Module: entity.jl

This module is dedicated to supporting a new structure called Entity, designed to encapsulate metadata using HllSets. This initiative builds on the innovative work by Mike Saint-Antoine of SimpleGrad.jl, who adapted Andrey Karpathy's foundational MicroGrad project. While the original frameworks by Mike S.-A. and Andrey K. utilize Numbers as the fundamental components of their neural networks, our approach replaces Numbers with HllSets.

It is often argued that Numbers in conventional neural network models are derived by transforming "real" things through numerical embedding. This method is effective for those elements of reality. However, we aim to shift our focus from data to metadata, thereby describing these elements more abstractly. A significant advantage of using metadata is its inherent ability to categorically separate elements into semantically similar groups. These groups are not strictly distinct; the sets of entities described by different metadata often overlap.

The most crucial aspect for us is that each piece of metadata correlates with a specific set of elements. In the realm of metadata, an HllSet serves as an embedding for a collection of these elements. This embedding is represented not as a numerical value but as a fixed-size bit-vector, specifically a 2-dimensional Tensor (64, P). In Julia, this is expressed as Vector{BitVector}, where each vector has a fixed length of 64 bits and the number of these bit-sets is determined by P. The parameter P defines the precision of the HyperLogLog approximation of the collection of entities.

HllSets offer a versatile means of comparing various instances through a broad array of metrics, effectively achieving the primary goal of embedding. However, HllSets deliver significantly more benefits. As evidenced in numerous instances, HllSets serve as a reliable substitute for original datasets and fully support all set operations.

Entity: formal introduction

Let us begin with a more formal definition of an Entity.

```
Python
mutable struct Entity{P}
       sha1::String
       hll::HllSets.HllSet{P}
       grad::Float64
       op::Union{Operation{FuncType, ArgTypes}, Nothing} where {FuncType,
       ArgTypes}
       # Constructor with keyword arguments
       function Entity{P}(hll::HllSets.HllSet{P}; grad=0.0, op=nothing) where
       {P}
           sha1 = string(HllSets.id(hll))
```

```
            new{P}(sha1, hll, grad, op)
        end
end
```

Any object instantiated from this struct will be an instance of Entity. When comparing this definition to the struct proposed by Mike S.-A., we observe several modifications: a few elements have been added, and one has been omitted. Specifically, the 'data' element has been substituted with 'HllSet'. In essence, we introduce only one novel element, which is 'sha1', serving as a SHA-1 based identifier for the Entity instance.

```
Python
mutable struct Value{opType} <: Number
    data::Float64
    grad::Float64
    op::opType
end
```

With the introduction of the Entity, we can now formally implement the concepts of **Static** and **Dynamic** structures. The Entity enables us to distinctly separate these structures by assigning them different sets of operations tailored to their specific functionalities.

Static Structure operations

HllSet embodies the core essence of the Entity and remains immutable within any given instance of an Entity context. Static operations facilitate the creation of new Entity instances through various operations, yet they do not alter the existing instances of an Entity.

In contrast, operations within the Dynamic Structure are designed to support modifications to an Entity instance. However, these modifications adhere to the overarching principle of immutability, which will be discussed further in the subsequent section.

Currently, we have identified several operations associated with the Static Structure:

1. **Negation**: This unary operation generates a new Entity instance by reversing the sign of the 'grad' field in the provided instance.
2. **Copy**: creates a new Entity instance that is a copy of the Entity argument.
3. **Union**: A binary operation that merges the HllSets of two Entity instances, resulting in a new Entity with a combined HllSet.
4. **Intersection**: Similar to the union operation, this creates a new Entity by performing an intersection on the HllSets of the provided Entity instances.
5. **XOR** (Exclusive OR): This operation also merges HllSets from two Entity instances but uses an exclusive OR approach.

6. **Complement** (Comp): This operation produces a new Entity instance containing elements from the HllSet of the first argument that are not present in the HllSet of the second argument.

The following paragraphs present the source code for all operations mentioned. Each operation is accompanied by a corresponding 'backprop' function, which propagates changes in the 'grad' field of the resulting Entity instance to the Entity instances of the input arguments. It is important to note that the unary operations, Negation and Copy, do not have associated backprop functions.

Negation:

```
Python
function negation(a::Entity{P}) where {P}
       return Entity{P}(HllSets.copy!(a.hll); grad=-a.grad, op=a.op)
end
```

Copy:

```
Python
function copy(a::Entity{P}) where {P}
       return Entity{P}(HllSets.copy!(a.hll); grad=a.grad, op=a.op)
end
```

Union:

```
Python
function union(a::Entity{P}, b::Entity{P}) where {P}
       hll_result = HllSets.union(a.hll, b.hll)
       op_result = Operation(union, (a, b))
       return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

We are providing the source code for the 'backprop' function specifically for the union operation. We will not be including it for other operations as they are nearly identical at present. We have chosen to keep them this way due to planned future enhancements that will likely differentiate them.

```
Python
function backprop!(entity::Entity{P},
```

```
        entity_op::Union{Operation{FuncType, ArgTypes}, Nothing}=entity.op)
        where {P, FuncType<:typeof(union), ArgTypes}

        if (entity.op != nothing) && (entity.op === entity_op) && (entity_op.op
        === union)
            entity_op.args[1].grad += entity.grad
            entity_op.args[2].grad += entity.grad
        else
            println("Error: Operation not supported for terminal node")
        end
end
```

Intersection:

```
Python
function intersect(a::Entity{P}, b::Entity{P}) where {P}
        hll_result = HllSets.intersect(a.hll, b.hll)
        op_result = Operation(intersect, (a, b))
        return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

XOR:

```
Python
function xor(a::Entity{P}, b::Entity{P}) where {P}
        hll_result = HllSets.set_xor(a.hll, b.hll)
        op_result = Operation(xor, (a, b))
        return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

Complement:

```
Python
# comp - complement returns the elements that are in the set 'a' but not in the
'b'
function comp(a::Entity{P}, b::Entity{P}; opType=comp) where {P}
        hll_result = HllSets.set_comp(a.hll, b.hll)
        op_result = Operation(comp, (a, b))
        # comp_grad = HllSets.count(hll_result)
```

```
    return Entity{P}(hll_result; grad=0.0, op=op_result)
    end
```

Dynamic Structure operations

This section outlines our operational implementation of the John von Neumann Self-Reproducing Automata concept. The implemented operations are designed to facilitate the creation of self-reproducing Neural Networks (NN), which will serve as the foundation for the Self-Generative System (SGS).

The primary operation responsible for the reproduction of NN elements (nodes) will be termed the **'advance'** operation. However, before we can implement this operation, it is essential to establish several supporting operations. These supportive operations will enable us to track and manage changes within the nodes of the NN and the relationships among them.

Changes in the HllSet (or simply "set") can be characterized by three distinct subsets that collectively define the overall set. More generally, these subsets can also describe the differences between two arbitrary sets (HllSets):

1. The HllSet representing elements **added** to the new HllSet compared to the old one.
2. The HllSet representing elements **retained** in the new HllSet that were inherited from the old one.
3. The HllSet representing elements that have been **deleted** in the new HllSet compared to the old one.

It is important to note that the operations discussed are based on the **complement (comp)** operation outlined in the previous section.

The operations responsible for managing changes in HllSets also oversee the grad field within the Entity. **This distinction highlights a significant difference between Static and Dynamic Structure operations.**

Added:

Python
```
function added(current::Entity{P}, previous::Entity{P}) where {P}
       length(previous.hll.counts) == length(current.hll.counts) ||
       throw(ArgumentError("HllSet{P} must have same size"))
       hll_result = HllSets.set_comp(previous.hll, current.hll)
       op_result = Operation(added, (current, previous))
       comp_grad = HllSets.count(hll_result)
       return Entity{P}(hll_result; grad=comp_grad, op=op_result)
```

```
end
```

Here is the backprop function. It effectively propagates changes through the `grad` property of the resulting Entity instance back to the Entity instances of the arguments. Similar to static operations, the backprop functions are nearly identical to those already provided. Therefore, we will omit them from the descriptions of subsequent operations.

```
Python
function backprop!(entity::Entity{P}, entity_op::Operation{FuncType, ArgTypes})
       where {P, FuncType<:typeof(added), ArgTypes}

       if (entity.op != nothing) && (entity.op === entity_op) && (entity_op.op
       === added)
           entity_op.args[1].grad += entity.grad
           entity_op.args[2].grad += entity.grad
       else
           println("Error: Operation not supported for terminal node")
       end
end
```

Retained:

```
Python
function retained(current::Entity{P}, previous::Entity{P}) where {P}
       length(previous.hll.counts) == length(current.hll.counts) ||
       throw(ArgumentError("HllSet{P} must have same size"))

       hll_result = HllSets.intersect(current.hll, previous.hll)
       op_result = Operation(retained, (current, previous))
       comp_grad = HllSets.count(hll_result)
       return Entity{P}(hll_result; grad=comp_grad, op=op_result)
end
```

AS you can see this operation uses HllSet intersection to get retained HllSet.

Deleted:

```
Python
function deleted(current::Entity{P}, previous::Entity{P}) where {P}
```

```
       length(previous.hll.counts) == length(current.hll.counts) ||
       throw(ArgumentError("HllSet{P} must have same size")
       )
       hll_result = HllSets.set_comp(current.hll, previous.hll)
       op_result = Operation(deleted, (current, previous))
       comp_grad = HllSets.count(hll_result)
       return Entity{P}(hll_result; grad=comp_grad, op=op_result)
end
```

Difference:

```
Python
function diff(a::Entity{P}, b::Entity{P}) where {P}
       d = deleted(a, b)
       r = retained(a, b)
       n = added(a, b)
       return d, r, n
end
```

This operation allows us to run all three previous operations in the batch. We will use it in the following operation.

Advance:

```
Python
# advance - Allows us to calculate the gradient for the advance operation
# We are using 'advance' name to reflect the transformation of the set
# from the previous state to the current state
function advance(a::Entity{P}, b::Entity{P}) where {P}
       d, r, n = diff(a, b)
       hll_res = HllSets.union(n.hll, r.hll)
       op_result = Operation(adv, (d, r, n))
       # calculate the gradient for the advance operation as
       # the difference between the number of elements in the n set
       # and the number of elements in the d set
       grad_res = HllSets.count(n.hll) - HllSets.count(d.hll)  # This is the
       simplest way to calculate the gradient

       # Create updated version of the entity
       return Entity{P}(hll_res; grad=grad_res, op=op_result)
end
```

This enhanced version of the advanced function enables us to discern trends by analyzing the dynamics of changes within added, retained, and deleted subsets. This accumulated insight allows us to forecast the future state of any given Entity instance. By implementing this function across all Entity instances within the neural network, we can anticipate alterations throughout the entire system. Specifically for the SGS, this capability means that we can successfully regenerate the SGS.

```
Python
"""
This version of advance operation generates new unknown set from the current
set
that we are using as previous set.
Entity b has some useful information about current state of the set:
    - b.hll - current state of the set
    - b.grad - gradient value that we are going to use to calculate the
gradient for the advance operation
    - b.op - operation that we are going to use to calculate the gradient for
the advance operation.
            op has information about how we got to the current set b.
            - op.args[1] - deleted set
            - op.args[2] - retained set
            - op.args[3] - added set
We are going to use this information to construct the new set that represents
the unknown state of the set.
"""
function advance(::Colon; b::Entity{P}) where {P}
       # Create a new empty set
       hll_res = HllSets.create(HllSets.size(a.hll))
       op_result = Operation(adv, (a, hll_res))
       # calculate the gradient for the advance operation as
       # the number of elements in the a set
       grad_res = HllSets.count(a.hll)  # This is the simplest way to calculate
       the gradient

       # Create updated version of the entity
       return Entity{P}(hll_res; grad=grad_res, op=op_result)
end
```

Here is the backpropagation function specifically designed to complement the advance function:

```
Python
function backprop!(entity::Entity{P}, entity_op::Operation{FuncType, ArgTypes})
where {P, FuncType<:typeof(adv), ArgTypes}
       if (entity.op != nothing) && (entity.op === entity_op) && (entity_op.op
       === adv)
           if entity_op.args[1].op !== nothing
               entity_op.args[1].grad += entity.grad
           end
           if entity_op.args[2].op !== nothing
               entity_op.args[2].grad += entity.grad
           end
           if entity_op.args[3].op !== nothing
               entity_op.args[3].grad += entity.grad
           end
       else
           println("Error: Operation not supported for terminal node")
       end
end
```

Backward propagation

This is still work in progress.

Evaluating Newly Implemented Operations

In this section, we will illustrate how to utilize the implemented operations:

```
Python
# Generate datasets from random strings
s1 = Set(randstring(7) for _ in 1:10)
s2 = Set(randstring(7) for _ in 1:15)
s3 = Set(randstring(7) for _ in 1:100)
s4 = Set(randstring(7) for _ in 1:20)
s5 = Set(randstring(7) for _ in 1:130)

# Add datasets to HllSets
HllSets.add!(hll1, s1)
HllSets.add!(hll2, s2)
HllSets.add!(hll3, s3)
HllSets.add!(hll4, s4)
HllSets.add!(hll5, s5)
```

```Python
entity1 = HllGrad.Entity{10}(hll1)
entity2 = HllGrad.Entity{10}(hll2)
HllGrad.isequal(entity1, entity2)

# Access the type parameter P
P_type = typeof(entity1).parameters[1]

println("The type parameter P is: ", P_type)

entity1
```

And here is output:

```Unset
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);
```

```Python
c = HllGrad.union(entity1, entity2)c
println(c)
```

Output:

```Unset
Entity(sha1: 04abb4ce6da6da921a395c74dc8f585e91c9580f;
 hll_count: 27;
 grad: 0.0;
 op: Main.HllGrad.Operation{typeof(union), Tuple{Entity{10},
Entity{10}}}(union, (
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);

,
Entity(sha1: aaadde0638b68333c4384820a6d505f4081a20fe;
 hll_count: 16;
```

```
 grad: 0.0;
 op: nothing);

)));
```

The result of the union operation reveals not only the resulting Entity instance but also the instances of the input arguments.

Let’s run backprop function:

```python
HllGrad.backprop!(c, c.op)
println(c.op)
c
```

And it’s output:

```
Main.HllGrad.Operation{typeof(union), Tuple{Entity{10}, Entity{10}}}(union, (
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);

,
Entity(sha1: aaadde0638b68333c4384820a6d505f4081a20fe;
 hll_count: 16;
 grad: 0.0;
 op: nothing);

))
Main.HllGrad.Operation{typeof(union), Tuple{Entity{10}, Entity{10}}}(union, (
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);

,
Entity(sha1: aaadde0638b68333c4384820a6d505f4081a20fe;
 hll_count: 16;
 grad: 0.0;
 op: nothing);
```

```
))
```

The result is somewhat longer than before, as it includes a complete trace of the arguments as well.