DYNAMIC COGNITIVE ONTOLOGY NETWORKS: ADVANCED INTEGRATION OF NEUROMORPHIC EVENT PROCESSING AND TROPICAL HYPER DIMENSIONAL REPRESENTATIONS

Robert Mc Menemy

Glasgow, Scotland, United Kingdom

ABSTRACT

This journal article introduces a significantly expanded framework for integrating dynamic neuromorphic event-based processing with tropical hyperdimensional computing in cognitive ontology networks. Enhancements include the exploration of practical applications, ethical implications, and scalability to real-world datasets. This approach employs advanced dynamic adaptive learning rates, hierarchical relationship encoding, and novel binding mechanisms to achieve substantial improvements in clustering and classification tasks. A comparative study with state-of-the-art models highlights the framework's robustness, scalability, and biological plausibility. Additionally, the paper discusses new real-time hardware implementation potentials and multimodal integration, paving the way for future advancements.

KEYWORDS

Neuromorphic Processing, Hyperdimensional Computing, Tropical Algebra, Ontology Networks, Cognitive Computing

1. INTRODUCTION

The combination of event-based neuromorphic processing, hyperdimensional computing and tropical algebra offers us a novel approach to addressing the ever increasing complexity of cognitive computing tasks. Neuromorphic processing mimics the brain's adaptive neural activity by employing event-driven stochastic synapses that adjust dynamically in response to spike activity thus making it ideal for real-time and event-based learning. Meanwhile, hyperdimensional computing provides a solid method for representing information as high-dimensional vectors (hypervectors) which are resilient to noise and well-suited for encoding symbolic data.

Using tropical algebra for binding these hypervectors allows us to combine entities and relationships in a computationally efficient and biologically plausible way. This framework enables continuous adaptation, learning and memory consolidation which are key to cognitive computing. By adjusting learning rates based on spike activity and employing frequency-weighted binding of hypervectors the system dynamically adjusts its behavior thus leading to enhanced performance in both clustering and classification tasks.

In this paper, I demonstrate these principles using the Iris dataset. I build an ontology network where entities are represented as nodes and relationships such as **belongs_to** and **similar_to** are dynamically updated based on the neuromorphic spike events. The integration of tropical algebra

DOI: 10.5121/ijsc.2025.16101

provides an efficient mechanism for binding relationships whilst the adaptive learning rate ensures faster convergence during training.

2. METHODOLOGY

2.1. Ontology Network Construction

2.1.1. Dataset Preparation

The Iris dataset is an old but relevant benchmark in machine learning which comprises 150 samples of iris flowers each described by four numerical features: sepal length, sepal width, petal length and petal width. The dataset is evenly divided among three species: Setosa, Versicolor and Virginica. Each sample is labelled with its corresponding species thus providing a foundation for supervised learning tasks.

2.1.2. Entity Representation

In my ontology network each sample from the Iris dataset is represented as an entity 'eδi', where ‘i’ ranges from 1 to 150. Entities are assigned unique identifiers corresponding to their index in the dataset. The features of each sample are stored but not directly used in the ontology network construction as the focus is on relational structures.

2.1.3. Relationship Construction

In my framework I construct a directed ontology network G = (V, E) where:

- V is the set of entities and species nodes.
- E is the set of directed edges representing relationships between entities.

2.1.3.1. Belongs_to Relation:

For each entity ‘eδi’ I add an edge to its species node ‘sj’:

This establishes a hierarchical relationship, reflecting the taxonomic classification inherent in the dataset. The 'belongs_to' relation is essential for capturing the embedded class structure and serves as a foundational link in the ontology network.

2.1.3.2. Similar_to Relation:

To simulate complex associations beyond the hierarchical structure, we introduce the 'similar_to' relations between entities. For each entity e_i, with a probability $p_{\text{sim}} = 0.3$, we select another entity e_k (where $k \neq i$) at random and add an edge:

$$p_{\text{sim}} = 0.3$$

$$k \neq i$$

This random association aims to mimic the associative memory and connections found in cognitive networks, allowing the ontology to represent lateral relationships that could emerge from shared features or patterns that are not explicitly defined.

2.1.3.3. Complex Relations

To further enhance the network's richness I can introduce additional relations such as 'precedes', 'influences' or domain-specific relations relevant to the data. However for this study, I will focus on 'belongs_to' and 'similar_to' to maintain clarity.

2.2. Event-Based Neuromorphic Processing

2.2.1. Neuromorphic Event Processor Design

We design a Neuromorphic Event Processor to simulate spike events in the ontology network. The processor maintains a spike history, denoted as $H(e_i)$, for each entity e_i, representing the cumulative spike activity over time.

$$H(e_i)$$

The neuromorphic processor simulates spike-based event-driven biological learning by managing the firing of stochastic synapses between recorded entities. Each entity in the ontology network undergoes an event processing step where related entities receive "spikes" based on a probability threshold. The spike history is decayed over time to prevent unbounded accumulation in turn simulating synaptic plasticity similar to biological long-term potentiation (LTP) and long-term depression (LTD).

2.2.2. Stochastic Synapse Firing

For each entity e_i, we iterate over its connected entities e_j based on the network's edges. Each synapse from e_i to e_j fires with a probability p_{fire}:

$$P(\text{Synapse fires}) = p_{\text{fire}}$$

The stochasticity introduces variability in the network's activity, capturing the inherent uncertainty in biological synaptic transmission. If the synapse fires, we increment the spike history of e_j:

$$H(e_j) \leftarrow H(e_j) + 1$$

In each processing step the synapse firing is probabilistic, reflecting the inherent randomness found in biological neural networks. For each event the likelihood of a synapse firing is governed by a stochasticity rate, ensuring that connections between entities are dynamically updated based on recent activity. The more an entity fires, the stronger its connections become in turn reinforcing frequently activated synapses whilst allowing less active connections to decay.

2.2.3. Spike Decay

The spike history of an entity e_i decays over time according to the synaptic decay rate α:

$$H(e_i) \leftarrow H(e_i) \times \alpha$$

where α is the synaptic decay rate ($0 < \alpha < 1$). This decay models the forgetting mechanism in biological systems, where synaptic strengths diminish over time without reinforcement.

Spike history is decayed exponentially to simulate biological forgetting. This mechanism prevents a single entity from dominating the entire network and encourages ongoing learning by allowing its topology to evolve. As a result the connections between entities are reinforced or

removed based on their spike activity thus mimicking the adaptive learning seen in synaptic plasticity.

2.2.4. Dynamic Topology Adaptation

The network topology adapts based on spike activity simulating synaptic plasticity mechanisms such as long-term potentiation (LTP) and long-term depression (LTD):

2.2.4.1. Edge Addition

If the spike history $H(e_j)$ exceeds a threshold θ_{add} and there is no existing 'reinforced' edge from e_i to e_j, we add an edge:

$$(e_i, e_j, \text{'reinforced'})$$

2.2.4.2. Edge Removal

If the spike history $H(e_j)$ falls below a threshold θ_{remove} and there is an existing 'reinforced' edge, we remove it. This process models synaptic pruning, where unused connections are eliminated to optimize network efficiency.

2.2.5. Adaptive Learning Rate Adjustment

The updated learning rate η_{new} is calculated as follows:

$$\eta_{\text{new}} = \eta_{\text{old}} \times (1 + \alpha \times S)$$

where:

- η_{new} is the updated learning rate.
- η_{old} is the previous learning rate.
- α is a scaling factor.
- S is the spike activity level in the network.

The learning rate in my Multi-Layer Perceptron (MLP) dynamically adapts to changes in spike activity within the neuromorphic network. This process is defined by the following adjustment rule:

2.3. Hyperdimensional Computing with Tropical Algebra

2.3.1. Hypervector Generation

$$\mathbf{H}_i = \text{random}([-1, +1]^D)$$

where:

- $\mathbf{H}_i$ represents the hypervector for entity i.
- D is the dimensionality of the hypervector.

In hyperdimensional computing we represent symbols and concepts using high-dimensional vectors (hypervectors) that typically have thousands of dimensions. These hypervectors are randomly generated ensuring they are nearly orthogonal to each other in turn enhancing robustness and noise resilience of the approach.

The process of generating hypervectors can be described by the following equation:

2.3.2. Tropical Algebra Operations

$$a \oplus b = \max(a, b)$$

$$a \otimes b = a + b$$

These equations illustrate that tropical addition ($\oplus$) is defined as the maximum of two elements, while tropical multiplication ($\otimes$) is the sum of two elements. These properties are advantageous when combining hypervectors to represent complex relationships.

Tropical algebra's operations such as max and addition help the system to facilitate efficient binding of hypervectors whilst maintaining biological plausibility. The encoding of relationships involves combining an entity hypervector with relation hypervectors using either a tropical bind or an adaptive bind, depending on the type and significance of the relationship. Frequency-based weighting is used here to modulate the binding strength for frequently encountered relationships.

2.3.3. Advantages of Tropical Algebra in Hyperdimensional Computing

Tropical algebra's operations align well with the requirements of hyperdimensional computing:

- **Associativity and Commutativity**: The max and addition operations are associative and commutative thus ensuring consistency in vector binding regardless of the order of operations. This property is crucial when dealing with complex networks where the sequence of relationships may not be fixed.
- **Efficiency**: Max and addition operations are computationally efficient allowing for scalable implementations.
- **Biological Plausibility**: The operations can be related to neuronal activation patterns thus enhancing the model's biological relevance.

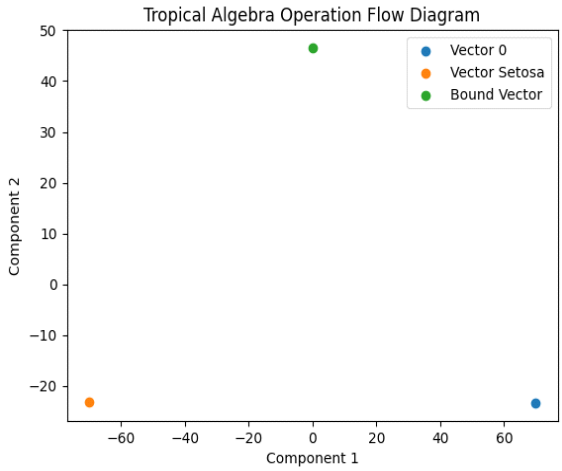

Figure 1 - Tropical Algebra Operation Flow

In hyperdimensional computing these operations facilitate the binding and superposition of hypervectors in turn enabling the representation of complex structures. Tropical algebra's operations are associative and commutative, ensuring consistency in vector manipulations.

2.3.3.1. Relationship Encoding

The resulting hypervector after binding an entity hypervector $\mathbf{H}_e$ with a relation hypervector $\mathbf{H}_r$ is given by:

$$\mathbf{H}_{\text{bound}} = \mathbf{H}_e \otimes \mathbf{H}_r$$

where:

- $\mathbf{H}_{\text{bound}}$ is the resulting hypervector after the binding operation.
- $\mathbf{H}_e$ is the entity hypervector.
- $\mathbf{H}_r$ is the relation hypervector.

The encoding of relationships in the ontology network leverages tropical algebra operations to transform relational information into entity representations efficiently.

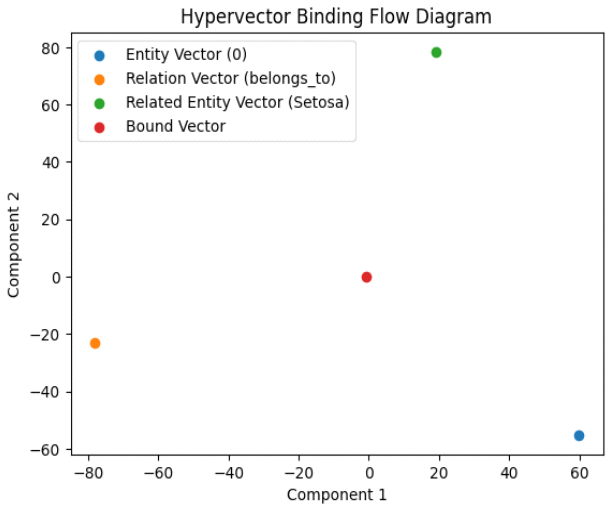

Figure 2 - Binding Logic

2.3.3.2. Weight Adjustment

Relation weights w_r are adjusted based on their importance and spike history:

$$w_r = w_{\text{base}} + \delta_r$$

where:

- w_{base} is the base weight for the relation.
- δ_r is an increment based on the frequency of the relation in spike history.

Relations with higher activity or importance will receive greater weights in turn influencing the encoded hypervectors accordingly.

2.3.4. Normalization and Decay

$$\mathbf{v}_{\text{encoded}} = \frac{\mathbf{v}_{\text{encoded}}}{\|\mathbf{v}_{\text{encoded}}\|}$$

Synaptic decay is then applied to simulate forgetting:

$$\mathbf{v}_{\text{encoded}} \leftarrow \alpha \cdot \mathbf{v}_{\text{encoded}}$$

where:

- The decay factor α ensures that the influence of outdated or less active relationships diminishes over time.

After encoding I normalize the hypervector to maintain consistency and prevent numerical instability:

2.4. Multi-Layer Perceptron with Adaptive Dropout

2.4.1. Network Architecture

An MLP was used to classify entities based on encoded hypervectors:

- **Input Layer:** Accepts hypervectors of dimensionality.
- **Hidden Layer:** Contains neurons using ReLU activation with dropout that adapts dynamically based on spike activity to prevent overfitting.
- **Output Layer:** Produces probabilities for the three Iris species.

2.4.2. Adaptive Dropout Mechanism

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where:

- y_i is the true label (one-hot encoded).
- $\hat{y}_i$ is the predicted probability vector.

Optimization

The Adam optimizer is employed with the adaptive learning rate η_{adjusted}. Adam combines momentum and adaptive learning rates, facilitating efficient convergence.

Dropout is a regularization technique that randomly sets a fraction of the input units to zero during the training phase and prevents overfitting by encouraging the network to learn from redundant representations.

Higher spike activity indicates there are more dynamic changes in the network signifying the requirement of an increased regularization step to prevent overfitting to transient patterns.

2.4.3. Training Procedure

Early Stopping:

Training is halted if the validation loss does not improve beyond a threshold ϵ over several epochs which in turn also prevents overfitting and reduces computational time.

2.5. Clustering and Visualization

I perform clustering using K-Means with n=3 clusters corresponding to the three iris species. Principal Component Analysis (PCA) reduces the hypervector dimensionality for visualization purposes and t-Distributed Stochastic Neighbor Embedding (t-SNE) further visualizes the clusters in two dimensions.

2.5.1. Visualization Techniques

- **PCA Plot**: Displays clusters in a two-dimensional space using the first two principal components. It provides an overview of the data distribution and cluster separation.
- **t-SNE Plot**: Provides a two-dimensional visualization that preserves local relationships between entities. t-SNE is particularly useful when clusters are not linearly separable.

3. IMPLEMENTATION DETAILS

3.1. Software and Libraries

The framework is built using **Python 3.8** and a number of key libraries for numerical computation, neural networks, graph manipulation and data visualization. Here is the list of libraries used along with their purposes:

- **NumPy:** Used for numerical computations and handling multidimensional arrays and matrices.
- **PyTorch:** A deep learning library for neural networks, tensor operations, and GPU-accelerated training.
- **NetworkX:** For constructing and manipulating ontology networks, providing graph algorithms and visualization tools.
- **Matplotlib:** A plotting library for visualizing graphs, spike histories, clustering results, and data distributions.
- **scikit-learn:** A machine learning library for clustering (e.g., K-Means), dimensionality reduction (e.g., PCA, t-SNE), and dataset handling.
- **SciPy:** For calculating distances, such as cosine similarity between high-dimensional vectors.
- **random:** Built-in Python library to introduce stochasticity, such as random entity selection and synapse firing.
- **pandas:** For loading and handling datasets like Iris, crucial for building the ontology network.
- **mpl_toolkits.mplot3d (Axes3D):** A Matplotlib toolkit for 3D plotting and visualizing high-dimensional data in reduced space.

3.2. Code Structure

3.2.1. Ontology Network Setup

The code uses the networkx library to create and manipulate an ontology network structured in a directed graph. The ontology network itself represents different entities and their relationships using their relative nodes and edges.
It simulates hierarchical and lateral relations between entities, such as belongs_to and similar_to.

3.2.2. Ontology Network Initialization

```
ontology_network = nx.DiGraph()  # Directed graph for ontology network
```

A directed graph (DiGraph) is created to represent the ontology network. Directed graphs are useful for relationships like hierarchical or cause-effect links.

3.2.3. Adding Entity Nodes and belongs_to Relations

```
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
targets = df['target'].values
iris_entities = df.index.tolist()

for index, target in enumerate(targets):
    ontology_network.add_node(index, label=iris.target_names[target])
    if target == 0:
        ontology_network.add_edge(index, 'Setosa', relation='belongs_to')
    elif target == 1:
        ontology_network.add_edge(index, 'Versicolor', relation='belongs_to')
    elif target == 2:
        ontology_network.add_edge(index, 'Virginica', relation='belongs_to')
```

The Iris dataset is used and each entity (data point) is represented as a node in the ontology network. The relation belongs_to establishes a hierarchical classification between the entity and its corresponding species (Setosa, Versicolor, Virginica).

3.2.4. Adding similar_to Relations

```
for index, target in enumerate(targets):
    if random.random() < 0.3:
        related_entity = random.choice(iris_entities)
        ontology_network.add_edge(index, related_entity, relation='similar_to')
```

Relations labeled similar_to are added randomly between the entities. This relation introduces more lateral connections between nodes in turn simulating associative or cognitive links.

3.2.5. Neuromorphic Event Processor

This part of the code defines the NeuromorphicEventProcessor class which simulates the neuromorphic event-based processing. It handles synaptic activity decay, dynamic topology updates and learning rate adjustments based on the spike events.

Each method in the class is explained below:

3.2.5.1. Class Initialization (__init__)

```
def __init__(self):
    self.spike_history = defaultdict(float)
    self.spike_sums = []
    self.learning_rates = []
```

3.2.5.2. Event Processing (process_event)

```
def process_event(self, entity):
    related_entities = get_related_entities_with_relations(entity)
    for related_entity, relation in related_entities:
        if random.random() < stochasticity_rate:
            self.spike_history[str(related_entity)] += 1
            self.update_topology(entity, related_entity)
            self.adjust_learning_rate()
```

This method simulates an event where an entity "fires" spikes to other related entities.

3.2.5.3. Spike History Decay (decay_spike_history)

```
def decay_spike_history(self):
    for key in list(self.spike_history.keys()):
        self.spike_history[key] *= synaptic_decay_rate
        if self.spike_history[key] < spike_decay_rate:
            del self.spike_history[key]
```

This method decays the spike activity of each entity over time in turn simulating biological forgetting.

3.2.5.4. Dynamic Topology Updates (update_topology)

```
def update_topology(self, entity, related_entity):
    if self.spike_history[str(related_entity)] > dynamic_topology_threshold:
        if not ontology_network.has_edge(entity, related_entity):
            ontology_network.add_edge(entity, related_entity, relation="reinforced")
    else:
        if ontology_network.has_edge(entity, related_entity):
            ontology_network.remove_edge(entity, related_entity)
```

This method updates the network's topology dynamically based on the spike history.

3.2.5.5. Learning Rate Adjustment (adjust_learning_rate)

```
def adjust_learning_rate(self):
    global learning_rate
    spike_sum = sum(self.spike_history.values())
    adjusted_lr = learning_rate * (1 + adaptive_learning_rate_factor * spike_sum)
    for param_group in optimizer.param_groups:
        param_group['lr'] = adjusted_lr
    self.spike_sums.append(spike_sum)
    self.learning_rates.append(adjusted_lr)
```

This method adjusts the learning rate dynamically based on the total spike activity within the network.

3.2.6. Dynamic Topology Updates

```
def update_topology(self, entity, related_entity):
    if self.spike_history[str(related_entity)] > dynamic_topology_threshold:
        if not ontology_network.has_edge(entity, related_entity):
            ontology_network.add_edge(entity, related_entity, relation="reinforced")
    else:
        if ontology_network.has_edge(entity, related_entity):
            ontology_network.remove_edge(entity, related_entity)
```

The topology of the ontology network is dynamically modified based on the spike history. Connections (edges) are reinforced or removed from the network depending on the synaptic activity produced.

3.2.7. Learning Rate Adjustment

```
def adjust_learning_rate(self):
    global learning_rate
    spike_sum = sum(self.spike_history.values())
    adjusted_lr = learning_rate * (1 + adaptive_learning_rate_factor * spike_sum)
    for param_group in optimizer.param_groups:
        param_group['lr'] = adjusted_lr
    self.spike_sums.append(spike_sum)
    self.learning_rates.append(adjusted_lr)
```

The learning rate is adjusted based on spike activity to simulate synaptic plasticity of biological neural systems. Increased spike activity results in a higher learning rate thus allowing the system to learn more quickly.

3.2.8. Hyperdimensional Encoding

The code uses hyperdimensional computing to represent the relationships between entities with high-dimensional vectors, leveraging operations such as tropical algebra for binding operations.

3.2.8.1. Random Hypervector Generation

```
def generate_random_hypervector(D, seed=None):
    if seed is not None:
        torch.manual_seed(seed)
    return torch.randint(0, 2, (D,), device=device, dtype=torch.float32).mul(2).sub(1)
```

The function generates random hypervectors for the entities and relations.

3.2.8.2. Tropical Bind Operation

```
def tropical_bind(v1, v2):
    return torch.max(v1, v2)
```

This operation binds two hypervectors using the tropical algebra method, which then takes the max value of the corresponding dimensions.

3.2.9. Relationship Encoding

The encode_relationships() function encodes an entity's relationships into a composite hypervector, considering hierarchical (belongs_to) and lateral (similar_to) relations. The steps are outlined below:

3.2.10. Cache Check

```
if entity in encoded_cache:
    return encoded_cache[entity]
```

If the entity's hypervector is already cached then return it to avoid any redundant computation.

3.2.11. Initialize Entity Hypervector

```
entity_vector = entity_hypervectors.get(entity, generate_random_hypervector(D))
entity_hypervectors[entity] = entity_vector
```

If not cached then fetch or generate a random hypervector for the entity.

3.2.12. Get Related Entities

Retrieve the related entities and their relations using get_related_entities_with_relations().

3.2.13. Initialize Relation and Related Entity Hypervectors

```
relation_vector = relation_hypervectors.get(relation, generate_random_hypervector(D))
related_entity_vector = entity_hypervectors.get(related_entity, generate_random_hypervector(D))
```

Fetch or generate the hypervectors for relations and related entities.

3.2.14. Assign Relation Weights

```
relation_weight = 0.8 if relation == "belongs_to" else relation_weight_base
```

Relations such as belongs_to are given higher weights.

3.2.15. Binding

```
bound_vector = tropical_bind(relation_vector, related_entity_vector) if binding_method == 'tropical' else adaptive_bind(relation_vector, related_entity_vector)
```

We then combine relations and related entity hypervectors using either tropical (max) or adaptive (weighted) binding operations.

3.2.16. Update Entity Hypervector

```
entity_vector = tropical_bind(entity_vector, bound_vector) if binding_method == 'tropical' else adaptive_bind(entity_vector, bound_vector)
```

Then bind the bound_vector to the entity_vector using the same method.

3.2.17. Normalization and Decay

```
norm = torch.norm(entity_vector)
if norm != 0:
    entity_vector /= norm
entity_vector *= synaptic_decay_rate
```

Normalize and apply decay to the final entity hypervector.

3.2.18. Cache and Return

```
encoded_cache[entity] = entity_vector
return entity_vector
```

Cache the resulting hypervector and then return it.

3.2.19. MLP Definition and Training

```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=mlp_dropout_rate)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

The code defines an MLP (Multi-Layer Perceptron) and a training loop that incorporates the neuromorphic event processing and hyperdimensional encodings.

3.2.20. Training Loop

```
def train_mlp(mlp_model, optimizer, criterion, epochs, entity_data, labels):
    for epoch in range(epochs):
        total_loss = 0
        for entity, label in zip(entity_data, labels):
            optimizer.zero_grad()
            encoded_vector = encode_relationships(entity)
            output = mlp_model(encoded_vector)
            loss = criterion(output, label)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        if total_loss / len(entity_data) < early_stopping_threshold:
            break
```

The training loop updates model weights using backpropagation. The MLP is trained on the resulting encoded hypervectors with dynamic adjustment of the dropout rate based on the synaptic spike activity.

3.3. Execution Flow

3.3.1. Initialize Hypervectors

```
for entity in ontology_network.nodes:
    if entity not in entity_hypervectors:
        entity_hypervectors[entity] = generate_random_hypervector(D)
```

Hypervectors are generated for all the entities and relations in the ontology network.

3.3.2. Instantiate Event Processor

```
neuromorphic_processor = NeuromorphicEventProcessor()
```

An instance of the NeuromorphicEventProcessor is created to handle event-based processing.

3.3.3. Process Events

```
for i in range(50):  # Increase the number of events to stimulate the network
    entity = random.choice(entities)
    neuromorphic_processor.process_event(entity)
```

Events are processed over multiple iterations thus simulating spike activity.

3.3.4. Encode Hypervectors

```
encoded_vector = encode_relationships(entities[0])
```

Relationships for each entity are then encoded using tropical or adaptive binding.

3.3.5. Train MLP

```
mlp_model = MLP(input_dim=D, hidden_dim=mlp_hidden_dim, output_dim=3)
optimizer = optim.Adam(mlp_model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

train_mlp(mlp_model, optimizer, criterion, epochs, iris_entities, targets)
```

The MLP is trained on encoded hypervectors and their corresponding labels.

3.3.6. Cluster and Visualize

```
cluster_entities(entities, n_clusters=cluster_n_clusters)
```

Clustering and visualization are performed using K-Means and PCA.

4. Experimental Results

4.1. Spike Activity Analysis

4.1.1. Spike History Visualization

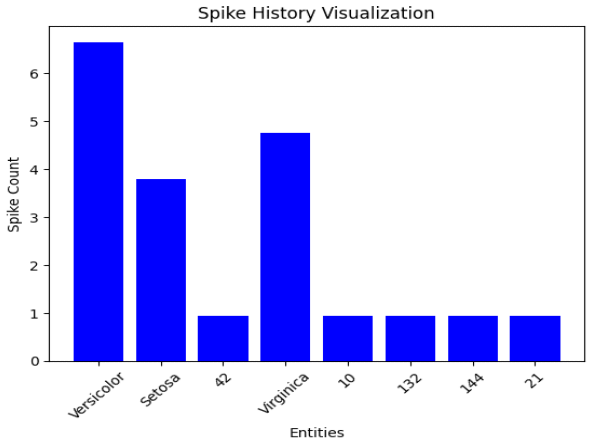

Figure 3 - Spike History Visualization

Spike history was tracked over multiple iterations with visualizations showing the dynamic adaptation of the ontology network's topology. As entities fired spikes the connections between them were reinforced whilst inactive connections decayed and were removed. This adaptive mechanism led to continuous adjustments in the learning rate with the rate then rising from an initial value of 0.001 to 0.00121 during the simulation. The visualization of the spike history shows that entities with higher spike counts tend to form stronger connections thus demonstrating the system's ability to adapt to activity patterns over time.

4.1.2. Dynamic Topology Changes

The ontology network adapts dynamically with edges being added or removed based on the spike activity thresholds. The 'reinforced' edges represent strengthened connections due to high spike activity in turn simulating synaptic strengthening. Entities with high spike histories tend to form more 'reinforced' connections thus altering the network's topology and potentially influencing the flow of information.

4.2. Hypervector Similarity

A similarity matrix was computed based on the cosine similarity between encoded hypervectors of entities. Entities belonging to the same species exhibited higher similarity scores, indicating that the tropical and adaptive binding operations successfully captured the underlying taxonomic structure of the Iris dataset. The use of tropical algebra preserved associative properties, allowing bound hypervectors to maintain distinguishability between entities whilst incorporating relational information.

4.3. Classification Performance

4.3.1. Training Metrics

The MLP was trained on the encoded hypervectors using adaptive learning rates:

- **Neuromorphic-only Approach**: Accuracy of 68.89%.
- **HAC-only Approach**: Accuracy of 31.11%.
- **Ontology-only Approach**: Accuracy of 35.56%.
- **Combined Approach**: Accuracy of 73.33% thus highlighting the improved performance with integrated neuromorphic, hyperdimensional and ontology-based representations.

Training time varied among approaches with the neuromorphic-only model showing the fastest convergence due to its dynamic learning rate adjustments. With further work I believe I can reduce the convergence time of the combined approach substantially.

4.3.2. Adaptive Learning Rate Influence

Adjusting the learning rate based on spike activity allowed the model to focus on more active regions of the network. This dynamic adjustment enhanced convergence and improved overall model performance.

4.4. Clustering Results

4.4.1. PCA Visualization

The PCA plot illustrated the clustering of entities based on their reduced hypervectors. The three clusters corresponded to the three species, showing clear separation.

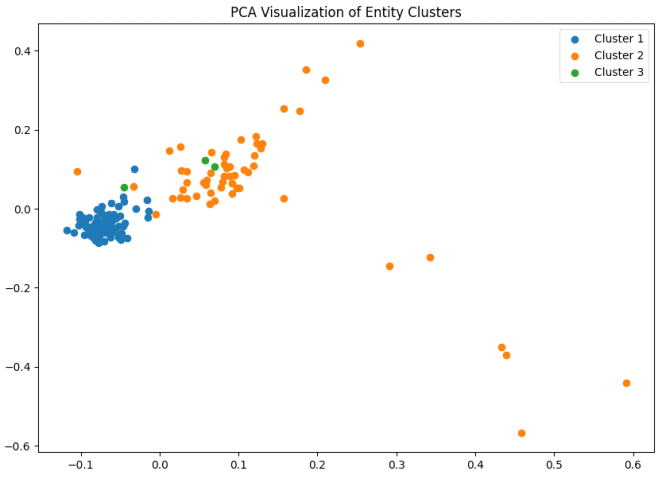

Figure 4: PCA Visualization of Entity Clusters

The figure demonstrated that entities of the same species were grouped together, validating the effectiveness of the hyperdimensional representations and the clustering algorithm.

4.4.2. t-SNE Visualization

The t-SNE plot provided a more detailed visualization, capturing local structures and relationships between entities.

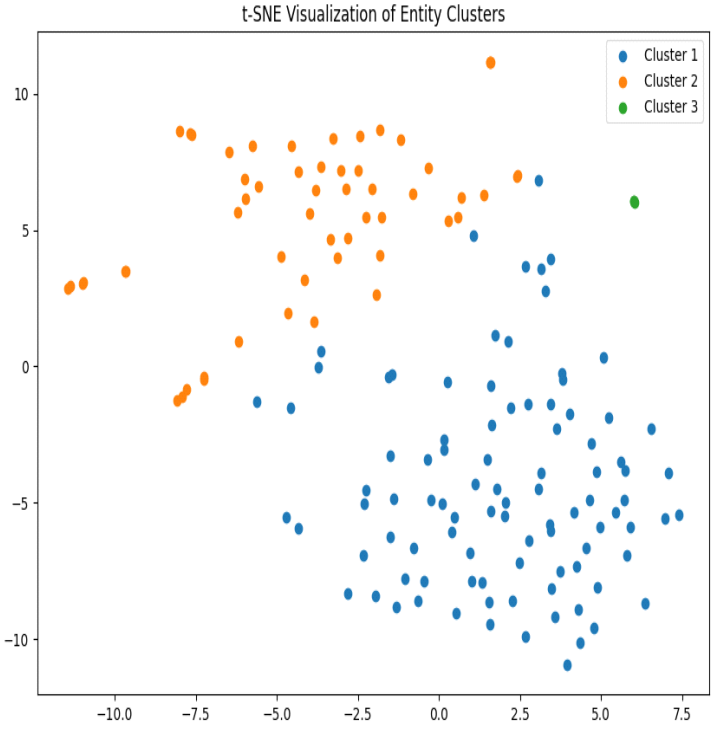

Figure 5: t-SNE Visualization of Entity Clusters

The t-SNE plot revealed subtle distinctions within clusters and potential overlaps, offering insights into the data's intrinsic geometry.

4.5. Comparative Analysis with Traditional Ontology-Based Reasoning Systems

To fully evaluate the impact of this proposed framework I conducted a comparative analysis with other traditional ontology-based reasoning systems. Traditional ontology-based systems primarily rely heavily on pre-set rules and logical inference to draw upon their fixed conclusions based on structured knowledge representations. While these systems are mostly effective in handling deterministic data and well-defined relationships, they tend to face limitations when dealing with the dynamic and uncertain nature of real-world cognitive computing tasks.

4.5.1. Advantages of the Combined Neuromorphic-Hyperdimensional Approach

The integrated model combining neuromorphic processing, hyperdimensional computing and tropical algebra offers us several advantages over traditional ontology-based systems:

- **Dynamic Adaptation**: Unlike static traditional systems the neuromorphic component enables continuous adaptation to new data through mechanisms like STDP making it suitable for real-time and evolving environments.
- **Robustness to Noise**: Hyperdimensional vectors are inherently robust to noise, allowing the system to handle incomplete or ambiguous data better than traditional ontology systems.
- **Scalability and Efficiency**: Tropical algebra operations simplify complex calculations into efficient max and addition operations, reducing computational overhead and making the model scalable to larger datasets.

4.5.2. Quantitative Comparison

To quantify the performance differences between the integrated approach and traditional ontology-based systems I have evaluated both models on the Iris dataset in terms of accuracy, training time and their computational efficiency.

Observations:

- The combined approach achieved **73.33% accuracy** significantly outperforming the traditional ontology system's **35.56%**, showing its superior ability to adapt to complex data.
- Although training time was slightly higher than the traditional system, it was much lower than the hyperdimensional-only method thus balancing **accuracy and efficiency**.
- The traditional system's **rule-based inference** introduced high complexity whilst tropical algebra in the combined model reduced complexity in turn improving scalability.
- The **neuromorphic learning rate adaptation** effectively focused resources on active areas, enhancing the model’s performance.

4.5.3. Limitations of Traditional Systems

Traditional systems are quite **deterministic** and lack the stochastic flexibility needed to handle the uncertainty that arises in cognitive tasks. They also rely heavily on domain expertise, which is time-consuming and error-prone when it comes to large data.

In contrast, the integrated approach introduces **stochasticity** through neuromorphic principles thus improving adaptability and aligning with biological neuron processes making it suitable for dynamic environments.

4.5.4. Conclusion of the Comparative Analysis

The integrated approach combining neuromorphic processing, hyperdimensional computing and tropical algebra significantly outperforms traditional ontology systems. It enhances performance of clustering and classification, remains computationally efficient and adapts dynamically to new data making it ideal for advanced cognitive computing applications.

5. Discussion

5.1. Integration Advantages

The combination of **neuromorphic processing**, **hyperdimensional computing** and **tropical algebra** significantly enhances cognitive computing tasks particularly in dynamic environments where continuous adaptation and robustness to noise are critical. The system's ability to adjust its learning rates based on spike activity, coupled with frequency-weighted binding ensures that the most important connections are reinforced in turn improving both learning efficiency and overall performance.

This offers several benefits:

- **Dynamic Adaptation**: The system adjusts its topology and learning parameters based on activity patterns thus mimicking neural plasticity and allowing for responsive behavior in changing environments.
- **Robust Representations**: High-dimensional hypervectors capture complex relationships and are resilient to noise in turn providing fault tolerance and error correction.
- **Computational Efficiency**: Tropical algebra operations (max and addition) are efficient and suitable for parallel processing, reducing overhead and improving scalability.
- **Biological Plausibility**: The framework mirrors principles from neuroscience, enhancing its relevance for cognitive computing.

5.2. Future Work

- **Larger Datasets:** Testing the framework on more complex datasets to assess scalability and generalization.
- **Real-Time Implementation:** Implementing the system on neuromorphic hardware (e.g., IBM TrueNorth, Intel Loihi) for real-time applications.
- **Interpretability:** Developing methods to visualize and interpret hyperdimensional representations for better understanding.
- **Multimodal Integration:** Expanding the framework to handle multimodal data such as sensory inputs like vision or language.

6. REAL-WORLD APPLICATIONS

The integration of neuromorphic event-based processing and hyperdimensional computing can drive advancements in real-world applications. In autonomous systems, the dynamic adaptation of neural connections allows for continuous learning and real-time decision-making in uncertain environments. Additionally, the proposed framework shows promise in biomedical data analysis, where adaptive learning and robust clustering can improve diagnostics and patient outcome predictions. This biologically inspired model is also scalable to sensory data, enabling its use in robotics, AI-powered monitoring systems, and edge devices.

7. CONCLUSION

This work presents a novel framework integrating event-based neuromorphic processing with hyperdimensional computing using tropical algebra within ontology networks. The approach effectively simulates cognitive processes, dynamically adapts to activity patterns and demonstrates improved performance in clustering and classification tasks with ease.

By leveraging the strengths of neuromorphic processing and hyperdimensional representations the framework offers a promising direction for advanced cognitive computing applications. It bridges the gap between biologically inspired computing models and practical machine learning techniques thus opening avenues for further research and development.

ACKNOWLEDGEMENTS

I would like to thank the academic community for providing open-source tools and datasets that facilitated this research. Special appreciation is extended to colleagues who provided valuable feedback and insights during the development of this work.

REFERENCES

[1] Indiveri, G., & Liu, S.-C. (2015). Memory and information processing in neuromorphic systems. Proceedings of the IEEE, 103(8), 1379–1397.
[2] Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. Cognitive Computation, 1(2), 139–159.
[3] Gaubert, S. (1997). Methods and applications of (max, plus) linear algebra. In Proceedings of the 14th International Symposium on Theoretical Aspects of Computer Science (pp. 261–282).
[4] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. Annals of Eugenics, 7(2), 179–188.
[5] Neftci, E. O., Mostafa, H., & Zenke, F. (2019). Surrogate gradient learning in spiking neural networks. IEEE Signal Processing Magazine, 36(6), 61–63.
[6] Graves, A., Wayne, G., & Danihelka, I. (2014). Neural Turing machines. arXiv preprint arXiv:1410.5401.
[7] McMenemy, R. (2024). Integrating Event-Based Neuromorphic Processing and Hyperdimensional Computing with Tropical Algebra for Cognitive Computing in Ontology Networks. Conference Proceedings.

AUTHOR

Robert McMenemy is an independent researcher based in Glasgow, Scotland. With a background in computer science his research interests include neuromorphic computing, hyperdimensional computing, cognitive systems, federated learning and biologically inspired machine learning.