el uso de precisión mixta en el espacio cerebral.

- `setup_multi_device(brain_space, num_devices, strategy=:auto)`: Configura la distribución en múltiples dispositivos.
- `configure_for_distributed(brain_space, num_nodes, gpus_per_node)`: Configura el modelo para entrenamiento distribuido multi-nodo.
- `configure_for_data_parallel(brain_space, num_devices)`: Configura el espacio cerebral para paralelismo de datos.
- `configure_for_model_parallel(brain_space, num_devices)`: Configura el espacio cerebral para paralelismo de modelo.
- `configure_for_pipeline_parallel(brain_space, num_devices)`: Configura el espacio cerebral para paralelismo de pipeline.

acceleration/MemoryOptimization.jl

- **Propósito**: Proporciona estrategias y herramientas para optimizar el uso de memoria en RNTA, incluyendo compresión de tensores, pools de memoria y estimación de uso.
- **Funciones**:
  - `get_global_memory_pool(max_pool_size_mb=1024)`: Obtiene o inicializa el pool de memoria global.
  - `allocate_from_pool(shape, device=:cpu)`: Asigna un tensor desde el pool de memoria, o crea uno nuevo si no hay disponible.
  - `release_to_pool(tensor, device=:cpu)`: Devuelve un tensor al pool para ser reutilizado.
  - `compress_tensor(tensor, scheme)`: Comprime un tensor según el esquema especificado.
  - `enable_gradient_checkpointing(brain_space, checkpoint_layers=:auto)`: Habilita el checkpointing de gradientes para ahorrar memoria.
  - `track_memory_usage(func)`: Monitorea el uso de memoria durante la ejecución de una función.
  - `optimize_memory_usage(brain_space, config)`: Aplica estrategias de optimización de memoria al espacio cerebral.
  - `configure_memory_strategy(brain_space, available_memory_gb)`: Configura estrategias de memoria basadas en la disponibilidad.

acceleration/TensorParallelism.jl

- **Propósito**: Implementa mecanismos para paralelizar operaciones tensoriales en múltiples dispositivos o núcleos, distribuyendo eficientemente tanto los datos como las operaciones.
- **Funciones**:

- `configure_parallel_environment(num_devices; device_type=:auto)`: Configura el entorno paralelo para utilizar los dispositivos disponibles de manera óptima.
- `create_device_mapping(available_devices)`: Crea un mapeo de IDs lógicos a dispositivos físicos.
- `optimize_partition_scheme(tensor_shape, operation_type, num_devices)`: Determina el esquema de partición óptimo para un tensor y operación dados.
- `optimize_block_partition(tensor_shape, num_devices)`: Optimiza una partición en bloques para un tensor dado.
- `factorize_close_to_cube(n)`: Factoriza un número en tres factores lo más cercanos posible a un cubo.
- `optimize_overlapping_partition(tensor_shape, operation_type, num_devices)`: Optimiza una partición con superposiciones para operaciones con dependencias locales.
- `optimize_dimension_partition(tensor_shape, num_devices)`: Optimiza una partición a lo largo de una dimensión específica.
- `create_tensor_partitions(tensor, scheme, num_devices)`: Crea particiones de un tensor según el esquema especificado.
- `create_block_partitions(tensor_shape, scheme, num_devices)`: Crea particiones en bloques 3D.
- `create_layer_partitions(tensor_shape, scheme, num_devices)`: Crea particiones por capas a lo largo de una dimensión específica.
- `create_dimension_partitions(tensor_shape, scheme, num_devices)`: Crea particiones a lo largo de una dimensión específica con puntos de corte definidos.
- `parallelize_operation(tensor, op_func, env_config; operation_type=:generic, custom_scheme=nothing)`: Paraleliza una operación en un tensor distribuido.
- `distribute_tensor(tensor, partitions, env_config)`: Distribuye un tensor entre múltiples dispositivos.
- `gather_results(results, partitions, original_tensor)`: Combina los resultados paralelos en un único tensor.
- `parallel_tensor_contraction(tensor_a, tensor_b, env_config)`: Realiza contracción tensorial en paralelo.
- `contract_tensors(subtensor_a, tensor_b)`: Función auxiliar para contracción tensorial.
- `parallel_volumetric_attention(tensor, attention_params, env_config)`: Aplica mecanismo de atención volumétrica en paralelo.

- `apply_volumetric_attention(subtensor, attention_params)`: Aplica atención volumétrica a un subtensor.

adaptation/PlasticityRules.jl

- **Propósito**: Implementa reglas de plasticidad inspiradas en neurociencia para ajustar dinámicamente las conexiones en la red neuronal tensorial.
- **Funciones**:
  - `apply_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica la regla de plasticidad especificada en el contexto a una neurona.
  - `apply_plasticity!(connection, pre_activation, post_activation, context)`: Aplica la regla de plasticidad especificada en el contexto a una conexión.
  - `apply_hebbian_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica plasticidad Hebbiana a una neurona.
  - `apply_hebbian_plasticity!(connection, pre_activation, post_activation, context)`: Aplica plasticidad Hebbiana a una conexión.
  - `apply_bcm_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica plasticidad BCM (Bienenstock-Cooper-Munro) a una neurona.
  - `apply_bcm_plasticity!(connection, pre_activation, post_activation, context)`: Aplica plasticidad BCM a una conexión.
  - `apply_stdp_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica plasticidad STDP (Spike-Timing-Dependent Plasticity) a una neurona.
  - `apply_stdp_plasticity!(connection, pre_activation, post_activation, context)`: Aplica plasticidad STDP a una conexión.
  - `apply_homeostatic_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica plasticidad homeostática para mantener niveles de actividad estables.
  - `apply_homeostatic_plasticity!(connection, pre_activation, post_activation, context)`: Aplica plasticidad homeostática a una conexión.
  - `apply_contextual_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica plasticidad contextual sensible al entorno.
  - `apply_contextual_plasticity!(connection, pre_activation, post_activation, context)`: Aplica plasticidad contextual a una conexión.
  - `apply_mixed_plasticity!(neuron, pre_activation, post_activation, context)`: Aplica una combinación ponderada de diferentes reglas de plasticidad.

- `apply_mixed_plasticity!(connection, pre_activation, post_activation, context)`: Aplica combinación de reglas a una conexión.
- `normalize_kernel!(kernel)`: Normaliza un kernel para evitar crecimiento descontrolado de pesos.
- `create_modulation_tensor(brain, type=:attention)`: Crea un tensor de modulación para plasticidad contextual.
- `apply_brain_plasticity!(brain, input_tensor, output_tensor, rule_type=HebbianRule)`: Aplica plasticidad a todas las neuronas y conexiones en el cerebro.
- `extract_neuron_input(brain, neuron, input_tensor)`: Extrae la entrada específica para una neurona desde el tensor de entrada global.
- `extract_tensor(tensor, field)`: Extrae la región del tensor correspondiente al campo espacial.
- `spatial_gradients(tensor)`: Calcula los gradientes espaciales de un tensor 3D.

adaptation/SelfPruning.jl

- **Propósito**: Implementa mecanismos de auto-optimización y poda de conexiones para mantener la eficiencia de la red.
- **Funciones**:
  - `self_prune!(brain, params=PruningParameters())`: Ejecuta el proceso de auto-poda en el espacio cerebral completo.
  - `activity_based_pruning(connections, params)`: Realiza poda basada únicamente en niveles de actividad.
  - `importance_based_pruning(brain, params)`: Realiza poda basada en importancia relativa de conexiones.
  - `balanced_pruning(brain, params)`: Realiza poda manteniendo un balance entre conexiones excitatorias e inhibitorias.
  - `adaptive_pruning(brain, params)`: Realiza poda adaptativa combinando múltiples estrategias.
  - `calculate_connection_importance(brain)`: Calcula la importancia relativa de cada conexión en el cerebro.
  - `calculate_neuron_centrality(brain, neuron_id)`: Calcula la centralidad de una neurona basada en sus conexiones.
  - `get_neuron_specialization(brain, neuron_id)`: Obtiene el nivel de especialización de una neurona.
  - `analyze_connections(brain)`: Analiza la estructura de conexiones del cerebro.

- `optimize_connection_strengths!(brain)`: Optimiza las fuerzas de conexión basado en su actividad.
- `redistribute_connection_resources!(brain, pruning_report)`: Redistribuye recursos de conexiones podadas para fortalecer conexiones importantes.

adaptation/Specialization.jl

- **Propósito**: Implementa mecanismos de especialización de neuronas tensoriales, permitiendo que estas se adapten y especialicen en funciones específicas.
- **Funciones**:
  - `specialize_neurons!(brain, params=SpecializationParameters())`: Aplica el proceso de especialización a todas las neuronas del cerebro.
  - `update_specialization!(neuron, params)`: Actualiza el nivel de especialización y posiblemente el tipo funcional de una neurona.
  - `calculate_activation_consistency(activation_history)`: Calcula la consistencia de los patrones de activación recientes.
  - `analyze_brain_specialization(brain)`: Analiza las estadísticas de especialización para todo el cerebro.
  - `identify_specialized_regions(brain, threshold=0.6)`: Identifica regiones de neuronas altamente especializadas.
  - `find_specialized_clusters(specialization_map, threshold)`: Encuentra clusters de alta especialización en el mapa 3D.
  - `grow_cluster!(binary_map, label_map, start_pos, label, positions)`: Expande un cluster desde una posición inicial usando búsqueda en anchura.
  - `register_functional_regions!(brain, threshold=0.7)`: Registra regiones funcionales en el cerebro basadas en especialización.
  - `specialize_connections!(brain, connections_threshold=0.5)`: Especializa las conexiones basadas en los tipos funcionales de las neuronas.
  - `specialize_connection!(connection, source_type, target_type)`: Modifica una conexión según los tipos funcionales de las neuronas conectadas.

adaptation/DynamicExpansion.jl

- **Propósito**: Implementa mecanismos para expansión dinámica del espacio cerebral, permitiendo que la red crezca en áreas que lo necesitan.
- **Funciones**:

- `identify_expansion_regions(brain)`: Identifica regiones del espacio cerebral que deberían expandirse.
- `update_region_map!(map, position, value; radius=2)`: Actualiza una región del mapa alrededor de una posición.
- `find_high_value_regions(map, threshold)`: Encuentra regiones con valores por encima del umbral.
- `expand_region!(brain, region)`: Expande una región específica del espacio cerebral.
- `should_expand_space(brain)`: Determina si el espacio cerebral debería expandirse.
- `calculate_saturation(state)`: Calcula el nivel de saturación del estado neuronal.

architecture/CorticalLayers.jl

- **Propósito**: Implementa capas neuronales inspiradas en la corteza cerebral con estructuras de conexión específicas.
- **Funciones**:
  - `initialize_neurons!(layer)`: Inicializa las neuronas de la capa cortical.
  - `create_lateral_connections!(layer)`: Crea conexiones laterales entre neuronas de la misma capa.
  - `create_lateral_connection!(layer, source, target)`: Crea una conexión lateral entre dos neuronas.
  - `neuron_distance(neuron1, neuron2)`: Calcula la distancia euclidiana entre dos neuronas.
  - `compute_connection_probability(distance, base_probability)`: Calcula la probabilidad de conexión basada en la distancia.
  - `connect_feedforward!(source_layer, target_layer)`: Establece conexiones feed-forward entre dos capas.
  - `connect_feedback!(source_layer, target_layer)`: Establece conexiones de retroalimentación entre dos capas.
  - `compute_positional_bias(reference_neuron, neurons, dimensions)`: Calcula un sesgo posicional para conexiones basado en la posición relativa.
  - `sample_with_weights(weights, count)`: Muestrea índices con probabilidades proporcionales a los pesos.
  - `forward_pass!(layer, input_tensor)`: Realiza una pasada hacia adelante a través de la capa.
  - `process_lateral_connections!(layer)`: Procesa conexiones laterales dentro de la capa.

- `update_output_attention!(layer)`: Actualiza el mapa de atención de salida basado en el estado de activación.
- `backward_pass!(layer, output_gradient)`: Realiza una pasada hacia atrás a través de la capa.
- `extract_local_gradient(global_gradient, position)`: Extrae gradiente local para una neurona en la posición dada.
- `compute_neuron_gradient(neuron, local_gradient, plasticity_factor)`: Computa gradiente para una neurona específica.
- `compute_connection_gradient(connection, source_gradient, target_gradient, plasticity_factor)`: Computa gradiente para una conexión específica.
- `propagate_to_next_layer!(layer, input_tensor=nothing)`: Propaga activación a la siguiente capa.
- `create_cortical_column(layer_configs)`: Crea una columna cortical con múltiples capas.

architecture/HippocampalMemory.jl

- **Propósito**: Implementa un sistema de memoria inspirado en el hipocampo para almacenamiento y recuperación de patrones.
- **Funciones**:
  - `store_pattern!(memory, pattern_tensor, label=""; context=nothing, metadata=Dict{Symbol,Any}())`: Almacena un nuevo patrón en la memoria hipocampal.
  - `update_existing_pattern!(memory, pattern_id, new_tensor, new_context)`: Actualiza un patrón existente con nueva información.
  - `find_similar_pattern(memory, query_tensor)`: Busca un patrón similar al tensor de consulta.
  - `tensor_similarity(tensor1, tensor2)`: Calcula la similitud entre dos tensores.
  - `update_similarity_index!(memory, pattern)`: Actualiza el índice de similitud con un nuevo patrón.
  - `extract_pattern_features(tensor)`: Extrae características para indexación de un tensor.
  - `retrieve_pattern(memory, query_tensor; context=nothing, top_k=1)`: Recupera los patrones más similares a la consulta.
  - `update_pattern_stats!(memory, pattern_id)`: Actualiza estadísticas de un patrón tras su acceso.
  - `consolidate_memory!(memory)`: Consolida la memoria eliminando patrones menos importantes.

- `clean_similarity_index!(memory, removed_ids)`: Limpia el índice de similitud eliminando patrones.
- `set_context!(memory, context_tensor)`: Establece el contexto actual para la memoria.
- `apply_memory_decay!(memory)`: Aplica decaimiento temporal a los patrones de memoria.
- `complete_pattern(memory, partial_pattern; threshold=0.6)`: Completa un patrón parcial usando la memoria.
- `associate_patterns(memory, pattern1_id, pattern2_id)`: Crea una asociación entre dos patrones.
- `summarize_memory(memory)`: Genera un resumen estadístico de la memoria.

architecture/PrefrontalSystem.jl

- **Propósito**: Implementa un sistema de razonamiento y toma de decisiones inspirado en el córtex prefrontal.
- **Funciones**:
  - `set_goal!(system, goal_tensor)`: Establece una meta para el sistema prefrontal.
  - `clear_goals!(system)`: Limpia todas las metas activas.
  - `set_context!(system, context_tensor)`: Establece el contexto para el sistema prefrontal.
  - `reason(system, input_tensor; reasoning_type=:default)`: Aplica razonamiento al tensor de entrada.
  - `combine_inputs(system, input_tensor)`: Combina entrada con contexto, metas y estado actual.
  - `create_reasoning_engine(brain, reasoning_type)`: Crea un motor de razonamiento del tipo especificado.
  - `deliberate!(system, input_tensor; options=nothing)`: Realiza un proceso de deliberación sobre múltiples opciones.
  - `evaluate_options(system, options)`: Evalúa múltiples opciones y devuelve puntuaciones.
  - `evaluate_option(system, option)`: Evalúa una única opción basada en metas y contexto actual.
  - `tensor_similarity(tensor1, tensor2)`: Calcula la similitud coseno entre dos tensores.
  - `process_sequential!(system, input_sequence)`: Procesa una secuencia de entradas de forma temporal.
  - `reset_state!(system)`: Reinicia el estado del sistema prefrontal.
  - `get_current_state(system)`: Obtiene el estado actual del sistema prefrontal.
  - `get_confidence(system)`: Obtiene el nivel de confianza actual del sistema.

- `get_active_goals(system)`: Obtiene las metas activas actuales.
- `get_reasoning_pathways(system; limit=5)`: Obtiene las trayectorias de razonamiento recientes.

architecture/AttentionalSystem.jl

- **Propósito**: Implementa un sistema de atención adaptativo para dirigir el procesamiento del espacio cerebral.
- **Funciones**:
  - `update_attention!(system, input_tensor)`: Actualiza el estado atencional basado en un tensor de entrada.
  - `update_foci!(system)`: Actualiza los focos de atención existentes.
  - `compute_salience_map(input_tensor, inhibition_map)`: Calcula un mapa de saliencia a partir del tensor de entrada y el mapa de inhibición.
  - `normalize_tensor(tensor)`: Normaliza un tensor al rango [0,1].
  - `compute_gradient_magnitude(tensor)`: Calcula la magnitud del gradiente en cada punto del tensor.
  - `compute_local_contrast(tensor)`: Calcula el contraste local para cada punto del tensor.
  - `create_bottom_up_foci!(system, salience_map)`: Crea nuevos focos de atención bottom-up basados en el mapa de saliencia.
  - `update_inhibition_map!(system, x, y, z, radius)`: Actualiza el mapa de inhibición después de crear un nuevo foco.
  - `apply_top_down_control!(system)`: Aplica control top-down al sistema de atención.
  - `recalculate_global_map!(system)`: Recalcula el mapa de atención global basado en los focos activos.
  - `apply_focus_to_map!(attention_map, focus, dimensions)`: Aplica un foco de atención al mapa de atención.
  - `set_top_down_attention!(system, control_tensor)`: Establece el control top-down para dirigir la atención voluntariamente.
  - `create_focus!(system, position; kwargs...)`: Crea explícitamente un nuevo foco de atención en una posición específica.
  - `decay_inhibition!(system, decay_rate=0.1)`: Decae el mapa de inhibición con el tiempo.
  - `get_attention_at(system, position)`: Obtiene el valor de atención en una posición específica.

- `get_most_attended_regions(system, n=3)`: Obtiene las n regiones más atendidas.
- `apply_attention!(system, target_tensor)`: Aplica el mapa de atención actual a un tensor objetivo.
- `focus_on_region!(system, center, radius)`: Dirige la atención explícitamente a una región específica.
- `merge_attention_maps(system, other_system; weight=0.5)`: Combina el mapa de atención de este sistema con otro.
- `track_moving_stimulus!(system, position, velocity, lifetime=10.0)`: Crea un foco que sigue un estímulo en movimiento.
- `create_attentional_mask(system, threshold=0.5)`: Crea una máscara binaria basada en el mapa de atención.
- `get_attention_statistics(system)`: Obtiene estadísticas generales sobre el estado actual de atención.
- `reset!(system)`: Reinicia el sistema de atención a su estado inicial.

core/Connections.jl

- **Propósito**: Define las conexiones entre neuronas tensoriales en el espacio tridimensional.
- **Funciones**:
  - `transmit(connection, source_state)`: Transmite información desde la neurona origen a la destino a través de la conexión.
  - `update_weight!(connection, pre_activation, post_activation, learning_rate)`: Actualiza el peso de la conexión basado en actividad pre y post sináptica.
  - `should_prune(connection, threshold)`: Determina si la conexión debería ser eliminada basada en su actividad reciente.
  - `connection_distance(source_neuron, target_neuron)`: Calcula la distancia entre dos neuronas conectadas.
  - `connection_probability(source_neuron, target_neuron, max_distance, base_probability)`: Calcula la probabilidad de conexión entre dos neuronas basada en su distancia.
  - `establish_connections!(connections, neurons, config)`: Establece conexiones iniciales entre neuronas.
  - `find_connection(connections, source_id, target_id)`: Encuentra una conexión específica entre dos neuronas.

- `get_outgoing_connections(connections, neuron_id)`: Obtiene todas las conexiones salientes de una neurona.
- `get_incoming_connections(connections, neuron_id)`: Obtiene todas las conexiones entrantes a una neurona.
- `prune_connections!(connections, threshold)`: Elimina conexiones débiles basado en un umbral de actividad.

core/SpatialField.jl

- **Propósito**: Define el campo espacial que representa una región en el espacio tridimensional.
- **Funciones**:
  - `contains(field, position)`: Verifica si una posición está contenida dentro del campo.
  - `distance(field, position)`: Calcula la distancia desde una posición al centro del campo.
  - `overlap(field1, field2)`: Calcula la superposición entre dos campos espaciales.
  - `shift_field!(field, direction, amount)`: Desplaza el campo en la dirección especificada.
  - `expand_field(field, factor)`: Expande el campo por un factor dado.
  - `extract_tensor(tensor, field)`: Extrae la región del tensor correspondiente al campo.
  - `clone(field)`: Crea una copia profunda del campo.

core/TensorNeuron.jl

- **Propósito**: Define la unidad neuronal fundamental basada en tensores, que opera con tensores 3D en lugar de escalares.
- **Funciones**:
  - `process_input(neuron, input_tensor)`: Procesa un tensor de entrada y actualiza el estado interno de la neurona.
  - `update_weights!(neuron, gradient, learning_rate)`: Actualiza los pesos (kernel de transformación) de la neurona según el gradiente.
  - `adapt_receptive_field!(neuron, input_sensitivities)`: Adapta el campo receptivo de la neurona basándose en la sensibilidad a diferentes regiones de entrada.
  - `clone(neuron)`: Crea una copia profunda de la neurona para propósitos de deliberación interna.
  - `should_expand(neuron)`: Determina si esta neurona debería expandirse (generar neuronas hijas) basado en su actividad y saturación.
  - `update_functional_type!(neuron)`: Actualiza el tipo funcional de la neurona basándose en sus patrones de activación.

- `apply_hebbian_learning!(neuron, pre_synaptic, post_synaptic, learning_rate=0.01f0)`: Aplica aprendizaje Hebbiano a los pesos de la neurona.
- `update_neuronal_state(current_state, activation, plasticity)`: Actualiza el estado de la neurona combinando el estado actual con la nueva activación.
- `calculate_saturation(state)`: Calcula el nivel de saturación del estado neuronal.
- `analyze_activation_patterns(history)`: Analiza patrones de activación para determinar el tipo funcional.

core/BrainSpace.jl

- **Propósito**: Define el espacio tridimensional que contiene y organiza las neuronas tensoriales.
- **Funciones**:
  - `populate_initial_neurons!(neurons, dimensions, config)`: Puebla el espacio cerebral con neuronas iniciales.
  - `sample_positions(dimensions, n)`: Muestrea n posiciones únicas dentro de las dimensiones dadas.
  - `forward_propagation(brain, input_tensor)`: Propaga un tensor de entrada a través del espacio cerebral.
  - `propagate_layer!(brain, neuron_activations, input_tensor, layer)`: Propaga activaciones a través de una capa específica de neuronas.
  - `update_attention_map!(brain, neuron_activations)`: Actualiza el mapa de atención basado en activaciones actuales.
  - `update_attention_region!(attention_map, position, activity_level, radius=2)`: Actualiza una región del mapa de atención alrededor de una posición.
  - `update_global_state!(brain, neuron_activations)`: Actualiza el estado global del cerebro basado en activaciones de neuronas.
  - `update_global_state_region!(global_state, position, activation)`: Actualiza una región del estado global alrededor de una posición.
  - `expand_space!(brain, regions=nothing)`: Expande el espacio cerebral, añadiendo nuevas neuronas y conexiones.
  - `expand_region!(brain, region)`: Expande una región específica del espacio cerebral.
  - `establish_new_connections!(brain)`: Establece conexiones para las neuronas recién añadidas.
  - `identify_expansion_regions(brain)`: Identifica regiones del espacio cerebral que deberían expandirse.

- `update_activity_region!(activity_map, position, value, radius=2)`: Actualiza una región del mapa de actividad alrededor de una posición.
- `find_activity_clusters(activity_map, threshold=0.5f0)`: Encuentra clusters de alta actividad en el mapa.
- `find_connected_region(binary_map, visited, start_pos)`: Encuentra una región conectada en el mapa binario.
- `count_neurons_in_region(brain, region)`: Cuenta el número de neuronas en una región específica.
- `visualize_activity(brain; options...)`: Genera una visualización del estado actual del espacio cerebral.
- `prepare_input_tensor(input_tensor, dimensions)`: Prepara un tensor de entrada para que tenga las dimensiones correctas.
- `self_prune!(brain)`: Realiza auto-optimización eliminando conexiones innecesarias.
- `should_expand_space(brain)`: Determina si el espacio cerebral debería expandirse.
- `clone_brain(brain)`: Crea una copia profunda del espacio cerebral para deliberación interna.