intelligence factory

WHITE PAPER

Ontology-Guided Augmented Retrieval (OGAR)

by Matthew Furnari, CTO

INTRODUCTION

Bridging the Gap Between Language and Action: Buffaly's Ontology-Based Approach to Controlling LLMs

The rapid advancement of large language models (LLMs) has revolutionized the field of artificial intelligence, enabling remarkable capabilities in natural language processing and understanding. However, harnessing the power of LLMs for practical applications presents significant challenges, particularly in controlling their behavior and bridging the gap between their language-based proficiency and the ability to execute concrete actions in the real world.

Buffaly is an implementation of OGAR (Ontology-Guided Augmented Retrieval), an AI-based data retrieval technique that redefines how organizations access and analyze complex data. Unlike conventional vector databases and retrieval-augmented generation (RAG) solutions, OGAR uses an ontology-based approach to provide hallucination-free, industry-specific insights, tailored for fields like healthcare, finance, and aerospace while keeping sensitive data from being sent to LLM providers such as OpenAI, where it might be compromised or used to train new models.

This white paper examines these challenges, exploring the inherent limitations of LLMs and presenting Buffaly as a promising framework for addressing these issues through a structured ontology and a flexible scripting language called ProtoScript. Buffaly's ontology-based approach offers a transparent, controllable, and industry-specific solution that overcomes many of the shortcomings of traditional LLM implementations.

This paper aims to provide a comprehensive understanding of Buffaly's approach to controlling LLMs and integrating them with action-oriented systems. It will explore the following key concepts:

The Difficulty of Controlling LLM Behavior: This section will elaborate on the inherent challenges of managing LLM outputs, examining their unpredictable nature and the risks associated with their potential for bias, inaccuracy, and malicious manipulation.

Bridging the Gap Between LLMs and Actions: This section will focus on the disconnect between LLMs' language proficiency and their ability to execute actions in the real world. It will discuss Buffaly's framework for translating language into actions, grounding LLM outputs in real-world contexts, and enabling dynamic action planning.

Buffaly's Ontology and ProtoScript: This section will examine the role of Buffaly's structured ontology and its scripting language, ProtoScript, in controlling LLM behavior and enabling action integration. It will discuss how these components provide a transparent and controllable framework for managing LLM reasoning processes and generating executable actions.

Buffaly as an Abstraction Layer: This section will examine how Buffaly provides an abstraction layer over meaning. This layer can be used to separate LLMs interpretation from function calling. This separation gives users greater control over how their applications understand language and how they interact with data.

By examining the challenges and opportunities associated with controlling LLMs and bridging the gap between language and action, this white paper aims to demonstrate the potential of Buffaly's ontology-based approach in unlocking new possibilities for AI development and deployment.

The Need for Ontology-Based Approaches in AI

LLMs are powerful tools for natural language processing, but they have limitations that make them insufficient for complex data retrieval and AI agent tasks. Ontology-based approaches, such as those implemented in Buffaly, address these limitations by providing a structured and flexible representation of knowledge that complements the capabilities of LLMs.

Here's why ontology-based approaches are necessary:

LLMs Struggle with Data Retrieval: LLMs rely heavily on word embeddings and vector databases for information retrieval. This approach can lead to inaccuracies and limitations in handling complex queries and semantic relationships.

For example, in RAG-SQL tasks, traditional RAG methods based on word embeddings struggle to differentiate between semantically similar but distinct queries. They also have difficulty incorporating external data sources, such as real time APIs or databases.

LLMs Lack Incremental Learning: LLMs are typically batch-trained and cannot easily incorporate new knowledge without retraining the entire model. This makes them unsuitable for tasks requiring continuous learning and adaptation to new information.

Ontology-based approaches, on the other hand, support incremental learning by allowing new concepts and relationships to be added to the ontology without disrupting the existing knowledge base.

LLMs Have Limited Reasoning Abilities: While LLMs excel at pattern recognition and language generation, they struggle with complex reasoning tasks that require understanding cause and effect, hypothetical scenarios, and logical inferences.

Ontology-based approaches, especially those leveraging Buffaly's capabilities, can address these limitations by representing knowledge in a way that supports logical reasoning and inference within a specific domain. For instance, Buffaly's ontology can capture details like "the device used to work" and infer the deeper meaning, such as it no longer working, enabling more sophisticated reasoning.

Controlling LLM Behavior is Difficult: LLMs are often black boxes, making it challenging to control their behavior and ensure they act safely and reliably.

Ontology-based approaches offer a more transparent and controllable way to guide LLM actions by defining constraints and rules within the ontology. This makes it possible to create AI agents that are more predictable and aligned with human intentions.

Bridging the Gap Between LLMs and Actions: LLMs are primarily language models and lack the ability to directly interact with the real world or execute actions.

Ontology-based approaches, like Buffaly, can bridge this gap by providing a framework for representing actions and integrating them with LLM-generated insights. This enables the development of AI agents that can understand natural language instructions and translate them into meaningful actions.

By combining the strengths of LLMs with the structure and reasoning capabilities of ontology-based approaches, AI systems can overcome the limitations of LLMs and achieve greater accuracy, flexibility, and control in performing data retrieval and agent-based tasks.

Limitations of LLMs for Data Retrieval

While LLMs can process natural language effectively, they struggle with complex data retrieval due to their reliance on word embeddings and vector databases. Here's why this approach is limited and how ontology-based solutions offer a more robust alternative:

The Problem with Word Embeddings

LLMs use word embeddings, which represent words as vectors in a high-dimensional space. The idea is that words with similar meanings will have vectors close to each other in this space. This allows for semantic search, where you can retrieve information based on the meaning of a query rather than exact keyword matching.

However, word embeddings face several challenges:

- **Contextual Ambiguity:** Words can have multiple meanings depending on the context. Embedding-based methods might struggle to discern the appropriate meaning without sufficient contextual awareness.

- **Nuanced Relationships:** Capturing intricate connections between entities, such as cause and effect, temporal relationships, or hypothetical scenarios, is difficult with word embeddings alone. They primarily capture semantic similarity, not the richness of complex relationships.

- **Limited Reasoning:** Word embeddings lack the ability to reason logically about the data. They cannot infer new information or make deductions based on existing knowledge.

Challenges with Vector Databases

LLMs often use vector databases to store and retrieve large amounts of text data. These databases store the vector representations of text chunks, allowing for efficient semantic search. However, vector databases also face limitations:

- **Chunk Size:** The accuracy of retrieval often depends on the size of the text chunks used. Smaller chunks might lack sufficient context, while larger chunks can introduce noise and irrelevant information.

- **Non-Specific Dimensions:** The dimensions in the embedding space are not readily interpretable, making it difficult to understand why certain results are retrieved and others are not.

How Buffaly and Ontology-Based Solutions Address These Limitations

Buffaly, with its ontology-based approach and ProtoScript language, offers several advantages for data retrieval compared to traditional LLM and vector database methods:

Structured Knowledge Representation: Buffaly utilizes an ontology, a structured representation of knowledge that defines concepts, entities, relationships, and rules. This allows for a more precise and comprehensive understanding of the data.

Nuanced Querying: ProtoScript, Buffaly's programming language, enables the creation of complex queries that go beyond simple keyword matching. It allows for querying based on relationships, hierarchies, and logical inferences.

Incremental Learning: Buffaly's ontology can be incrementally updated with new knowledge without requiring complete retraining. This makes it adaptable to evolving data and use cases.

Transparency and Control: The ontology provides a transparent representation of the knowledge and reasoning processes, enabling greater control over the system's behavior.

Buffaly as an Abstraction Layer

Buffaly is a system that provides an abstraction layer over meaning, separating LLM interpretation from function execution. This abstraction gives users greater control over how applications understand language and interact with data, ensuring more reliable and flexible behavior.

Buffaly achieves this separation by using a meaning representation layer that is independent of any specific LLM or function-calling mechanism. Instead of relying on the LLM's interpretation alone, Buffaly utilizes a graph-based approach to represent knowledge. In this system, meaning is depicted as a network of concepts and relationships, which is used both to interpret language and to trigger actions.

This approach is notably more flexible and powerful than traditional natural language understanding methods that often rely on rigid, hand-crafted rules. Buffaly's graph-based model allows it to adapt to new information more easily. For instance, when encountering a new word, Buffaly can simply add it to its conceptual graph, enhancing its understanding and response capability without the need for extensive reprogramming.

How this Works in Practice:

Consider an application that allows users to query a database using natural language. Without Buffaly, the application might rely solely on an LLM to interpret the query and generate a corresponding SQL command. However, the LLM might not always generate a correct or efficient query due to its inherent limitations in precise function execution.

With Buffaly's abstraction layer, the meaning of the user's query is represented independently of the LLM's interpretation. Buffaly's graph-based reasoning engine then takes over, ensuring the generation of a more accurate and efficient SQL query. Furthermore, because the meaning is represented explicitly, the underlying implementation can be changed without impacting the rest of the system. For example, if the original query implementation was based on SQL, it could easily be replaced with an API call without disrupting the overall structure. Buffaly ensures that the meaning remains intact, regardless of how the action is executed.

Key Benefits of this Approach:

- **Increased Accuracy:** By separating LLM interpretation from function execution, Buffaly ensures more precise language understanding and query generation.

- **Increased Efficiency:** Buffaly's graph-based reasoning generates more optimized SQL queries compared to those created by LLMs alone.

- **Increased Flexibility:** The abstraction layer allows for greater adaptability, enabling the application to support new domains or query types without being tightly bound to the limitations of the LLM. Furthermore, changes to the underlying implementations, such as switching from SQL to API calls, can be made seamlessly without affecting how the system understands the meaning.

Buffaly can serve as a bridge between LLMs and code execution, combining the broad natural language capabilities of LLMs with the precision and control of traditional programming languages. For example, Buffaly can translate natural language instructions into function calls or map abstract concepts in language to concrete data structures. This duality allows developers to harness the strengths of both LLMs and Buffaly's graph-based reasoning for more controlled and efficient applications.

Buffaly provides a vital abstraction layer that separates LLM interpretation from function execution. This distinction gives users more control over how applications process language and interact with data, making the system more transparent and accurate. In contrast to the sometimes unreliable outputs of LLMs, which can be factually incorrect or offensive, Buffaly's controlled approach ensures a higher level of precision and reliability. Moreover, because meaning is explicitly represented, developers have the flexibility to change the underlying implementation—such as replacing a SQL query with an API call—without affecting the rest of the system.

Real-Time Information Retrieval with Buffaly

Buffaly excels at retrieving real-time information from external sources and integrating it into workflows for further execution by an LLM. One of its key advantages over using LLMs alone is that Buffaly is fully programmable. This programmability allows users to add new capabilities or modify existing ones, enabling them to tailor the system to their specific application needs. Buffaly also offers a layer of control and safety when building AI agents capable of real-time processing.

A prime example of this is **RAG-SQL**, which leverages a ontology metabase to write SQL queries, integrated with SQL workbench tools. When a user provides a directive in natural language, Buffaly searches a semantic database for similar directives. If a match is found, Buffaly suggests one of the associated queries. If no match is found, it uses word embeddings to identify the most similar directives and presents them as options. If a completely new query is required, Buffaly builds a prompt based on similar directives and associated queries, which is then sent to an LLM to generate a new SQL query. This approach improves both the accuracy and efficiency of query generation by separating meaning from execution and using Buffaly's reasoning engine.

Another example is **SemDB and CRM Integration**, where Buffaly integrates its Semantic Database (SemDB) with a CRM system to automatically extract and update customer information from interactions like phone calls and emails. When a call ends or an email is received, the system sends the audio or text to SemDB. Using a combination of LLMs and data extractors, Buffaly analyzes the interaction, extracting relevant details such as names, phone numbers, and email addresses. This information is then used to update the CRM, adding missing details like an email address to a lead.

In the case of **Real-Time Transcription and Analysis**, Buffaly works with a transcription service to process live audio streams from phone calls. A WebSocket streams the audio data to a transcription service, which returns the transcript to Buffaly. Buffaly then uses data extractors to analyze the transcript, identifying key information such as the speaker's name, phone number, or email address. This data is used to update the CRM or trigger other actions, all in real time.

Buffaly's capabilities for real-time information retrieval are further enhanced by the following techniques:

- **Semantic Search:** Buffaly uses semantic search, often powered by word embeddings, to find related data or queries. This enables it to retrieve relevant information from its ontology or database, even when the input isn't an exact match.

- **Few-Shot Learning:** Buffaly can utilize few-shot learning by storing pairs of directives and queries in its semantic database. When a new query comes in, Buffaly performs a semantic search to retrieve relevant pairs and presents them as examples to the LLM. This allows the LLM to generate appropriate responses based on a minimal number of examples.

- **Data Extractors:** Buffaly employs data extractors to pull specific information from unstructured data, such as transcripts or emails. These extractors can be programmed to recognize entities, identify patterns, or answer specific semantic questions. The extracted data is then used to update databases, trigger actions, or provide context to LLMs.

This level of flexibility makes Buffaly particularly effective in handling complex real-time scenarios, especially where traditional Retrieval-Augmented Generation (RAG) systems might struggle. Its programmable nature allows it to be adapted to various domains and use cases, making it a versatile and powerful tool for real-time applications.

Overcoming the Incremental Learning Limitations of LLMs with Buffaly and Ontology-Based Solutions

LLMs face a significant hurdle when it comes to incremental learning. Once trained, they struggle to incorporate new knowledge without a computationally expensive and time-consuming retraining process. This limitation stems from their architecture and training methodology, which contrasts sharply with the dynamic and adaptive nature of human learning.

Challenges with Incremental Learning in LLMs

Traditional LLMs are typically trained on massive static datasets. **Adding new information necessitates retraining the entire model on the combined old and new data.** This process is:

- **Resource Intensive:** Retraining demands significant computational power, time, and access to large datasets.
- **Prone to Catastrophic Forgetting: I**ntroducing new information can disrupt existing knowledge, causing the model to “forget” previously learned patterns.

- **Impractical for Dynamic Environments:** In real-world scenarios where data constantly evolves, frequent retraining is not feasible.

Solutions with Buffaly and Ontology-Based Approaches

Buffaly, employing an ontology-based approach and leveraging the power of ProtoScript, **offers a more flexible and adaptable solution to incremental learning:**

Structured Knowledge Representation: Buffaly’s ontology represents knowledge in a structured and modular manner, enabling the addition of new concepts and relationships without requiring a complete model overhaul. This modular structure facilitates the isolation of new information, minimizing the risk of disrupting existing knowledge.

ProtoScript for Dynamic Updates: ProtoScript, Buffaly’s programming language, provides mechanisms for dynamically updating the ontology with new information. This allows for incremental learning without the need for complete retraining, enabling the system to adapt to evolving data and changing environments.

Targeted Learning: Buffaly’s architecture allows for focused learning within specific sub-graphs of the ontology. This targeted approach further reduces the impact of new information on unrelated parts of the knowledge base, promoting more efficient and stable learning.

Learning from Few Examples: Buffaly leverages its graph structure to identify and utilize covarying properties between inputs and outputs, allowing it to learn new concepts and relationships from limited examples. This contrasts with LLMs, which typically require vast amounts of data for effective learning.

LLMs Have Limited Reasoning Abilities

While LLMs excel at tasks involving language generation, they often struggle with complex reasoning and logical deduction, especially when faced with real-world situations requiring more than pattern recognition. This limitation arises because they rely on statistical correlations learned from large datasets, rather than employing explicit knowledge representation and reasoning mechanisms.

LLMs' Struggle with Reasoning

Lack of Explicit Knowledge: LLMs store knowledge implicitly in their network parameters, which makes it difficult to represent and reason about specific facts and relationships in a clear, structured way. This implicit nature hinders their ability to perform logical inference or handle symbolic manipulation effectively.

Buffaly's Ontology-Based Reasoning

Buffaly, utilizing a structured ontology and the expressive power of ProtoScript, offers a more robust and controllable approach to reasoning in AI systems. The ontology allows for the explicit representation of knowledge, enabling the system to reason about facts, relationships, and rules in a more logical and transparent manner.

- **Explicit Knowledge Representation:** Buffaly employs an ontology to define concepts, entities, and relationships in a clear and structured way. This explicit representation allows for logical reasoning based on defined rules and constraints, moving beyond statistical correlations to more symbolic and deductive reasoning.

- **Graph-Based Inference:** ProtoScript enables the creation of rules and constraints within the ontology. These rules can be used to infer new information or make deductions based on existing knowledge, enhancing the system's reasoning capabilities.

- **Handling Hypothetical Scenarios:** The ontology's structure allows for the representation of hypothetical situations and potential outcomes. By manipulating entities and relationships within the ontology, Buffaly can explore the consequences of different actions or scenarios, facilitating a form of "what-if" analysis and reasoning about possibilities.

- **Understanding User Intent:** In natural language understanding, an ontology representing user intentions and goals can be used to interpret ambiguous requests or resolve pronoun references based on the context of the conversation. ProtoScript can define rules for disambiguation and intention recognition, enabling more accurate and relevant responses.

ProtoScript Enables Flexible Reasoning

ProtoScript plays a crucial role in enabling Buffaly's reasoning capabilities:

Defining Rules and Constraints: ProtoScript allows users to define the rules and constraints that govern the system's reasoning processes. This enables the encoding of domain-specific knowledge and expert heuristics, enhancing the accuracy and relevance of the system's deductions.

Manipulating Entities and Relationships: ProtoScript provides mechanisms for creating, updating, and querying entities and relationships within the ontology. This allows for the dynamic exploration of different scenarios and the reasoning about potential consequences of actions or changes within the knowledge base.

Integrating External Data Sources: ProtoScript facilitates the integration of external data sources into the ontology, enriching the knowledge base and expanding the scope of the system's reasoning capabilities.

Buffaly's approach, combining a structured ontology with the power of ProtoScript, offers a more robust and adaptable alternative to LLMs for tasks requiring reasoning and logical deduction. It provides a framework for building AI systems that are not just good at pattern recognition but also capable of understanding and reasoning about the world in a more nuanced and human-like way.

Real World Examples

Real-World Example: In practical applications, Buffaly addresses the limitations inherent in word embedding-based models, particularly with semantic clustering and inference. Consider two similar user problem statements: "Receiving Too Many Test Kits" and "Receiving Test Kits." Using standard word embeddings, these two phrases would cluster closely because they share many common terms. However, in practice, the meaning behind these phrases is quite different. "Receiving Too Many Test Kits" implies an issue (over-supply), while "Receiving Test Kits" is neutral and lacks the context of an issue.

In a more accurate understanding, "Receiving Too Many Test Kits" should be more semantically aligned with a query like “Cancel Test Kits,” as both relate to addressing an issue with the supply of kits. This is where Buffaly’s ontology-based approach excels. Buffaly’s ability to make inferences at a semantic level allows it to align these concepts more precisely. Even though the words in “Receiving Too Many Test Kits” and “Receiving Test Kits” are similar, Buffaly can distinguish between their meanings based on context.

Additionally, words like “no” and “more” can drastically change the meaning of similar phrases. For example, “wants more test kits” and “wants no test kits” are vastly different in meaning, yet word embeddings might group them closely because they share common terms like “wants” and “test kits.” In this case, Buffaly can account for the nuanced differences in meaning by representing these phrases in a more structured and contextual way, which word embedding approaches alone cannot achieve.

Diagramming the Scenario:

1. Standard Word Embedding-Based Clustering:
    - *Cluster A:* [“Receiving Too Many Test Kits”, “Receiving Test Kits”] (Word embeddings see these as semantically similar due to shared terms like “Receiving” and “Test Kits”)
2. Contextual Misalignment (Word Embedding Limitation):
    - Even though “Receiving Too Many Test Kits” implies an issue and “Receiving Test Kits” does not, word embeddings treat these as close concepts without accounting for the deeper meaning (i.e., “too many” indicates a problem).
    - Similarly, phrases like “wants more test kits” and “wants no test kits” would be treated similarly by word embeddings, despite their clear difference in intent.
3. Buffaly’s Inference-Based Approach:
    - *New Alignment:* Buffaly leverages explicit knowledge and contextual rules from its ontology to infer that “Receiving Too Many Test Kits” implies corrective action, which can be inferred as “Cancel Test Kits.” This moves “Receiving Too Many Test Kits” away from “Receiving Test Kits” and closer to actions like “Cancel Test Kits” in the conceptual space.
    - *Cluster B:* [“Receiving Too Many Test Kits”, “Cancel Test Kits”] (Buffaly’s inference places these together based on the understanding that an excess of test kits logically leads to cancellation).
    - Additionally, Buffaly would distinguish between “wants more test kits” and “wants no test kits” based on its understanding of the words “more” and “no,” ensuring that their distinct meanings are captured.

This example highlights the limitations of using word embeddings alone to understand meaning, as they often overlook contextual differences between phrases. Buffaly's ontology-based approach, which combines semantic inference with word embeddings, provides a more nuanced and robust way to understand language and make logical deductions. By representing meaning in a structured, context-aware framework, Buffaly offers a more sophisticated alternative to traditional word embedding-based models for natural language understanding.

Real-World Example: When answering a question like "Does the device turn on?" different responses can convey vastly different meanings despite similar wording. Possible answers include "It does [turn on]," "It used to turn on," "It is turning on," "It had been turning on," and "It has been turning on." While these responses share common elements such as "turn on," their actual meanings differ significantly, especially when considering the temporal aspects and the current functionality of the device.

Word Embedding Limitation:

In word embedding-based models, phrases like "*It has been turning on*" and "*It had been turning on*" are likely to be clustered closely together because they share similar wording, particularly the verbs "*has*" and "*had*," and the phrase "*turning on*." However, their meanings are quite different: "*It has been turning on*" implies recent or ongoing functionality, while "*It had been turning on*" refers to a past state that no longer holds. Word embeddings treat these phrases similarly, which leads to incorrect conclusions.

Additionally, word embeddings would likely treat "*It is turning on*" and "*It used to turn on*" as part of the same cluster since both involve the concept of the device's functionality. However, "*It is turning on*" suggests a current state of activation, while "*It used to turn on*" implies the device no longer works. The nuances of these differences are lost when word embeddings group them together based on shared terms.

Lastly, "*It does [turn on]*" would often be placed in a separate cluster because it lacks the explicit "*turning on*" phrase, even though it still serves as an implicit confirmation of functionality. This separation does not align with the actual meaning, as "*It does [turn on]*" should be treated as an affirmative response akin to "*It is turning on*" or "It has been turning on."

Buffaly's Inference-Based Approach:
Buffaly, leveraging its ontology-based reasoning, is able to differentiate these responses more accurately. By explicitly recognizing temporal and contextual cues, Buffaly can place these responses into clusters that more accurately reflect their intended meanings.

Diagramming the Scenario:

1. **Word Embedding-Based Clustering (Incorrect Alignment):**
    - *Cluster A:* ["It has been turning on", "It had been turning on"]
      (Word embeddings treat these similarly because of shared phrases like "turning on," even though one implies recent functionality and the other refers to a past state that no longer holds.)
    - *Cluster B:* ["It is turning on", "It used to turn on"]
      (Word embeddings place these together based on the shared concept of the device's activation, but their meanings differ dramatically — one implies current functionality and the other implies that the device no longer works.)
    - *Cluster C:* ["It does [turn on]"]
      (Because this phrase lacks the explicit "turning on" phrase, word embeddings would often treat it as a separate response, even though it is still an affirmative answer to the question.)
2. **Buffaly's Inference-Based Clustering (Correct Alignment):**
   Affirmative/"Yes"-Equivalent Cluster:
   Buffaly recognizes that responses like "*It is turning on,*" "*It has been turning on,*" and "*It does [turn on]*" are all affirmative in nature and indicate that the device is either currently functioning or has been functioning recently:
    - *Cluster A:* ["It is turning on", "It has been turning on", "It does [turn on]"]
      (Buffaly groups these based on their shared meaning that the device is functional, either now or in the recent past, despite differences in phrasing.)
3. **Negative/"No"-Equivalent Cluster:**
   Responses like "It used to turn on" and "It had been turning on" are both closer to a "no" answer, as they imply that the device is no longer functional:
    - *Cluster B:* ["It used to turn on", "It had been turning on"]
      (Buffaly correctly groups these together, understanding that both indicate the device is not operational anymore, even though the word embeddings might miss the subtle difference between "has" and "had.")

Semantic Interpretation:

- **Affirmative/“Yes”-Equivalent Responses:**
    - “*It is turning on*” — The device is currently in the process of activating.
    - “*It has been turning on*” — The device has recently been turning on, suggesting it is likely functional.
    - “*It does [turn on]*” — Though lacking the explicit “turning on” phrase, this is still an affirmative answer implying that the device works.
- These responses all provide a clear “yes” answer to the question “Does the device turn on?” and should be clustered together.
- **Negative/“No”-Equivalent Responses:**
    - “*It used to turn on*” — The device worked in the past but no longer functions.
    - “*It had been turning on*” — The device was operational at some point in the past, but this is no longer the case.
- These responses are closer to “no,” as they indicate that the device is no longer turning on.

Conclusion:

While word embeddings cluster these responses based on shared terms like “turning on” or similar verb forms (“has,” “had”), Buffaly understands the contextual and temporal differences between the phrases. Buffaly correctly distinguishes between affirmative answers (“It is turning on,” “It has been turning on,” “It does [turn on]”) and negative answers (“It used to turn on,” “It had been turning on”), clustering them based on their actual meanings rather than surface-level similarities. This highlights the power of ontology-based reasoning in accurately understanding language nuances, which word embeddings alone cannot achieve.

Challenges in Controlling LLM Behavior and Buffaly's Approach

Controlling the behavior of LLMs is a major challenge in AI development. Their reliance on statistical patterns learned from massive datasets makes it difficult to predict and direct their output in a precise and reliable manner. This lack of control can lead to undesirable outcomes, such as generating biased or inappropriate content or failing to align with user intentions. Buffaly's ontology-based approach, using ProtoScript, provides a more controlled and transparent framework for managing LLM behavior, making it possible to audit, refine, and establish control over the LLM's outputs.

Difficulties in Controlling LLMs:

- Black-Box Nature: LLMs operate as complex black boxes, making it hard to understand the reasoning behind their outputs or correct biases. This opacity complicates efforts to control behavior or ensure alignment with ethical standards.

- Sensitivity to Input Phrasing: LLMs can produce drastically different responses based on subtle variations in input phrasing. This unpredictability makes it challenging to consistently achieve desired outputs.

- Susceptibility to Prompt Injection Attacks: Carefully crafted input prompts can manipulate LLMs, potentially leading to harmful content generation, underscoring the need for strong control mechanisms.

- Limited Ability to Incorporate External Knowledge: LLMs often struggle to integrate external knowledge, making them less adaptable to new domains or tasks that require specialized information.

Buffaly's Ontology-Based Control

Buffaly offers a structured and programmable solution to address these issues. Through an explicit knowledge representation, Buffaly's ontology defines concepts, relationships, and rules transparently, making it easier to audit and refine LLM behavior. This explicit framework provides developers with control over the LLM's outputs, ensuring they adhere to desired guidelines and reduce the risk of undesirable content generation.

By serving as an **abstraction layer**, Buffaly separates LLM interpretation from function execution, making it easier to audit the LLM's decision-making processes, guide reasoning paths, and refine behavior based on external constraints. The use of **ProtoScript** allows developers to define clear constraints and reasoning rules, ensuring that outputs remain within acceptable bounds while also integrating external knowledge sources to refine decision-making.

ProtoScript's Role in Controlling LLM Behavior:

- **Defining Output Constraints:** ProtoScript allows developers to establish rules that ensure LLM outputs are precise and aligned with task-specific constraints, preventing irrelevant or harmful content generation.

- **Guiding Reasoning Paths:** Developers can influence the LLM's decision-making process by using ProtoScript to create rules that guide reasoning and ensure consideration of relevant information.

Buffaly as an Interface Between LLMs and Code

Buffaly acts as an interface between LLMs and code, leveraging LLMs for natural language understanding while maintaining precise control over execution. This interface ensures that LLM outputs align with specific tasks and goals, allowing developers to refine and control the system's behavior in a reliable, transparent way.

By combining Buffaly's structured ontology with ProtoScript, AI systems can audit, control, and refine LLM behavior more effectively, leading to more predictable and aligned outcomes. Buffaly provides the tools to mitigate the risks inherent in LLMs, offering a robust framework for real-world applications.

Bridging the Gap Between LLMs and Actions with Buffaly

Large language models (LLMs) excel at language-based tasks, but their ability to interact with the real world and take concrete actions remains limited. This disconnect arises from their focus on language processing rather than understanding and executing actions within a dynamic environment. Buffaly aims to bridge this gap by providing a framework for integrating LLMs with action-oriented systems, leveraging the strengths of both approaches to create more powerful and practical AI applications.

LLMs' Limitations in Action Execution

- **Abstract Nature of Language:** LLMs primarily operate within the realm of language, manipulating words and symbols without a direct connection to the physical world or the ability to perform actions within it. This abstraction limits their applicability in scenarios that require concrete interactions, such as controlling a robot or interacting with a software application.

- **Lack of Grounding in the Real World:** LLMs typically lack a grounded understanding of the real world and the consequences of actions within it. They might generate instructions or plans that are logically sound but practically infeasible or even dangerous if executed without considering real-world constraints and limitations.

- **Inability to Handle Dynamic Environments:** LLMs struggle to adapt to dynamic environments where conditions change rapidly and require real-time adjustments to plans and actions. Their static knowledge base and limited ability to process real-time sensory information hinder their effectiveness in such scenarios.

Buffaly's Approach to Action Integration

Buffaly addresses these limitations by providing a framework for connecting LLMs to action execution systems, allowing them to:

- **Translate Language into Actions:** Buffaly utilizes its ontology and ProtoScript to map natural language instructions or plans generated by LLMs into concrete actions that can be executed by external systems or agents. This translation process involves resolving ambiguities, grounding concepts in the real world, and generating executable commands or scripts.

- **Ground LLM Outputs in the Real World:** Buffaly grounds LLM outputs in the real world by integrating with databases, APIs, and other systems that provide real-time information about the environment. This grounding allows the system to validate the feasibility of LLM-generated plans, identify potential conflicts or risks, and adjust actions accordingly.

- **Enable Dynamic Action Planning:** Buffaly supports dynamic action planning by incorporating feedback from the environment and adjusting plans in real-time based on changing conditions. This adaptability enables the system to handle unforeseen events and to execute actions effectively even in complex and unpredictable environments.

ProtoScript's Role in Action Integration

ProtoScript plays a crucial role in bridging the gap between LLMs and actions by:

- **Defining Actionable Concepts:** ProtoScript allows developers to define concepts within the ontology that represent actionable entities and processes in the real world. This includes specifying the parameters, preconditions, and effects of actions, providing a structured representation that LLMs can use to reason about and generate action plans.

- **Creating Action Execution Mechanisms:** ProtoScript enables the creation of procedures and functions that translate high-level action plans generated by LLMs into executable commands for specific systems or agents. This includes handling the details of communication protocols, data formats, and error handling, ensuring reliable and robust action execution

By providing a framework for connecting LLMs to action execution systems, Buffaly allows for the creation of AI applications that can not only understand language but also act upon that understanding in a meaningful and effective way. This integration unlocks new possibilities for AI to impact the real world and solve practical problems in various domains.

Buffaly's Implementation of Ontology Guided Augmented Retrieval

Prototypes: Abstraction, Isolation, and Meaning Representation

Abstraction and Isolation

Prototypes, ProtoScript, and the ontology provide a level of abstraction that separates LLMs from direct control over specific functions. Buffaly uses a graph-based approach where knowledge is represented in the form of prototypes, which are programmatic structures embodying concepts and relationships. ProtoScript, a C#-based programming language, simplifies the creation and manipulation of these graph structures.

This abstraction allows LLMs to focus on natural language processing and semantic understanding, while the ontology handles the execution of specific actions. For example, when a user requests "show all users created today," the LLM translates this into a semantic representation. The ontology then utilizes predefined transforms to generate the appropriate SQL query: "select * from Users where DateCreated > dateadd(day, -1, getdate())."

The ontology isolates the LLM from SQL syntax and database interactions, allowing it to operate without direct control over the called functions. This separation enhances modularity and flexibility, making it easy to incorporate new functionalities without modifying the LLM itself.

Representing and Understanding Meaning

The ontology and prototypes serve as a comprehensive interface and mechanism for representing and understanding meaning. The ontology stores knowledge like a grammar or state machine, representing:

- **Facts:** Barack Obama was President.
- **Code:** C#, HTML, ProtoScript.
- **Relationships:** Between different entities and concepts.

Prototypes offer a universal way to represent and manipulate information within these graphs. This enables Buffaly to handle various operations, allowing the system to reason about knowledge in a standardized way.

The process of representing meaning involves mapping lexemes (strings) to sememes (units of meaning). For example, the lexeme “buffalo” can map to different sememes: BuffaloCity, BuffaloAnimal, or BuffaloAction. Prototypes implement these sememes and their relationships, providing structured knowledge storage.

The ontology enhances Buffaly’s ability to capture nuanced relationships. In a CRM scenario, it can represent relationships between leads, accounts, and interactions. For example, it can answer “how many patient calls did Natalia make yesterday?” by translating it into precise SQL queries. The ontology also handles timeframes, hypothetical scenarios, and cause-and-effect relationships for deeper language understanding.

Prototype Data Structure

Here are some key points about **Prototypes**:

- **Purpose:** Prototypes were developed to provide a more efficient and adaptable data structure for AI tasks, surpassing traditional formats like JSON. The design prioritizes:
    - **Scalability:** By using integers as the core of the Prototype structure, they offer improved performance and can efficiently handle large datasets.
    - **Flexibility:** Prototypes are highly versatile and can represent a wide variety of data types, ranging from simple properties and collections to more complex relationships. They are adaptable to the specific needs of AI projects, supporting a wide range of data types including code, natural language, and even abstract ideas.
- **Serialization:** Prototypes are designed to be easily serialized to and from databases or JSON, making them highly practical for storage and data transfer. This flexibility allows seamless integration with existing data infrastructures and makes it easier to exchange data between systems. Their ability to transform into JSON or other database-friendly formats adds to their scalability and usability in real-world applications.
- **Basic Structure:** Prototypes have a straightforward yet powerful structure:
    - **Name:** Each Prototype is identified by a name, functioning similarly to a class or object in traditional programming, and is represented as a string.
    - **ID:** A unique integer ID is assigned to each Prototype, enabling fast internal processing and efficient referencing across large datasets.
    - **Properties and Collections:** Prototypes contain properties, akin to fields in a class, and can manage collections, which are essentially lists of other Prototypes. This allows Prototypes to represent complex, hierarchical data structures with ease.

- **Versatility in Data Representation:** Prototypes can represent various types of data, from structured elements like code to unstructured content such as natural language and conceptual ideas. This adaptability makes them suitable for a wide range of AI applications, from language models to knowledge representation systems. Prototypes go beyond simple data containers, functioning as flexible building blocks that can encapsulate diverse forms of information.
- **Tokenization and Granularity:** Similar to the way tokenization works in LLMs, Prototypes share commonalities with how language is broken down into tokens. However, they offer more flexibility in descending natively to sub-word or multi-word tokens. This means Prototypes can represent granular pieces of information—down to individual sub-word components or extended multi-word phrases—allowing for more refined and context-aware data processing.
- **Contextual Awareness:** Due to their flexible structure and integration with systems like Buffaly, Prototypes can evolve dynamically as they receive more context. This capability, combined with **type mutability**, enables them to change their structure and meaning based on incoming data or updated relationships. As a result, Prototypes support a more contextual and evolving understanding of data.

Relationships and Type Mutability

Prototypes can express relationships through inheritance ("is-a" relationships) and property references. They support **multiple inheritance**, and **type mutability** allows prototypes to dynamically change type during runtime based on context. For instance, "Buffalo" can initially be a generic prototype, later classified as City, Animal, or Action depending on context.

Representing Meaning

Prototypes implement sememes and map them to lexemes. For example, "buffalo" could represent BuffaloCity, BuffaloAnimal, or BuffaloAction. This structure allows Buffaly to disambiguate meanings in context.

Extending Functionality

ProtoScript allows prototypes to have member functions, extending their capabilities. These functions can handle:

- **Categorization:** Classifying prototypes into specific categories.
- **Transformation:** Modifying prototype structure or properties.

- **Other Operations:** Supporting complex reasoning and interactions.

Example Scenario

In a CRM system, the ontology could have prototypes for “Lead,” “Account,” and “Interaction.” These prototypes can represent properties like “Name,” “Company,” and “Status,” allowing Buffaly to answer complex queries such as “show all leads from healthcare companies with pending appointments.” The ontology processes this query through graph operations, retrieving the relevant prototypes based on their properties.

Analyzing Type Mutability in Buffaly's Ontology

Type mutability in Buffaly aligns with the principles of a **sense enumeration lexicon** rather than thin semantics. Here's why:

- **Explicit Representation of Senses:** Buffaly explicitly represents multiple senses for words, such as "BuffaloCity," "BuffaloAnimal," and "BuffaloAction."
- **Contextual Disambiguation:** Buffaly determines the appropriate meaning of a word based on context.
- **Dynamic Type Assignment:** Prototypes dynamically change type as more information becomes available.

Distinguishing from Thin Semantics

Buffaly's system is not confined to a fixed set of categories, supporting dynamic type assignment as needed. This flexibility goes beyond thin semantics, aligning more closely with sense enumeration.

Buffaly's type mutability approach facilitates a nuanced understanding of language, handling polysemy and ambiguity effectively by dynamically adjusting the type of a prototype based on context. This provides a richer semantic representation and deeper language comprehension.

Polysemy, Homonymy, and Prototypes in Buffaly

Buffaly, a prototype-based ontology system, addresses the challenges of homonymy (words with multiple unrelated meanings) by creating separate prototypes for each distinct sememe. For polysemy (words with multiple related meanings), Buffaly uses more flexible constructs, such as multiple inheritance, properties, and transformations, to capture the nuanced meanings that arise in different contexts while maintaining the underlying relationships between them.

Distinct Prototypes for Homonyms

For homonymous words, Buffaly represents each meaning with a unique prototype (or sememe). This ensures clear distinctions between unrelated meanings. For example, the word “buffalo” could refer to an animal, a city, or an action, and each of these distinct meanings is mapped to a separate prototype in ProtoScript, Buffaly’s programming language:

protoscript

```
[Lexeme.SingularPlural(“buffalo”, “buffaloes”)]
prototype BuffaloAnimal : Animal;

[Lexeme.Singular(“buffalo”)]
prototype BuffaloCity : City;

[Lexeme.Singular(“buffalo”)]
prototype BuffaloAction : Action;
```

This approach provides an explicit separation of homonymous meanings, treating each as a distinct entity within the ontology. Each meaning is represented by its own prototype, ensuring no confusion between unrelated senses.

Handling Polysemy with Multiple Inheritance, Properties, and Transformations

For polysemous words, Buffaly uses constructs like **multiple inheritance**, **properties**, and **transformations** rather than creating separate prototypes for each meaning. Polysemous words often have meanings that are related, and Buffaly captures these relationships through shared properties or behaviors.

For example, the word “bank” (as in a financial institution) and “bank” (as in the side of a river) share commonalities but differ in specific attributes. In Buffaly, both meanings might share a common prototype but diverge through different properties or transformations. Instead of creating separate prototypes for each sense, Buffaly uses inheritance to define the core shared attributes and then differentiates the specific contexts using additional properties or behaviors.

```
prototype RiverBank : Location {
}

prototype FinancialBank : Location {
}
```

This allows Buffaly to maintain the underlying relationship between related meanings while providing enough flexibility to handle their distinct contexts.

Contextual Disambiguation and Type Mutability

Buffaly’s system supports **type mutability**, allowing prototypes to change their type based on contextual clues. This dynamic adjustment is crucial for resolving polysemy and homonymy in real-time language processing. For instance, the system might initially identify “bank” in a sentence, but as more information becomes available, Buffaly can dynamically assign it the correct type—”RiverBank” or “FinancialBank”—depending on the context.

The system’s deterministic tagger uses heuristics, training data, and context to select the most appropriate interpretation of a word. This dynamic type assignment ensures that Buffaly accurately captures the meaning as the surrounding language evolves.

Advantages of Buffaly’s Approach

Buffaly’s strategy for handling polysemy and homonymy offers several key advantages:

- **Explicit Representation for Homonyms:** Buffaly clearly separates unrelated meanings of homonyms by creating distinct prototypes for each sememe, reducing ambiguity.

- **Flexible Representation for Polysemes:** Instead of rigidly splitting related meanings, Buffaly uses multiple inheritance and properties to capture the subtle distinctions between related meanings, ensuring flexibility in understanding context-specific nuances.

- **Dynamic Meaning Assignment:** Through type mutability, Buffaly can adjust meaning dynamically as more contextual information is processed, improving accuracy in real-world scenarios.

Integration with LLMs

Combining Buffaly with Large Language Models (LLMs) can address some of its limitations. LLMs excel at understanding language complexity and can assist Buffaly in generating new prototypes or refining existing ones based on new data. This integration can enhance Buffaly's ability to handle complex language processing tasks, particularly with polysemous and homonymous words.

Enhancing Graphs with Functions in ProtoScript

ProtoScript, a C#-based programming language designed for building ontologies and AI systems, extends the concept of graphs beyond static data representation by embedding functions directly within the graph structure. This innovative approach allows ProtoScript to offer enhanced expressiveness, flexibility, and learning capabilities for graph-based systems.

Functions as Integral Graph Components

In ProtoScript, functions are treated as first-class entities within the graph, allowing them to be directly associated with specific prototypes. Unlike traditional graph databases that rely on external query languages or separate processing engines for functional operations, ProtoScript embeds functions within the graph itself. This seamless integration enables more complex interactions within the graph and allows for dynamic behaviors to be encoded alongside the data.

Advantages of Functional Integration

- **Increased Expressiveness:** By embedding functions, ProtoScript can represent and manipulate more intricate relationships and operations. For example, a prototype representing a “SalesLead” can include a function that calculates the “Lead Score” based on attributes such as engagement or recent activity. This function is stored directly within the graph, ensuring that both data and logic are interconnected, leading to richer semantic modeling.

- **Enhanced Flexibility:** ProtoScript allows functions to be triggered dynamically based on specific conditions or events. This enables the graph to respond to changes in data or user input in real time, creating more adaptive and context-aware systems. For example, functions can adjust a prototype’s properties or relationships based on new information, making the system highly responsive to evolving data.

- **Simplified Learning and Manipulation:** The integration of functions simplifies the process of learning from and transforming graph structures. Functions in ProtoScript can be used for tasks such as graph transformation, learning graph abstractions, or performing advanced operations like graph completion.

This tight coupling between graph data and functional logic makes graph manipulation more intuitive and powerful.

Examples of Functional Integration

Several examples illustrate how functions enhance ProtoScript's graph structures:

- **Type Mutability:** ProtoScript enables dynamic type changes for prototypes at runtime using functions. For instance, an untyped "Buffalo" prototype can be assigned a specific type such as "BuffaloAnimal" by invoking a function which adjusts the type based on contextual information.

- **Dynamic Property Addition:** Functions allow new properties to be added to prototypes without altering their original definition. This enables flexible, evolving data structures that can adapt to new requirements or data over time.

- **Subtype Definition with Functions:** ProtoScript uses functions to define subtypes based on specific criteria. For example, a subtype like "CityInNewYork" can be dynamically defined for cities whose location property equals "NewYorkState" using a function that performs categorization or filtering based on property values.

ProtoScript's integration of functions into graph structures marks a significant leap in graph-based knowledge representation. By embedding functions directly into the graph, ProtoScript transforms graphs from static data models into dynamic, interactive systems. This allows for more expressive and flexible AI applications, enabling systems to learn, adapt, and reason in ways that static data alone cannot support. Functions within the graph allow for real-time adaptation, making ProtoScript a powerful tool for building intelligent, responsive AI systems.

Buffaly's Language Model: A Hybrid Approach

Buffaly's language model is a hybrid system that merges elements of traditional symbolic AI, such as **prototypes and deterministic tagging**, with the advanced capabilities of modern Large Language Models (LLMs). This approach leverages the strengths of both paradigms while addressing their individual limitations, offering a powerful and flexible solution for complex AI tasks.

Key Components

Several key components define Buffaly's language model:

- **Prototypes:** Prototypes are data structures in ProtoScript (a C#-based language) that represent concepts and their relationships. They provide an explicit way to represent semantic information. For example, the word "lead" could have different prototypes representing its various meanings, such as "Lead (a person)" or "Lead (a material)."

- **Lexemes and Sememes:** A lexeme refers to a basic lexical unit like a word or phrase, while a sememe represents its underlying meaning. Buffaly maps lexemes to sememes using ProtoScript annotations. For instance, "buffalo" can map to different sememes like "BuffaloAnimal," "BuffaloCity," and "BuffaloAction," each representing a distinct meaning.

- **Sequences:** Buffaly defines sequences of prototypes using grammatical rules and semantic relationships. These sequences, similar to Chomsky grammars, guide the interpretation of natural language input. For example, Buffaly could recognize sequences like "City Animal" or "Animal Action" to understand language structure.

- **Deterministic Tagger:** Buffaly's deterministic tagger is a rule-based system that parses unstructured text into a graph structure. It applies heuristics and past training data to identify lexemes, hypothesize sememes, and match sequences, creating a semantic representation of the input.

- **LLM Integration:** Buffaly enhances its capabilities by integrating LLMs like ChatGPT or Gemini, which assist in disambiguating complex language and providing deeper contextual understanding. The LLMs aid the deterministic tagger by analyzing text, identifying entities, and refining interpretation probabilities.

How It Works

Buffaly's language model operates through tokenization, hypothesis generation, sequence matching, and graph transformation:

1. **Tokenization:** The input text is broken down into lexemes (words or phrases).
2. **Hypothesize a Sememe:** The deterministic tagger proposes possible sememes (meanings) for each lexeme using heuristics.
3. **Sequence Matching:** The tagger searches for matching sequences of prototypes based on grammatical and semantic expectations.
4. **Collapsing to Interpretation:** When a sequence matches, the lexemes collapse into sememes, confirming their interpretation.
5. **Graph Transformation:** As Buffaly processes sequences, it constructs a graph that represents the meaning of the input text and the relationships between concepts.

This iterative process builds a semantic graph, which serves as a structured representation of the input.

Going Beyond Traditional Approaches

Buffaly's language model surpasses traditional methods of part-of-speech tagging and basic semantic networks through:

- **Functions within Graphs:** ProtoScript allows Buffaly to embed functions directly into graph structures. These functions enable dynamic calculations, context-aware behaviors, and complex relationship modeling, enriching the system's expressiveness.

- **LLM Integration:** LLMs improve Buffaly's ability to resolve ambiguity and handle complex language, providing more context and depth to the interpretation process.

- **Domain-Specificity:** Buffaly's architecture is tailored for specific domains, enabling it to offer more efficient and accurate language understanding for targeted use cases.

Strengths and Potential

Buffaly's hybrid approach offers notable strengths:

- **Controllability and Transparency:** By explicitly representing knowledge through prototypes and rules, Buffaly provides greater transparency and control than typical black-box LLMs.

- **Efficiency and Scalability:** Buffaly's deterministic tagging and domain-focused design enable more efficient processing, making it scalable for large datasets.

- **Explainability:** The structured nature of prototypes allows for clear explanations of the system's decision-making process, improving trust and interpretability.

- **Incremental Learning:** Buffaly has the potential for incremental learning through adjustments to its prototypes and rules, enhancing its adaptability over time.

Buffaly and LLMs: A Symbiotic Relationship

Buffaly's Standalone Capabilities and Weaknesses

Buffaly excels in representing and manipulating knowledge in a graph-based format using prototypes, which are defined programmatically in ProtoScript. These prototypes can represent various types of information, such as:

- **Facts:** For instance, Buffaly can store simple assertions like "Barack Obama was President."
- **Code:** Buffaly is capable of representing and manipulating code structures, showing potential for code generation tasks.
- **Relationships:** Buffaly captures complex relationships between entities, such as those found in CRM systems, by modeling interactions between leads, accounts, and data points.

Weaknesses:

- **Parsing Limitations:** Buffaly relies on predefined rules and structures for parsing language, struggling with complex sentences, ambiguous phrases, and nuanced language, which may require manual intervention.
- **Scalability Challenges:** As the complexity of Buffaly's ontology increases, managing and processing large graphs can become computationally expensive.

LLMs: Strengths and Limitations

LLMs excel at understanding complex language and processing vast datasets, with strengths such as:

- **Broad Learning:** LLMs are trained on massive datasets, allowing them to understand a wide range of concepts and handle complex sentence structures effectively.
- **Generalization:** LLMs can generalize across different domains and adapt to a variety of tasks, even those they weren't explicitly trained for.

Weaknesses:

- **Lack of Transparency:** LLMs function as black boxes, making it difficult to understand how they reach certain conclusions, which poses challenges for debugging and reliability.

- **Hallucinations and Bias:** LLMs are prone to generating incorrect information (hallucinations) and may reflect biases in their training data, which impacts their trustworthiness.

- **Limited Actionability:** LLMs are excellent at language generation but cannot directly execute tasks or interact with external systems without integration.

Synergistic Integration: Buffaly + LLMs

The combination of Buffaly and LLMs forms a symbiotic relationship that maximizes the strengths of both systems:

- **Enhanced Parsing and Understanding:** LLMs can handle the complexity and ambiguity of language, simplifying Buffaly's parsing tasks and reducing its reliance on manual intervention.

- **Improved Learning and Adaptability:** LLMs can help Buffaly identify areas where its ontology needs improvement and can assist in generating or refining prototypes based on new data.

- **Actionability and Control:** Buffaly's graph structure translates LLM-generated insights into executable actions, making the system capable of interacting with external systems and performing tasks.

Examples of Synergistic Interactions

- **RAG-SQL (Retrieval Augmented Generation for SQL):** LLMs help Buffaly interpret user queries and convert them into structured representations, which Buffaly then uses to generate precise SQL queries.

- **Automated CRM Agent:** Buffaly uses LLMs to understand customer interactions, triggering appropriate actions like appointment scheduling or customer follow-ups within the CRM system.

- **SemDB (Semantic Database):** LLMs analyze and categorize unstructured text data, enriching Buffaly's semantic representations and improving search and analysis capabilities.

By integrating Buffaly's structured knowledge representation with LLMs' advanced language processing, this hybrid system offers a powerful, adaptable, and transparent solution for AI tasks. This collaboration between Buffaly and LLMs paves the way for AI systems that can better understand, interact with, and impact the world around them.

Buffaly Compared to Other Approaches

Buffaly vs. Knowledge Graphs: A Comparative Analysis

While both Buffaly and knowledge graphs utilize graph structures to represent and process information, they differ significantly in their capabilities and approaches.

Knowledge graphs primarily focus on representing factual knowledge in a structured format, typically using triples to connect entities and their relationships. For example, a knowledge graph might contain facts like "Thomas is a train" or "Thomas has wheels," representing these relationships in a graph structure.

Buffaly, on the other hand, aims to move beyond simple fact representation and into the realm of learning, reasoning, and action execution. It uses a more flexible graph-based data structure and a scripting language called ProtoScript to represent not just factual knowledge, but also procedural knowledge and more complex relationships, such as hypothetical scenarios, cause and effect, and temporal relationships. This enables Buffaly to capture a broader range of information and to reason about those relationships in a more sophisticated way.

Furthermore, **Buffaly's learning capabilities distinguish it from traditional knowledge graphs.** Buffaly has the ability to learn from few examples, to learn incrementally, and to adapt to new information without the need for massive datasets or extensive retraining. This flexibility in learning makes Buffaly more suitable for dynamic environments and tasks that require adaptability and continuous learning.

Here's a more detailed breakdown of the key differences:

Knowledge Graphs:

- **Focus:** Representing factual knowledge.
- **Structure:** Typically use triples (subject-predicate-object) to represent relationships.
- **Learning:** Limited learning capabilities; often rely on manual curation or rule-based systems.
- **Reasoning:** Basic reasoning based on defined relationships and inference rules.
- **Action Execution:** Generally not designed for action execution; focus on knowledge representation and retrieval.

Buffaly:

- **Focus:** Learning, reasoning, and action execution, in addition to knowledge representation.
- **Structure:** Uses a more flexible graph-based data structure with various node types and relationships, represented in ProtoScript.
- **Learning:** Capable of learning from few examples, incremental learning, and adaptation to new information.
- **Reasoning:** More sophisticated reasoning capabilities, including handling hypothetical scenarios, cause and effect, and temporal relationships.
- **Action Execution:** Designed for action execution by translating LLM-generated plans into executable actions.

In essence, **while knowledge graphs excel at storing and retrieving factual information, Buffaly takes a more holistic approach, aiming to bridge the gap between language, knowledge, and action.** By leveraging its flexible graph structure, learning capabilities, and action execution framework, Buffaly seeks to enable the development of more intelligent and adaptable AI systems that can effectively interact with and reason about the real world.

Buffaly Compared to Abstract Meaning Representation and Semantic Networks

Buffaly, Abstract Meaning Representation (AMR), and semantic networks all represent knowledge using graph structures, yet they differ in their objectives, levels of abstraction, and methods of learning and reasoning.

Buffaly

Buffaly focuses on connecting language comprehension with action execution. This system aims to translate the output of Large Language Models (LLMs) into real-world actions using a combination of graph-based data structures, a scripting language called ProtoScript, and learning algorithms. Buffaly prioritizes transparency and controllability, offering a more interpretable and manageable approach compared to the black-box nature of neural networks.

Abstract Meaning Representation

Abstract Meaning Representation (AMR) is a graph-based formalism for representing the semantic structure of sentences, focusing on capturing the core meaning of a sentence in a way that is independent of specific languages or syntactic variations. **AMR graphs typically use nodes to represent concepts and edges to represent semantic roles or relationships between those concepts.**

Some parallels can be drawn between Buffaly and AMR:

- **Graph-Based Representation:** Both Buffaly and AMR employ graph structures to represent knowledge, emphasizing the relationships between concepts.
- **Semantic Focus:** Both systems strive to capture the meaning behind language, going beyond surface-level syntax.

However, **key differences exist in their scope and objectives**:

- **Action Execution:** Buffaly explicitly aims to bridge the gap between language and action, while AMR primarily focuses on semantic representation.
- **Learning and Reasoning:** Buffaly has learning capabilities, whereas AMR typically relies on manual annotation or rule-based systems for graph construction.

Semantic Networks

Semantic networks represent knowledge as a network of interconnected concepts, with links representing relationships between those concepts. These networks can vary in their complexity and levels of abstraction, ranging from simple hierarchical structures to more intricate graphs with multiple types of nodes and relationships.

Buffaly shares some commonalities with semantic networks:

- **Graph-Based Knowledge Representation:** Both systems use graphs to structure and organize knowledge, emphasizing the connections between concepts.
- **Semantic Relationships:** Both approaches represent relationships between concepts, enabling reasoning and inference based on those connections.

However, **Buffaly distinguishes itself from traditional semantic networks in several ways:**

- **Focus on Action:** Buffaly's primary objective is to enable action execution, while semantic networks typically focus on knowledge representation and reasoning.
- **Learning Capabilities:** Buffaly incorporates learning algorithms to adapt its knowledge base, while many semantic networks rely on manual curation or rule-based systems.

Integration with LLMs: Buffaly is specifically designed to interface with LLMs, translating their output into actionable commands, a feature not typically found in traditional semantic networks.

Rooted Graphs, Leaf-Based Transforms, and Connections to Masked Language Modeling and Graph Theory in Buffaly

Buffaly is a graph-based learning system that utilizes rooted graphs and leaf-based transforms, integrating concepts from masked language modeling and graph theory to build more adaptive AI systems.

Rooted Graphs in Buffaly

Buffaly uses **rooted graphs** as a core structure for representing information and facilitating learning. In a rooted graph, there is a designated root node, often representing the primary subject or concept being analyzed. From this root, the graph branches out to capture the relationships and properties associated with the concept.

Rooted graphs are structured differently depending on the task. For example, when handling sequences, the root might represent the source sequence, with child nodes encoding elements or transformations applied to that sequence. In language processing tasks, the root could represent a sentence, while the child nodes represent words, phrases, or their semantic interpretations.

This use of rooted graphs provides several key advantages:

- **Reduced Search Space:** By focusing on a designated root, the system can limit its search, enabling more efficient processing and analysis.
- **Structure for Transformation:** The hierarchical nature of rooted graphs provides a clear framework for applying transforms, capturing new information or relationships as the graph evolves.
- **Representation of Context:** Rooted graphs effectively encapsulate contextual information relevant to a concept or sequence, aiding in more sophisticated reasoning and understanding.

Leaf-Based Transforms

Leaf-based transforms are essential to Buffaly's ability to generalize from examples and enable incremental learning. These transforms operate specifically on the leaf nodes of a graph, which represent the most detailed elements of the data.

Leaf-based transforms leverage the concept of “shadows,” generated by comparing two graphs to identify similarities and differences. By focusing on shared patterns or structures at the leaf level, Buffaly can extract reusable components or sub-graphs, allowing the system to:

- **Transfer Learn:** Leaf-based matching allows Buffaly to transfer knowledge between examples. For instance, if it learns to translate “show me all patients” into a specific SQL query, it can apply this knowledge to a similar request like “show me all leads.”

- **Escape the Curse of Dimensionality:** Identifying and reusing common sub-graphs allows Buffaly to generalize from fewer examples, reducing the number of explicit examples required for training.

- **Mimic Masked Language Modeling:** This process of identifying and utilizing common sub-graphs mirrors masked language modeling, where systems learn to predict missing words based on surrounding context.

Relationship to Masked Language Modeling

Masked language modeling, a technique widely used in large language models (LLMs), shares conceptual similarities with Buffaly’s use of leaf-based transforms. Both methods involve:

- **Pattern Recognition:** Identifying recurring structures or sequences in the data.
- **Context-Based Prediction:** Using surrounding information to predict missing or hidden elements.
- **Generalization:** Applying learned patterns to new data or scenarios.

While masked language modeling typically works with linear sequences of tokens, Buffaly applies these principles to graph structures, potentially offering advantages in representing complex relationships and enhancing explainability.

Connections to Broader Graph Theory

Buffaly’s use of graphs and graph-based operations aligns with broader trends in AI development, particularly in:

- **Geometric Deep Learning:** Extending deep learning techniques to non-Euclidean data, such as graphs and manifolds.

- **Graph Neural Networks:** Neural networks designed to work with graph-structured data, performing tasks such as node classification, link prediction, and graph generation.

- **Knowledge Representation and Reasoning:** Graphs provide a flexible and natural framework for representing knowledge and implementing reasoning algorithms.

By explicitly representing knowledge through graphs, Buffaly promotes greater control, transparency, and explainability than models that rely solely on implicit representations within neural networks.

Buffaly's use of rooted graphs and leaf-based transforms demonstrates the potential for graph-based learning systems to address key challenges in AI. By combining these techniques with explicit knowledge representation and LLM integration, Buffaly provides a promising foundation for building AI systems that are more controllable, transparent, and efficient.

Conclusion: The Promise and Potential of Buffaly in Shaping the Future of AI

This white paper has explored the critical challenge of bridging the gap between the language proficiency of large language models (LLMs) and their ability to take concrete actions in the real world. While LLMs exhibit remarkable capabilities in understanding and generating human-like text, their abstract nature and lack of grounding in real-world contexts hinder their practical application in domains that require action execution and dynamic decision-making. **Buffaly** emerges as a potential solution to this challenge, providing a framework for controlling LLM behavior and seamlessly integrating them with action-oriented systems.

Buffaly's innovative framework enables LLMs to interact effectively with action-oriented systems by grounding language in real-world contexts, ensuring that generated actions are feasible, safe, and adaptable. Through its integration with real-time information sources, Buffaly enhances LLM capabilities, allowing them to make dynamic decisions and execute actions that align with user goals and environmental constraints. Furthermore, Buffaly's focus on transparency, control, and real-time adaptability sets it apart from traditional AI approaches, providing developers with the tools to refine, audit, and govern LLM outputs in a reliable way.

Concrete examples of Buffaly's potential in real-world applications include **automated business processes, RAG-SQL,** and **RAG-CRM**. These examples demonstrate the versatility of Buffaly's approach in enabling LLMs to interact with and control various systems, extending their capabilities beyond language processing into the realm of practical action execution.

Furthermore, Buffaly's focus on **transparency and controllability** sets it apart from traditional black-box AI systems. By providing a structured ontology and a human-readable scripting language, Buffaly allows developers to understand and manage the reasoning processes of LLMs, ensuring that their actions align with intended goals and ethical considerations. This emphasis on transparency and control is crucial in fostering trust and accountability in AI systems, addressing concerns about the potential risks associated with uncontrolled LLM behavior.

The development of Buffaly represents a significant step forward in realizing the full potential of LLMs for practical applications. By bridging the gap between language and action, Buffaly paves the way for a new generation of AI systems that can not only understand and generate human-like text but also interact with the real world in a meaningful and impactful way. As research and development in this field continue, Buffaly holds the promise of transforming industries, streamlining processes, and ultimately enhancing human capabilities through the power of controlled and actionable AI.