NeuroTox

10th October 2018

Model Layout and Pseudocode of basic Artificial Construct

---===---

Overview of main components and members

---===---

```
//====------------
//==--      Notes
//====------------
```

//The Node_ID can be thought of as a reference to a node. In this example it is used as such, and at times similar to an integer value which could be thought of as the address of the node.

//This pseudo code assumes familiarity with c++ syntax and the idea

of pointers. The CAN uses an array of pointers to nodes as a scaffold to build the constructs.

//Syntax

-The structure of functions, and general logic structures follow c++ sytax. An exception is for loops which resemble the python iterator based for loops.

-For loops:

```
for (CONDITIONS_OR_ITEMS) as ITERATOR_VARIABLE
```

~CONDITIONS_OR_ITEMS: Either a set of conditions such as counting from one to ten, or a set of items such as looking through every entry in a data base.

~ITERATOR_VARIABLE: An identifier for the current iteration value, or the current item in the list.

-The continuation of a line is shown by indenting by one space for the continued lines.

//LIST OF TERMS

-Construct: The tree built that represents a pattern

-Node: The basic building block of the networks.

-Axons and Dendrites: Connections between nodes. The axon can be thought of as the sending portion of the connection and the dendrite as receiving.

-Unit of data: When referring to state nodes or inputs a single data type such as integer or character may not be sufficient to describe the input. For example you may have your input set be a string of

hashes with 128 characters each; as far as you are concerned each 128 character string is one unit of data. So for this reason I will refer to units of data rather than a specific type.

```
//====----------------
//==--      Construct
//====----------------
```

--== Contains:

-Current Active Nodes (CAN)

-Node Network

-Input and Output tables

```
--== Functions:

//Eval is used to search the node network for the input pattern.
void Eval()
```

```
//Build is used when training to construct a representation of the
 input data in the node network.
void Build()


//Takes an input into the construct so that it can be read in and
 built, or queried. The input can be a bytestring, array of integers,
 array of floats, or any set of data that can be represented by a one
 dimensional array.
void Submit_Input(Data p_Input)


//Starting at a given tier (Low_Tier) the CAN is charged until the
 topmost given tier is reached. This function is called after the CAN
 has been built and uses the scaffold erected in the CAN.
void Charge(int Low_Tier, int High_Tier)


//For all of the charged treetop nodes after evaluation gather the
 patterns that the treetop nodes represent and store the gathered
 patterns in the output. Other information about the treetops can be
 gathered, such as the charge.
void Gather_Output()



//====-----------------------------------------

//==--      Current Active Nodes (CAN)
```

//====--

--== Contains:

//The CAN requires access to the node networks members that allow for

-Access to the node network

//The CAN builds the input from the construct.

-Access to the input of the construct

//The node scaffold is a two dimensional array of Node_ID references. It is expanded to hold the construct necessary to represent the input pattern. The first index represents the height of the scaffold in tiers which is equal to the length of the input set. The number of nodes on each tier decreases with each step up the scaffold by one to form a pyramidal structure.

-Node_ID * Scaffold[Number_Of_Tiers] [Number_Of_Nodes_On_Each_Tier]

--== Functions:

//Fills out the entire CAN, using preexisting nodes where possible and creating new ones when needed.

```
void Build()


//Fills out the CAN but only with preexisting nodes, it does not
 create new ones.
void Query()


//Resizes the CAN scaffold based upon the size of the input.
void Resize()


//Gathers the state nodes associated with each unit of input.
void Fill_State()


//Gathers the state node associated with each unit of input, but does
 not create them if they are not found. Used for building a query.
void Query_State()


//Builds the node tiers after the input has been read in as state
nodes.
void Build_Tiers_Full()


//Builds the node tiers after the input has been read in as state
 nodes but does not create new nodes.
void Build_Tiers_Query()
```

```
//Reinforces the nodes currently in the CAN scaffold.
void Reinforce()




//====----------------------
//==--       Node Network
//====----------------------

--== Contains:

-Nodes arranged in tiers




--== Functions:

//Takes a unit of input data and returns the node that is associated
 with the unit of data. If no node currently exists for that unit of
 data then a new node is created and associated with the unit of
 data.
Node_ID Get_State_Node(Given_State)

//The same as Get_State_Node(), however, no node is created if the
```

unit of data is not found.

```
Node_ID Get_State_Node_For_Query(Given_State)
```

//Create a connection from p_From node to p_To node for the given dendrite.

```
void Create_Connection(Node_ID p_From, Node_ID p_To, string
 p_Dendrite)
```

//When the CAN is building it needs to know if there is a node on a higher tier linking two lower tier nodes together. This function searches for the linking node, if no node exists then one is created.

```
Node_ID get_Upper_Tier_Connection(Node_ID left_Node, Node_ID
 right_Node)
```

//When the CAN is building for a query this function searches for then returns the upper tier linking node, however, if one does not exist then false is returned instead of a node.

```
Node_ID Does_Upper_Tier_Connection_Exist(Node_ID left_Node, Node_ID
 right_Node)
```

//The pattern represented by the node is backpropagated into an output suitable to hold it. This is done by using an algorithm to trace the lower legs in the correct order to retrieve the information originally used in the construction of the tree that the current node is at the top of.

```
Node_ID Get_Node_Output_Pattern(Node_ID p_Node)
```

The backpropagation is initiated at the topmost treetop node. During backpropagation the left dendrite is done slightly different than the right one. The left sends a signal down both of its dendrites; whereas the right one only sends a signal down its right dendrite. This creates a wave effect when plotted that outputs the pattern used to construct the tree exactly as it was input. This is why keeping dendrite order is so important; if we did not track dendrite order the output would be a meaningless mess.

```
void Backpropagate(Node_ID p_Node)

//Backpropagate a left dendrite linked node.
void Backpropagate_Left(Node_ID p_Node)

//Backpropagate_Right a right dendrite linked node.
void Backpropagate_Right(Node_ID p_Node)
```

```
//====----------
//==--        Node
```

//===----------

--== Contains:

-Current Charge

-Base Charge

//The action potential threshold and modifier charge function as a filter when evaluating a network. Without these values then every single node that gets a charge will pass that charge all the way to the top of the network. If you have a deep network with few inputs you can charge the entire network at once which is usually not desirable.

-Action Potential Threshold (APT)

-Modifier Charge

-Reinforcement Counter (RC)

-Axons linking to the dendrites of higher nodes.

//Node that state nodes do not have two dendrites; rather they link to a single node or contain a unit of data.

-Left Dendrite linking to a lower node.

-Right Dendrite linking to a lower node.

//Normal nodes do not have a state, unless you store the entire pattern represented by each node with that node to allow for skipping the backpropagation saving on calculations.

-Data_Unit State

--== Functions:

//Adds to the nodes current charge, if the charge is over the action potential threshold then it fires and sends a charge to all of the nodes its axons connect to in the higher tiers.

void Charge(float p_Charge)

//Adds to the nodes current charge, the Modifier charges role is to act as a filter of sorts. When using the RC charging you modify the input charge based upon the RC score.

void Add_To_Charge(float p_Charge)

//Checks if the right dendrite value matches the given p_Right_Dendrite

bool Does_Right_Dendrite_Match(Node_ID p_Right_Dendrite)

---=================================---

Main components pseudo code.

---===================================---

```
//====----------------

//==--       Construct

//====----------------
```

--== Contains:

-Current Active Nodes (CAN)

-Node Network

-Input and Output tables

--== Functions:

```
//Eval is used to search the node network for the input pattern.
void Eval()
{
   Gather_Input()
```

```
   CAN.Query()
   Charge()
   Gather_Output()
}

//Build is used when training to construct a representation of the
 input data in the node network.
void Build()
{
   Gather_Input()
   CAN.Build()
}

//Takes an input into the construct so that it can be read in and
 built, or queried. The input can be a bytestring, array of integers,
 array of floats, or any set of data that can be represented by a one
 dimensional array.
void Submit_Input(Data p_Input)
{
   Reads p_Input into the table or array where the CAN can access it.
}

//Starting at a given tier (Low_Tier) the CAN is charged until the
 topmost given tier is reached. This function is called after the CAN
 has been built and uses the scaffold erected in the CAN.
```

```
void Charge(int Low_Tier, int High_Tier)
{
   for (Low_Tier to High_Tier as Current_Tier)
   {
      for (Nodes in CAN.Scaffold on Current_Tier as Current_Node)
      {
      Current_Node->Charge()
      }
   }
}
```

```
//For all of the charged treetop nodes after evaluation gather the
 patterns that the treetop nodes represent and store the gathered
 patterns in the output. Other information about the treetops can be
 gathered, such as the charge.
void Gather_Output()
{
   for (Each charged treetop node as Current_Treetop_Node)
   {
       Current_Treetop_Node.get_Pattern()
      Take the pattern gathered and add it to the output along with
      any other data wanted such as charge, reinforcement counter,
      etc.
   }
}
```

```
//====----------------------------------------
//==--        Current Active Nodes (CAN)
//====----------------------------------------
```

--== Contains:

//The CAN requires access to the node networks members that allow for

-Access to the node network

//The CAN builds the input from the construct.

-Access to the input of the construct

//The node scaffold is a two dimensional array of Node_ID references. It is expanded to hold the construct necessary to represent the input pattern. The first index represents the height of the scaffold in tiers which is equal to the length of the input set. The number of nodes on each tier decreases with each step up the scaffold by one to form a pyramidal structure.

-Node_ID * Scaffold[Number_Of_Tiers] [Number_Of_Nodes_On_Each_Tier]

--== Example CAN with input of “1001”

//After Resize() with the length of 4 on an input.

0-> <_NULL_> <_NULL_> <_NULL_> <_NULL_>

1-> <_NULL_> <_NULL_> <_NULL_>

2-> <_NULL_> <_NULL_>

3-> <_NULL_>

//State layer has been filled.

0-> <_1_> <_2_> <_2_> <_1_>

1-> <_NULL_> <_NULL_> <_NULL_>

2-> <_NULL_> <_NULL_>

3-> <_NULL_>

//All upper tier nodes have been filled out.

0-> <_1_> <_2_> <_2_> <_1_>

1-> <_3_> <_4_> <_5_>

```
2->  <_6_> <_7_>

3->  <_8_>
```

--== Functions:

```
//Fills out the entire CAN, using preexisting nodes where possible
 and creating new ones when needed.
void Build()
{
   Resize(Length_Of_Input)
   Fill_State()
   Build_Tiers_Full()
   if (Tracking reinforcement values){ Reinforce() }
}

//Fills out the CAN but only with preexisting nodes, it does not
 create new ones.
void Query()
{
   Resize(Length of the current input)
   Fill_State()
   Build_Tiers_Full()
```

```
}

//Resizes the CAN scaffold based upon the size of the input.
void Resize()
{
   //This is where the access to the construct input first comes into
    play.

   Number_Of_Tiers = Length_Of_Input
   for (Number_Of_Tiers as Current_Tier)
   {
      //A pattern forms a pyramidal tree structure where each tier
       from bottom to the top has one less node than the tier below
       it. When the top tier is reached only one node is left, this
      is the treetop node.

       Number_Of_Nodes in Current_Tier =
      (Number_Of_Tiers - Current_Tier)
   }
}

//Gathers the state nodes associated with each unit of input.
void Fill_State()
{
   //The state tier reads in the input so its length is equal to the
```

```
    input length.
   for (Each unit of data in Input as Current_Data_Unit)
   {
     //Each unit of input data corresponds to a node on the state
      tier. So the input "101" has three nodes on the state tier.

     Nodes.Get_State_Node(Current_Data_Unit)
   }
}

//Gathers the state node associated with each unit of input, but does
 not create them if they are not found. Used for building a query.
void Query_State()
{
   //The state tier reads in the input so its length is equal to the
    input length.

   for (Each unit of data in Input as Current_Input_Data_Unit)
   {
     //Each unit of input data corresponds to a node on the state
       tier. So the input "101" has three nodes on the state tier.
     The state tier is the lowest tier so it has an index of 0
      (Scaffold[0]).

     Scaffold[0] [index of Current_Input_Data_Unit] =
```

```
      Nodes.Get_State_Node_For_Query(Current_Input_Data_Unit)
   }
}

//Builds the node tiers after the input has been read in as state
 nodes.
void Build_Tiers_Full()
{
   //The tree build from the pattern culminates in a single node,
    this node is the treetop node. We do not go to this tier when
    building because to do so would be requesting a node for a tier
    higher than the highest tier in the current tree.

   for ((Number_Of_Tiers - 1) as Current_Tier)
   {
      //Each node has two lower connections with the exception of the
       state tier, because of this we do not search from one end of
       the tier to the other. If we were to go to the end node we
       would have the end node as a left leg and no right leg to
       search for.

      for ((Number_Of_Nodes in Current_Tier) - 1 as Current_Node)
      {
         //Assuming the nodes are in an array the current node may be
           at index
```

```
      Nodes_In_Current_Tier[Current_Node]
          While the next node may be at
      Nodes_In_Current_Tier[Current_Node + 1].

          Nodes.Get_Upper_Tier_Connection(Current_Node, Next_Node)
      Set the current CAN reference node to hold the ID of the
       upper tier connection gathered.
      }
   }
}

//Builds the node tiers after the input has been read in as state
 nodes but does not create new nodes.
void Build_Tiers_Query()
{
   //The tree build from the pattern culminates in a single node,
    this node is the treetop node. We do not go to this tier when
    building because to do so would be requesting a node for a tier
    higher than the highest tier in the current tree.

   for ((Number_Of_Tiers - 1) as Current_Tier)
   {
      //Each node has two lower connections with the exception of the
        state tier, because of this we do not search from one end of
        the tier to the other. If we were to go to the end node we
```

```
        would have the end node as a left leg and no right leg to
        search for.

      for ((Number_Of_Nodes in Current_Tier) - 1 as Current_Node)
      {
         //Assuming the nodes are in an array the current node may be
           at index
       Nodes_In_Current_Tier[Current_Node]
          While the next node may be at
       Nodes_In_Current_Tier[Current_Node + 1].

         Nodes.Does_Upper_Tier_Connection_Exist(Current_Node,
           Next_Node)
      Set the current CAN reference node to hold the ID of the
            upper tier connection gathered.
      }
   }
}

//Reinforces the nodes currently in the CAN scaffold.
void Reinforce()
{
   for (Number_Of_Tiers as Current_Tier)
   {
      for (Number_Of_Nodes in Current_Tier as Current_Node)
```

```
    {
        Current_Node.Reinforce()
    }
  }
}
```

//====----------------------

//==--   Node Network

//====----------------------

--== Contains:

-Nodes arranged in tiers

--== Functions:

//Takes a unit of input data and returns the node that is associated with the unit of data. If no node currently exists for that unit of data then a new node is created and associated with the unit of data.

```
Node_ID Get_State_Node(Given_State)
{
   Search nodes on state tier for Given_State
   if (State node was found)
   {
     Return the found node
   }
   else
   {
     Create new state node with Given_State
     Return the newly created node
   }
}

//The same as Get_State_Node(), however, no node is created if the
 unit of data is not found.
Node_ID Get_State_Node_For_Query(Given_State)
{
   Search Nodes on State tier for Given_State
   if (State node was found)
   {
     Return the found node
   }
   else
   {
```

```
      Return no node found
   }

}

//Create a connection from p_From node to p_To node for the given
 dendrite.
void Create_Connection(Node_ID p_From, Node_ID p_To,
 string p_Dendrite)
{
   if (p_Dendrite == "left")
   {
      p_From.Add_Axon(p_To)
       p_To.Left_Dendrite = p_From
   }

   if (p_Dendrite == "right")
   {
      p_From.Add_Axon(p_To)
      p_To.Right_Dendrite = p_From
   }
}

//When the CAN is building it needs to know if there is a node on a
 higher tier linking two lower tier nodes together. This function
```

```
 searches for the linking node, if no node exists then one is
 created.
Node_ID get_Upper_Tier_Connection(Node_ID left_Node, Node_ID
 right_Node)
{
   Does_Upper_Tier_Connection_Exist()
   if (a connection was found)
   {
      return the connection found
   }
   new_Node = create a new node
   Create_Connection(left_Node, new_Node, “left”)
   Create_Connection(right_Node, new_Node, “right”)
   return new_Node
}

//When the CAN is building for a query this function searches for
 then returns the upper tier linking node, however, if one does not
 exist then false is returned instead of a node.
Node_ID Does_Upper_Tier_Connection_Exist(Node_ID left_Node, Node_ID
 right_Node)
{
   //The axons are an array of Node_IDs so they can be iterated
    through.
   for (every axon in the left_Node as Current_Axon)
```

```
   {
      if (Does_Right_Dendrite_Match(Current_Axon))
      {
      return Current_Axon.Node_ID
      }
   }
}
```

//The pattern represented by the node is backpropagated into an output suitable to hold it. This is done by using an algorithm to trace the lower legs in the correct order to retrieve the information originally used in the construction of the tree that the current node is at the top of.

```
Node_ID Get_Node_Output_Pattern(Node_ID p_Node)
{
   Backpropagate(p_Node)
}
```

The backpropagation is initiated at the topmost treetop node. During backpropagation the left dendrite is done slightly different than the right one. The left sends a signal down both of its dendrites; whereas the right one only sends a signal down its right dendrite. This creates a wave effect when plotted that outputs the pattern used to construct the tree exactly as it was input. This is why keeping dendrite order is so important; if we did not track dendrite

```
 order the output would be a meaningless mess.
void Backpropagate(Node_ID p_Node)
{
   //Start the process with p_Node.

   Backpropagate_Left(p_Node)
}


//Backpropagate a left dendrite linked node.
void Backpropagate_Left(Node_ID p_Node)
{
   //If a left dendrite exists then initiate a back propagation along
      it, then Along the right side.

   //If no left dendrite exists then that means that this node is a
      state node and the state should be output instead.

   if (p_Node.Dendrite_Left != NULL)
   {
      Backpropagate_Left(p_Node.Dendrite_Left)
      Backpropagate_Right(p_Node.Dendrite_Right)
   }
   else
   {
     Add p_Node.State to the pattern output.
```

```
   }
}

//Backpropagate_Right a right dendrite linked node.
void Backpropagate_Right(Node_ID p_Node)
{
   //If a right dendrite exists then initiate a back propagation
    along it, then along the right side.

   //If no right leg exists then that means that this node is a state
    node and the state should be output instead.

   if (p_Node.Dendrite_Right != NULL)
   {
      Backpropagate_Right(p_Node.Dendrite_Right)
   }
   else
   {
     Add p_Node.State to the pattern output.
   }
}
```

//====----------

//==--   Node

//====----------

--== Contains:

-Current Charge

-Base Charge

//The action potential threshold and modifier charge function as a filter when evaluating a network. Without these values then every single node that gets a charge will pass that charge all the way to the top of the network. If you have a deep network with few inputs you can charge the entire network at once which is usually not desirable.

-Action Potential Threshold (APT)

-Modifier Charge

-Reinforcement Counter (RC)

-Axons linking to the dendrites of higher nodes.

//Node that state nodes do not have two dendrites; rather they link to a single node or contain a unit of data.

-Left Dendrite linking to a lower node.

-Right Dendrite linking to a lower node.

//Normal nodes do not have a state, unless you store the entire pattern represented by each node with that node to allow for skipping the backpropagation saving on calculations.

-Data_Unit State

--== Functions:

//Adds to the nodes current charge, if the charge is over the action potential threshold then it fires and sends a charge to all of the nodes its axons connect to in the higher tiers.

```
void Charge(float p_Charge)
{
   Add_To_Charge(p_Charge)

   if (Current_Charge >= APT)
   {
      for (Every axon as Current_Axon)
      {
         Current_Axon->Charge(Base_Charge)
      }
   }
```

```
}

//Adds to the nodes current charge, the Modifier charges role is
 to act as a filter of sorts. When using the RC charging you modify
 the input charge based upon the RC score.
void Add_To_Charge(float p_Charge)
{
   if (Using_RC)
   {
      Current_Charge += (p_Charge * Modifier_Charge) * RC
   }
   if (!Using_RC)
   {
      Current_Charge += p_Charge * Modifier_Charge
   }
}


//Checks if the right dendrite value matches the given
 p_Right_Dendrite
bool f_Does_Right_Dendrite_Match(Node_ID p_Right_Dendrite)
{
   if (Right_Dendrite == p_Right_Dendrite)
   {
     Return true
```

```
    }
    else
    {
      Return false
    }
}
```