Hierarchical Temporal Memory Agent in standard Reinforcement Learning Environment

and beefing up framework performance

An-Pang Chang

January 27, 2020

National Sun-Yet-San University

Bachelor's degree project report

Contents

List of Figures

List of Tables

1 Introduction

There has been an explosion in the filed of Artificial Intelligence(AI) in the recent years. With new techniques, especially neural networks, computers can learn to separate dogs from cats, beat the best human in ~~chess~~[1] Go[1] and even generate anime faces[2]. - Given enough data and computing resource is provide to train the algorithm. However, these learning algorithms may not generalize outside their domain and they mostly rely on brute-force instead of generalization and reasoning to perform tasks.

If we plot neuron based algorithms on a 1D spectrum, where the axis is how mathematical they are. Neural networks are located at an far end of the spectrum where it operates in a purely mathematical way. And spiking neural networks located at the opposite end. Where the simulated neurons are made as similar as possible to their real-world version. While most ML algorithms falls around the extremes of the line. This project chooses to use an algorithm located at the middle of the spectrum. It attempts to be biologically possible while only preserving the core operating principles.

The algorithm we selected is Hierarchical Temporal Memory (HTM). HTM is a machine learning model designed to emulate the human Neocortex. Which is believed to be the source of human intelligence[3]. Since the first description of the model in 2004 and the public release in 2016. The model has gained lots of new additions, features and a group of followers. But despite the developments around the model, it is still far from being a fully functional model of the Neocortex. And thus this project is looking forward to explore some of the problems faced by HTM and find potential solutions. Especially regarding the topic of computing performance and reinforcement learning.

1.1 Contribution

Up to now there are only four relevant work regarding RL using HTM. The work done by Otahal[4] for his MSc thesis, Gomez[5] in his BSc project, Sungur[6] for his MSc thesis and Heyder[7] for their Bachelor's thesis. In order, their work provides the foundation for combining HTM and RL, comparing RL based on RL to Q-Leaning under a Markov Decision Process, combining HTM + RL with TD(λ) based agents and benchmarking HTM based learning agents under formal environments.

However all the mentioned work are limited to running on a relatively small HTM network due to the frameworks available at the time. And since then, new neural phenomenons have been discovered. Thus our project aims to provide two contributions.

- Provide a fast HTM library
- Incorporate newly discovered neural phenomenon into HTM based agents

[1]Just a joke, computer can beat the best human in chess since 1997. Hmm... I can't find something I can cite.

2 Background

2.1 Hierarchical Temporal Memory

Hierarchical Temporal Memory is a biologically inspired model, modeling the behavior of the Neocortex. It is an online, unsupervised and an local learning algorithm (i.e. it does not learn via gradients or any global state). Like neural networks, the model is made of so called "layers" and "neurons". Though they work very differently.

2.1.1 Sparse Distributed Representation (SDR)

All input and output in HTM are encoded into Sparse Distributed Representation. These are spares bit fields that build the basis of the computing system. Typically the density of a SDR ($num\_of\_1\_bits/num\_all\_bits$) is recommended to be around 2% [8]

2.1.2 HTM Neurons

At the core of HTM is the HTM-neuron, a model of the pyramidal neuron[9].

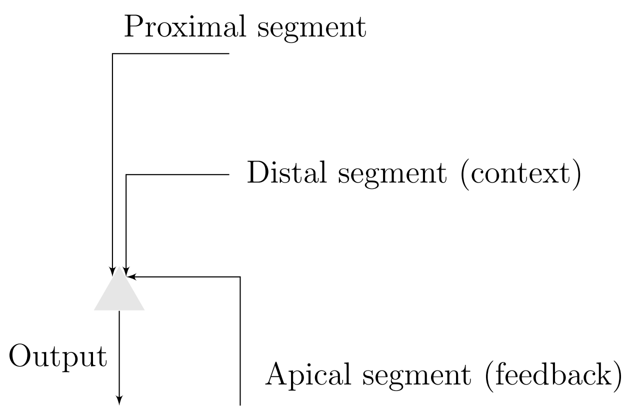

Figure 1: An illustration of a HTM Neuron

From a computation point of view. The HTM neuron have 3 inputs and 1 output. The output of a HTM neuron is binary, either an 1 or a 0. The proximal input(dendrites) can be loosely viewed as the input to neurons in an artificial neural network, it is the main source of input and the main factor determining if the neuron fires or not. The apical and proximal dendrites on the other hand works on a group level, They regulate how a bunch of neurons react. Although HTM neurons have multiple inputs. HTM doesn't need all of them to be wired up to operate. In fact proximal dendrites are the only one absolutely needed.

Figure 1 is a simplified version of the HTM neuron. In a real-world situation there will be more than a single Proximal, Distal and Apical segment coming into the HTM neuron. Where each dendrite is a match-and-threshold unit as illustrated in Figure 2. Synapses grows from the segment to input cells and the inputs flow trough these synapses to the segment (given the permanence of the synapse is great enough). Then segment activates when enough input signal reaches it. When activated, a signal is sent to the HTM neuron.

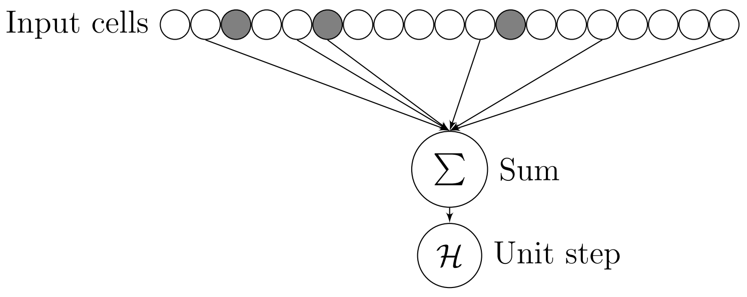

Figure 2: An illustration of a segment

Mathematically and programmatically, Segments evaluates the function:

```python
def segment(input, synapses, permanence, connected_threshold,
    active_threshold):
  assert(len(synapses) == len(permanence))
  s = 0
  for s, p in zip(synapses, permanence):
    s += 1 if p > connected_threshold and input[s] is True else 0
  if s < active_threshold:
    s = 0
  return s
```

Just to illustrate that a HTM neuron can have many segments.

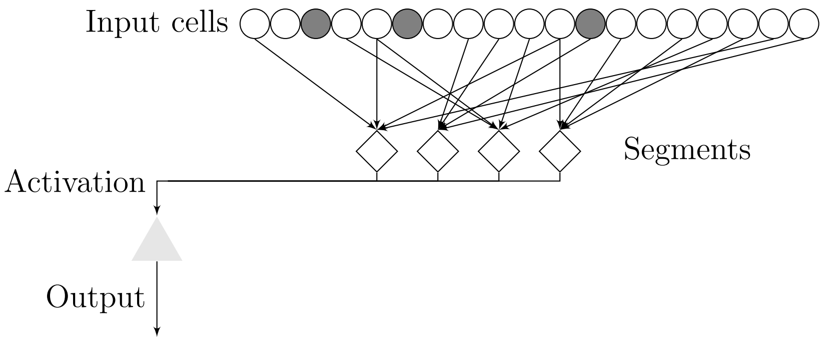

Figure 3: Multiple segments

It is good to have a model of the neuron. But it makes no good if the model can't learn anything. The learning algorithm HTM neuron is very simple, resembling how STDP[10] used in spiking neural networks.

1. If the cell is not active, don't learn
2. If the segment is not active, don't learn
3. Enumerate over every synapse in the active segments
4. If the input which the synapse is connected to is on, increase the permanence
5. Otherwise, decrease the permanence

That's it.

The half-harted joking tone aside, this is a surprisingly simple learning rule. But also very interesting. The act of not updating inactive segments contributes to the realtime learning capability of HTM and avoided the catastrophic forgetting problem.

2.1.3 Spatial Pooler

Spatial Pooler[11], or SP for short is the simplest and the first layer typically used HTM. The function of a Spatial Pooler is to convert an arbitrary SDR into a SDR with fixed density and semantical meaning. It consists of a field of cells, where each cell have it's own perspective field. The perspective field describes which input cell a given cell is connected to. Consider the topology to be more or less like a fully-connected/dense layer in a neural network. (Though it works very differently).

From the point of view of a HTM neuron. Spatial Poolers are an array of HTM neurons where they are solely made of one proximal segment. They perform some computation using the input and spits out whatever the result it generates.

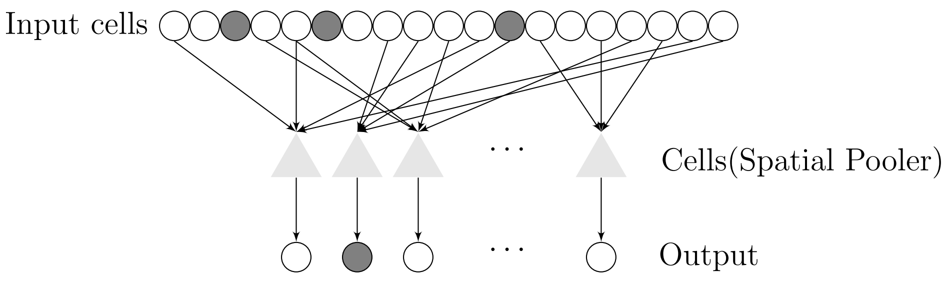

Figure 4: An illustration of a Spatial Pooler

The inference process of a SP is simple. Since it only contains a single proximal segment and no distal/apical segments regulating it's behavior. The inference process is as simple as summing the input of the cells and finding out weather the sum is greater than the activation threshold. However, this simple inference rule leads to some cell being hyper-activated and density of the SP's output goes up gradually over time. Thus boosting and inhibition is introduced to regulate the SP.

Boosting is the process of increasing or decreasing each cell's activity based on the cell's past activation history. Allowing under-activating cells to express itself and to have a chance to learn (remember I said that only activated cells learn?). Given P the average activity of the cell, target activity T and boosting strength S. We can calculate the boost factor $F = exp(-(T - P) * S)$. The boost factor is calculated for each cell and multiplied to the sum of the connect input before checking weather the segment should be activated. If you are very good at math and equations, you'll immediately see how the boost factor F is 1 when $P = T$. When $P < T$, F becomes greater then 1, which is multiplied with to cause the cell to become more activated. And while $P > T$, the boost factor becomes lesser than 1, suppressing the activation of the neuron.. Figure 5 shows how boosting stabilizes activation.

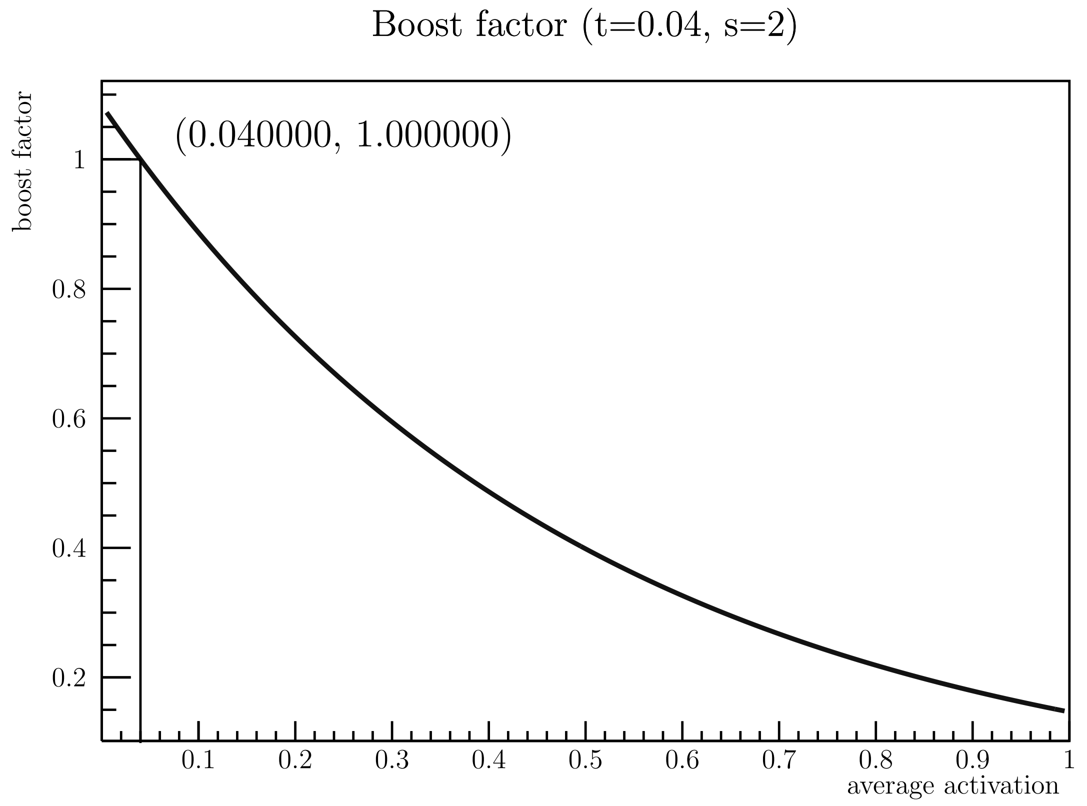

Figure 5: Plot of boost factor given different average activation

There are two inhibition method in HTM, but only global inhibition will be discussed here. The other method, local inhibition, although being more biologically consistent, is too computationally expensive that virtually no one uses it. Global inhibition is the process of making only the top K cells with the highest activation being allowed to be active, cells that have enough activation to surpass the activation threshold but not in the top K are suppressed. Forcing only at most around (since cells can have the same activation strength) K cells to be activated. This ensures stable activation each time as boosting can only promote stable activation over a long time period.

The learning process of a SP is simple, as mentioned in section 2.1.2, only active segments in active cells learn, others don't.

2.1.4 Temporal Memory

Temporal Memory(TM) is the other main layer in HTM. From a high-level point of view, TM is a sequence memory. It learns and predicts sequence on-the-fly. i.e. Train a TM on an repeated sequence of {1, 2, 3}. Then ask TM what's after 2, it will respond 3.

Further more, it have some welcoming properties.

- Predictable behavior when met with ambiguity
- Predicts nothing when condition is unknown
- Does not forget after training on new sequence

Temporal Memory is also setup in a way that when met with ambiguity (ex: trained with both sequence {1,2,3} and {3,2,1}). When the TM is asked to predict that's after 2 without giving it context, the TM will respond with "both 1 and 3". Likewise, training on the same sequence, asking the TM to predict what's after 6 will cause the TM to respond "nothing" since it have never seen the input 6 nevertheless what's after 6. While not forgetting what's previously learned is a general feature of HTM, mainly thanks to only updating weights for active segments. You can train a TM on {1, 2, 3}, then train on {4, 5, 6}. When asked what's after 2, the TM can recall what have been learned form the 1st training and reply 2.

From the view of HTM neurons, TM is formed by an array of mini-columns - HTM neurons that are stacked together. They act as a group and are regulated by the distal and apical segments as shown in Figure 6.

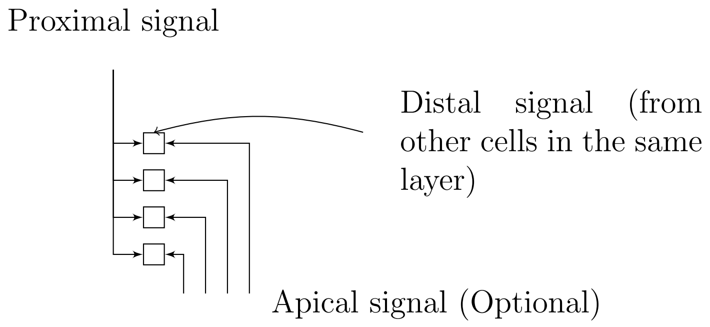

Figure 6: A mini-column with 4 cells per column

The mini-column works as a complex state machine, following the rules:

1. If proximal signal is active but no cell is in a predictive state, every cell turns active.
2. If proximal signal is active and at least one cell is predictive, the predictive cells become active
3. If the proximal signal is not active, every cell becomes inactive.
4. If Apical segments exists and there are more than 1 cell in an active state, cells without an active apical segment turns off.
5. Cells become predictive when one of it's distal segments become active.

In general the proximal segments are the main driving force running the mini-column. The distal segments allows mini-columns to predict what might be happening in the future based on past activation history. And apical segments acts like a guide to reduce the ambiguity of the mini-column's internal context.

2.1.5 Comparing to real neurons

In reality, layers in the cortical column performs both Spatial Pooling and Temporal Memory at the same time. It is only for the convenience and abstraction that it is separated in HTM.

2.2 NuPIC - the reference HTM implementation

NuPIC[12] (Numenta Platform for Intelligent Computing) is the official and the reference implementation of HTM by Numenta. Though we would like to use NuPIC in this project. The fundamental design of NuPIC leads to it's low performance and other problems.

- Unnecessary memory allocation
- Constantly switching between dense and sparse representation
- No parallel processing
- Bad memory management in the core library
- Only Python2 support
- Development slowed down
- Unpleasant API

2.3 HTM.core - the community fork of NuPIC

Since the development of NuPIC have slowed down in 2018. The community have forked NuPIC into HTM.core[13] and have added new features like Python3 support and various optimizations. But problems from NuPIC still exists. Including:

- Unnecessary memory allocation
- Constantly switching between dense and sparse representation
- No parallel processing

3 Etaler - our implementation of HTM

Etaler[14] is our high performance HTM framework in the C++ language. Mainly providing the following features.

- A C++ implementation of HTM algorithm
- An OpenCL implementation of HTM algorithm
- Common Tensor operators

We also decided to adapt the data-oriented design that most deep learning framework uses; instead of the object-oriented design that NuPIC and HTM.core employs.

3.1 Tensor and backends

To support both CPU and OpenCL at the same time and select-able at runtime. Etaler implements a front-end(API)/back-end(memory and compute) architecture. i.e. When Etaler is asked to create a new tensor/perform operations, the request is forwarded to a backend, processed then returned. *Note: Besides a few exceptions, asking the OpenCL backend to perform operations on a tensor created by the CPU backend will result either in an abortion or exception.*

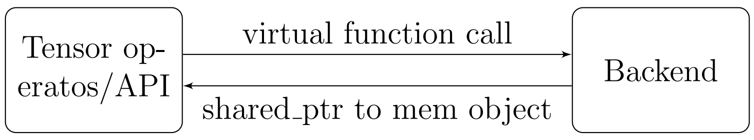

Figure 7: How backend/API calls works

At the time of writing this report, Etaler supports[2] tensors of several internal data types[3] as shown in Table 1.

Storage type	DType
int	Int32
float	Float
bool	Bool
half	Half

Table 1: Supported data type and type ID

Details will be explained in later sections.

[2]half (half precision floating point values) are supported on x86 CPU with software emulation, native hardware support under an Aarch64 processor and supported on OpenCL devices if the *cl_khr_fp16* extension is available.

[3]Name in OpenCL terms.

3.2 General backend design and workflow

et::Backend is the parent class of all backends. Which provides empty virtual function of HTM algorithm and tensor operators (calling them throws exceptions stating the function is not available). Then we inherent from this class to provide actual implementation of the functions.

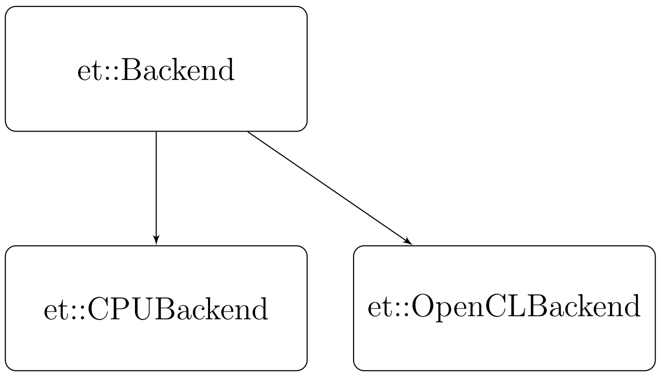

Figure 8: Backed inheritance

3.2.1 The implementation of Tensor

Typically the first thing users requests the backend is to create a tensor. Yet, Tensors in Etaler is quite a convoluted object. The reasoning follows:

1. Every backend must have a *createTensor* function
2. Ideally *createTensor* should return a tensor
3. Tensors need to know which backends it belongs to (to call the correct tensor operators)
4. Tensor.hpp have to be included in Backend.hpp
5. Backend.hpp have to be included in Tensor.hpp
6. Happy circular dependency! $_{notreally}$

The situation must be avoided, otherwise the entire system won't even compile. Unfortunately the solution we came up is quite convoluted.

1. *BufferImpl* is a class, which works as a proxy for backends to store data
2. *TensorImpl* is a class, in which
   - Stores a pointer to Backend
   - Backend is forward declared (no need to include Backend.hpp)
   - Also stores a pointer to *BufferImpl*
3. When *createTensor* is called, the backend allocates memory and store the memory object in *XXXBufferImpl*
4. Then allocate *XXXTensorImpl*, point the buffer pointer to the one we got from the last step. And point the backend pointer to *this*.

5. Return the *XXXTensorImpl* as a *TensorImpl* pointer to the caller
6. The *Tensor* class automatically wraps around the returned pointer.

There's actually more going on. Including using *std::shared_ptr* to handle reference counting, setting up mechanism so the backend won't be deallocated before all the tensors it created can release themselves, etc... They are important details of the implementations but doesn't effect the discussion here.

3.2.2 Tensor operators and HTM algorithm APIs

One key observations we made is that like neural networks, HTM can be composed by several tensor operations. Somewhat complex operations but nevertheless. And some of these operations are reusable across different HTM layers.

For example, Figure 9 shows the Spatial Pooler can be decomposed into 3 operations.

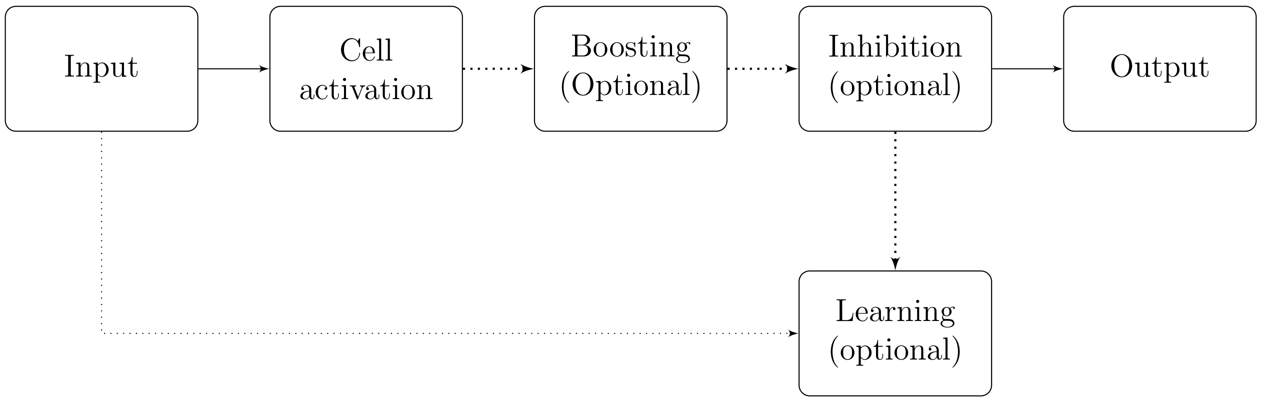

Figure 9: Decomposing the Spatial Pooler's operation

And the Temporal Memory can also be decomposed into a few operations.

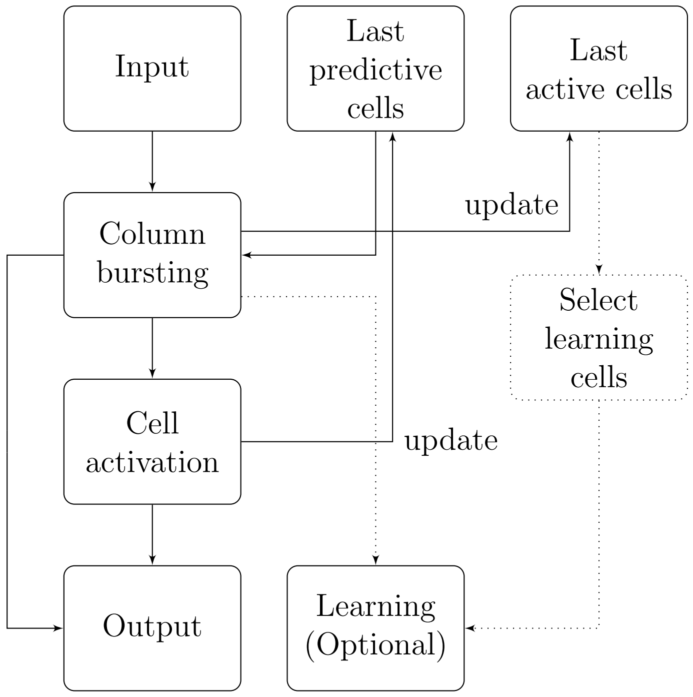

Figure 10: Decomposing the Temporal Memory's operation

Though the graphs for the two layers look very different. They contain few nodes of the exact same operation - the Learning and the Cell Activation node. This allows us to implement HTM in a modular manor. Each operation can be implemented as a method in the backend and called when needed. Furthermore, theoretically we can have the CPU work on the first half of the problem then send the needed data to the GPU and ask the GPU to do the second half. No saying this is faster, but it is a great tool for debugging and experiments. The same modular nature holds true for ordinary tensor operators.

3.3 Tensor and SDR, synapse representation

Tensors in Etaler acts like any tensor from any library, such as the ones in PyTorch[15]. From a user's perspective, they are N-Dimensional arrays with operator overloads to perform common operations (subscription, addition, multiplication, etc...)

Unlike NuPIC and most other HTM implementations. Which internally represents synapses as a structure of an *int* and a *float* to indicate which cell the synapse is connecting to and the permanence of the synapse. Then an array (actually, an *std::vector*) of synapses form a segment, an array of segments forms a cell and cells forms a SP or a TM. Etaler represent the synapses of any SP/TM as two tensors of the same shape. A tensor of *int* and a tensor of *float*. Where the last dimension of the tensors represents the amount of synapses in a cell (Etaler implements 1 segment per cell, so segments are ignored) and the rest of the dimensions represents the shape of the cells.

For example, say we have a field of cells of shape {16, 32} (a rectangular region of neurons consists of 16 by 32 neurons) and each cells can have up to 128 synapses. Then the cells are represented as two tensors of shape {16, 32, 128}.

Also unlike NuPIC, which represents SDRs as both binary arrays and sparse arrays. Etaler always represents SDR as a tensor of type *bool*.

3.4 CPU backend

The CPU backend is straight forward. Everything is written in C++ and parallelized using TBB. The only complication we met is to support different types of input for different fields. (ex: *foo()*'s first parameter can be *int* or *bool* and the second parameter can be *int* or *float*). This is solved by ≈200 lines of template and template-meta programming to provide the necessary investiture to detect pre-condition failures in a convenient manor and throwing exceptions accordingly.

3.5 OpenCL backend

The OpenCL backend is a lot more complex due to the nature of OpenCL. While the basic structure is exactly like the CPU backend's. There are a few difference. Mainly:

1. Compiles OpenCL kernels and cache them in memory at runtime.
2. JIT compile kernels instead of using templates.
3. Optimized towards running on GPUs

The first and the last decision comes for free. The only reason to have a OpenCL backend is then we can run HTM on a GPU, so we are going to optimize towards it. And the default OpenCL programming model is to have everything compiled while running. - Thanks to the programming model, we can inject run-time information into the OpenCL kernels to allow compile-time evaluation and more optimizations. Leading to a slightly better performance.

Figure 11 shows how a operation request and kernel compilation/generation is handled in the OpenCL backend.

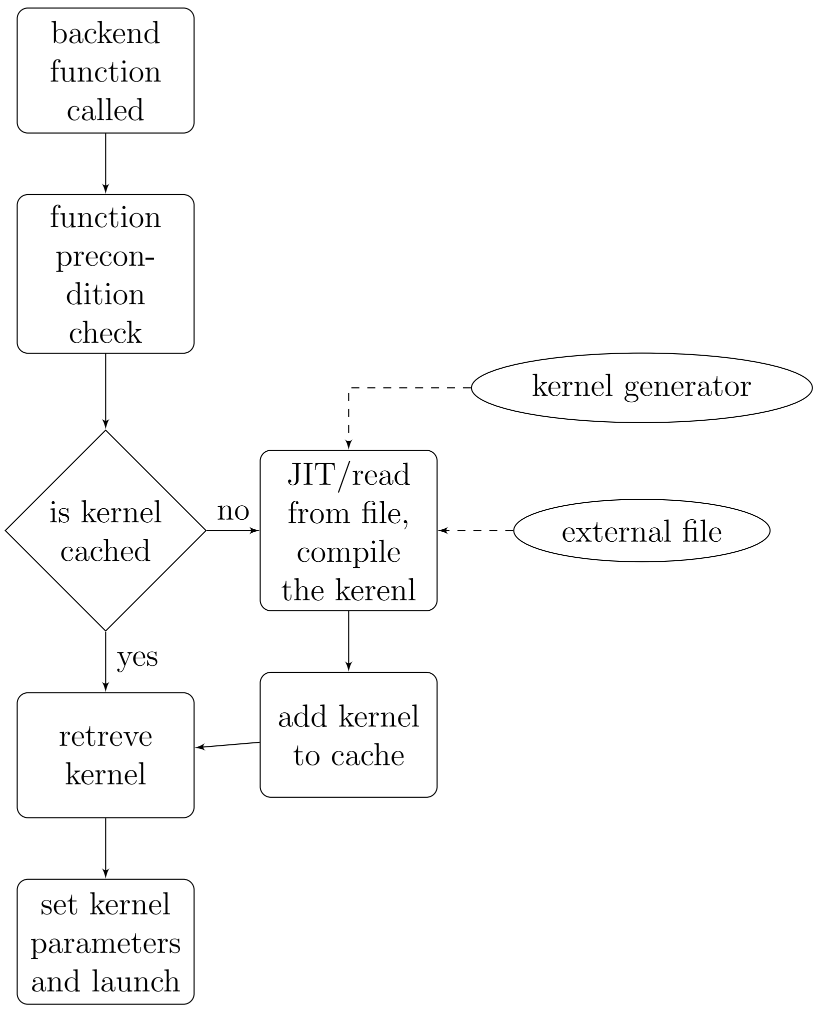

Figure 11: Typical OpenCL backend function control flow

3.6 Heterogeneous computing and backend inter-operation

As modern computers and compute servers may have more then a single beefy CPU or a beefy GPU. The ability to distribute compute on CPU and all OpenCL devices is critical to run large experiments. We have to also make the distribution of computing as seamless as possible to the user.

The design of Etaler's backend inter-op system starts with a few observations.

1. HTM layers are typically small.
    - No need to run the same layer on different backends simultaneously
    - i.e. No need for layer parallelism
2. Data parallelism is mostly useless in HTM (since it runs in real-time)
3. Data transfer takes way less time than inference+learning
4. Layers in complex HTM networks can mostly run independently

Under these observations, we assume that the optimal solution would be to spread the execution of layers across backends, then transfer the outputs to the backend where it is needed.

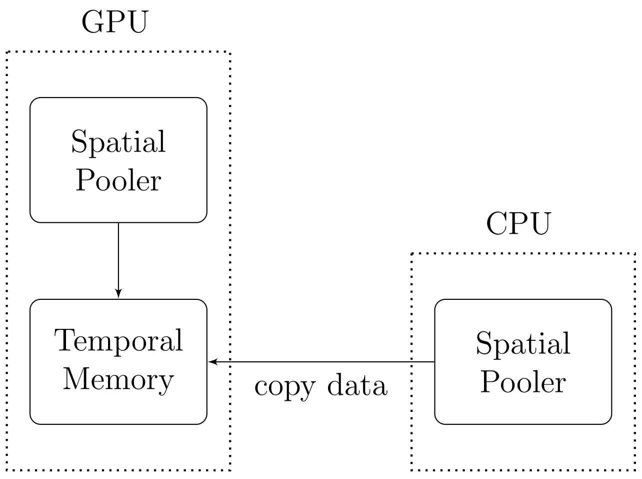

Figure 12: Example use of multiple backends

3.7 Performance

Here we demonstrate the resulting performance of our HTM system comparing to the community developed HTM.core. Particularly, we are comparing on an AMD Ryzen7 1700 Oct-core Processor and a Nvidia RTX 2080Ti GPU connected via a PCIe 3.0 x8 link.

3.7.1 Spatial Pooler

Figure 13 is our benchmark of our Spatial Pooler implementation by crafting a set of random SDR with 10% density, feeding them one-by-one into the each Spatial Pooler implementation. Recording how long et takes on average to perform an inference and learning step.

The Spatial Poolers are bencharked under the following parameter.
input_shape = output_shape
global_density = 0.1
inhibition_method = global
perm_inc = 0.1
perm_dec = 0.1
potential_pct = 0.75
boost_strength = 0 (disabled)

3.7.2 Temporal Memory

Again to benchmark the performance of the Temporal Memory layer, we crafted random inputs with 10% density, then feed the random inputs into the TM layer, measuring the average execution time. The results are shown in Figure 14

The Temporal Memories are bencharked under the following parameter.
initial_perm = 0.21
perm_inc = 0.1
perm_dec = 0.1
connected_perm = 0.1
num_segments_per_cell = 1

max_synapses_per_segment = 1024
max_new_synapses = max_synapses_per_cell
predicted_segment_decay = 0
cells_per_column = 16

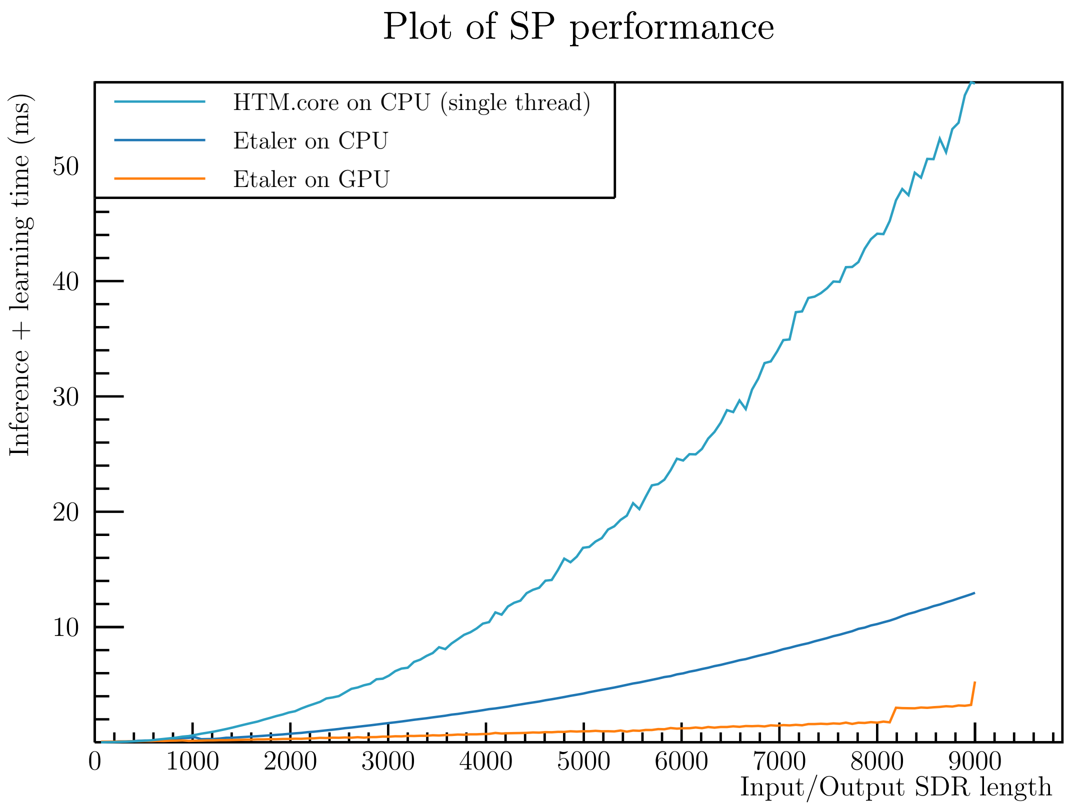

Figure 13: Average time for Etaler and HTM.core to run a SP layer under different input sizes.

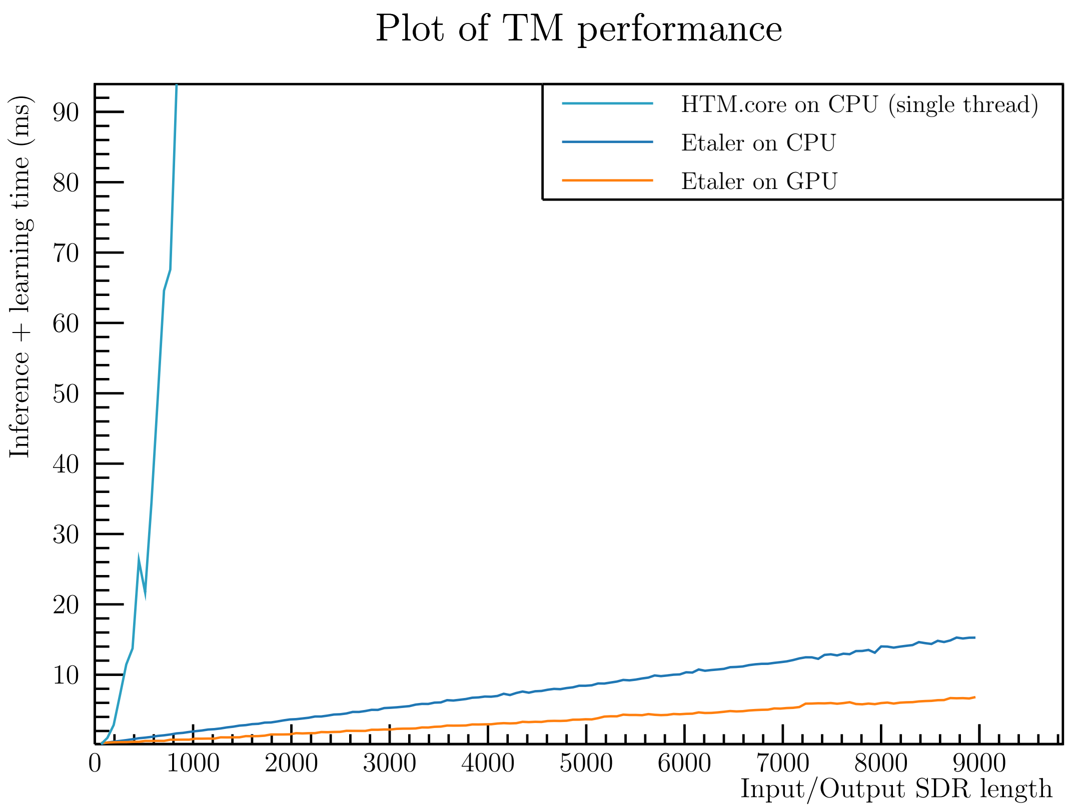

Figure 14: Average time for Etaler and HTM.core to run a TM layer under different input sizes.

3.8 The performance characteristic of HTM

We gained access to a far more powerful workstation months after the entire project concluded (before finishing the report). The new work station run a AMD Threadripper 3970X processor with 32 Zen2 cores and 128MB of L3 cache (comparing to 8 Zen1 cores and 16MB of L3 cache); both with a dual channel DDR4 memory at 2600MT/s. Due to it is after the project concluded; we are unable to perform proper benchmark like we've done above. So we've selectively ran benchmarks of a few sizes and made the result into Table 2; measured in milliseconds.

time(ms)/SP size	256	512	1024	2048	9000
R7 1700X	0.0760	0.1519	0.3817	0.9792	13.0239
TR 3970X	0.0689	0.1131	0.2311	0.5244	10.6992

Table 2: Performance of the SpatialPooler on R7 1700X versus TR 3970X.

As the table shown, the Spatial Pooling algorithm does not scale well against the number of available cores and the amount of cache. Thus we hypothesize that the algorithm is bounded by the memory bandwidth. And the hypothesis is supported by the fact that the Spatial Pooling algorithm is constantly reading data for the synapses and only the SDR is reused. We didn't conduct deep analysis of the subject but we expect further analysis to support our hypothesis.

Table 3 shows how well the algorithm scales with Simultaneous Multi Threading feature on the R7 1700X processor. In sort, the algorithm barely benefits from SMT even though SMT on the processor doubles the available thread count. This again aligns with our hypothesis that algorithm is very memory bounded.

time(ms)/SP size	256	512	1024	2048	9000
SMT on	0.0760	0.1519	0.3817	0.9792	13.0239
SMT off	0.0782	0.1611	0.3894	0.9956	13.1125

Table 3: Performance of the SpatialPooler on R7 1700X with and without SMT.

4 Optimizations

We implemented some non-travel optimizations for the library. Not all of them pays out.

4.1 Things that works

4.1.1 Loading SDR into OpenCL local memory

GPU tends to have high bandwidth but high latency memories and they only have little cache to work with. But the memory access pattern when performing the inference or learning process in HTM is far from ideal(linear). Causing lots of cache invalidation and reading/writing to DRAM, harming the performance. We discovered that in most cases the local memory can hold the entire SDR to perform the task. By doing so we can decrease not only the non-linearity of memory access but also the over access count.

4.1.2 half-precision on OpenCL

Since HTM is bottle-necked by memory bandwidth and being very noise tolerance, theoretically, using smaller yet less-accurate data types to store synaptic permanence should increase run-time performance while not affecting the model's prediction capability.

Thus we added support to run HTM in half-precision mode when the OpenCL extension *cl_khr_fp16* [16] is detected. This leads to up to 40% inference performance increase on a Intel HD 620 iGPU. However using half-precision floating point values is a anti-optimization on a x86 CPU. We'll explain why in the next section.

4.1.3 Use counting sort to select inhibition threshold on OpenCL

The global inhibition function requires us to find the top-K value of the cell activity. The top-k selection problem is well studied for large cases. But HTM operates in a entirely different paradigm, where we are dealing with at most selecting the 100th value out of an array of 10000 values.

The standard GPU solution of partial sort and merge method is inefficient in such case. After experiments, we found the best performing method is to use the local memory as a scratch pad and run counting sort on one single processing element.

4.1.4 Ignore checking for null synapses when possible

Since we store synapses in two tensors. One for storing the index and the other for the permanence. And the Temporal Memory algorithm grows synapses as it learns. We must check weather a given synapses is connected to a cell before checking if the permanence is great enough to make a connection in the cell activation process. But such check is not required for a Spatial Pooler. As they have all their synapses pre-grown and won't grow/decay synapses. This presents a chance for optimization for the Spatial Pooler. Thus we made it that under OpenCL, we ignore checking if the synapse is connecting to nothing when working in a Spatial Pooler. This turns out to provide a 30% performance increase.

4.2 Things that kinda works

This section includes optimizations that makes no performance increase. But at reduces memory usage or what not. Or works some times.

4.2.1 IEEE 754 half precision floating point on Aarch64 processor

Since all standard Aarch64 processor supports processing IEEE 754 half precision floating point numbers natively. We thought using 16 bit floating point numbers on Aarch64 processors could greatly improved the amount of synaptic permanence to be cached. But testing results shows otherwise. Using 16 bit floating point numbers doesn't improve the inference speed.

4.2.2 Using smaller data types for synapse index

It is possible to use a smaller data type (ex: int16) to store the synapse connections target when the layer's input is small enough. By using smaller data types, we hope to decrease the amount of memory bandwidth needed to run HTM; thereby increasing speed. But experiments show no performance increase on both CPU and OpenCL.

4.2.3 Compressing SDRs for lower local memory usage

Since SDRs are sparse binary arrays. The default method of storing Boolean values by storing them as a 1 byte value is far from efficient. By bit-shifting and and/or-ing values, we can have a guaranteed 8:1 compression. Allowing the kernel to use less local memory. Thereby the GPU should be able to schedule more work-groups on each compute unit. Though more work will have to be done for each work-item, the benefit of better latency hiding and caching should theoretically out gain the cons. But after experiments we find that even at the maximum capacity of the GPU's local memory, we are not gaining any performance by doing so. In fact we loose around 20% of performance in this case. Thus compression is only used when the SDR can't fit in local memory without compression.

4.3 Things we tried that doesn't work

This section shows the optimizations we tried and found it merely is a backward step or the performance/memory trade-off isn't worth it.

4.3.1 O(1) space complexity cell activity on GPU by presorting synapses

One optimization we tried is to sort the synapses and the respecting synaptic permanence on the GPU so the activity forwarding process doesn't have to store a lookup table in the global memory. Since the synapses are stored as a ND array, where it's 0 N-2th dimension (in column major order) stores all the synapses for each cell and the N-1th dimension stores the synapse itself. We thought we could make efficient use of the GPU by performing merge-sort[17] on each OpenCL work-group for each cell. Which turns out to be a very, very bad idea. Due to merge-sort causes O(n $\log n$) global memory accesses (comparing to the O(n) accesses using a lookup table) and the inefficient use of work-items towards the end of the sorting. The performance dropped by a factor of 10 on a NVIDIA RTX 2080Ti GPU.

4.3.2 IEEE 754 half precision floating point on x86 processor

Another optimization idea was to use the IEEE 754 16 bit floating point number [18] instead of using the standard IEEE 754 32bit floating point numbers [18] to represent the neuron synapse permanence. The idea was that HTM should be noise tolerant enough that we can using half precision numbers and get basically the same results, while using half the memory and have better cache hit rate. And hopefully the performance benefits can out-run the cost of converting half-precision into single-precision values in software (since x86 processors doesn't support conversion of halfs to floats in hardware).

But benchmarking shows otherwise. It turns out using half precision numbers causes 50% of performance loss.

4.3.3 xorshift as RNG with a single seed across work items in OpenCL

Initially the *reverseBurst* function in OpenCL receives a seed *s*, then we use *s+global_id* as the seed for xorshift[19] to generate random number in the OpenCL kernel. But this turns out to not being random enough and causes the Temporal Memory algorithm to learn poorly. We fixed this by having a huge array of pre-generated, high quality random numbers and seed xorshift using the array. (Instead implementing a proper but slow RNG on the GPU)

4.3.4 POCL's CUDA backend

It is a well known fact that NVIDIA's OpenCL implementation is outdated, buggy and full of weird issues when stuff goes wrong. POCL[20] is a open source, performance portable OpenCL implementation. Which, includes a CUDA/NVIDIA PTX backend. It looks like a good replacement for NVIDIA's OpenCL library and we hope it comes with the missing *cl_khr_fp16* extension. But that turns out to not be the case. The code POCL generates is very slow and it does not support the *cl_khr_fp16* extension.

4.4 Possible Optimizations

Despite us have tried as much optimization strategies as we can. There are a lot of optimizations we either can't try due to the lack of equipment or the lack of time.

4.4.1 Processors with two independent memory subsystems

Devices like Intel's Xeon Phi(Knight's Ferry/Landing/Corner/etc..) family, Xilinx's U280 FPGA accelerator card and Intel's Stratix 10 MX FPGA development kit allows pragmatically allocating memory on different memory subsystems while being able to access data from any region without constants. This opens up the opportunity to store synapses and permanences on different memory subsystems. At least doubling the available bandwidth thus a theoretical 2x performance improvement. (Not counting the better DRAM access pattern.)

4.4.2 Pre-loading synapses on local memory when SDR can't fit on local memory

We are pretty uncertain weather if this will work. Currently we try to pre-load SDR from global memory to local memory for faster access. When this is not viable, we instead operate entirely on global memory. Local memory is not used in this case. Instead of wasting the memory, we can load synapses in small chunks onto local memory then perform our usual computation from there. This should give up better access patters in the cost of more instructions being executed and worse GPU thread latency hiding due to increase local memory usage.

4.4.3 Using dual-port BRAM on FPGAs to store SDR

As described above. We can store SDRs into local memory to reduce memory access. The same could be implemented on FPGAs (either using Verilog/VHDL or by HLS/OpenCL). Modern FPGAs support dual-port BRAM cells; allowing 2 accesses per clock cycle (versus the typical 3-cycle access for local memory on a GPU). This can be used to greatly speed up HTM calculations comparing to a GPU. GPU cores stall a lot on accessing it's local memory.

4.4.4 Using MSMC global buffer to store SDR on TI DSP

Again this is a hardware specific optimization. Texas Instruments' DSP processors support a large, shared, on-chip, very fast (faster than local memory), large (enough to hold SDR for any partial application) memory referred as the MSMC. The logic behind this proposal is the same as using BRAM on FPGA for better processor utilization. Furthermore, in this case the local memory on the DSP can be freed up for other purpose like caching synapse/permanence values; improving the access pattern.

4.4.5 Efficitive ILP under VLIW

Assuming the RAM (or using MSMC described above) is fast enough to feed the DSP's VLIW core. A VLIW pipeline can theoretically efficiently calculate the overlap score. Figure 15 show the optimal sequences of packets in a 3 way VLIW processor - 3 execution units and the first 2 of them can act as a load store unit. [4].

In an ideal case - 1 cycle memory access and no branch penalty, we can evaluate 1 synapse every 5 cycles.[5] In a more realistic case - 50 cycle DRAM latency, 5 cycles L2 cache latency, 95% L2 hits, 1 cycle L1 latency, 50% L1 hits and the SDR lives on MSMC with 2 cycle latency, 99% branch accuracy and a penalty of 5 cycles. We can calculate on average it takes 12.5 cycles per evaluation.[6]

Figure 16 shows it is possible to make a synaptic evaluation taking only 4 VLIW packets given some additional features. a. We have 4 execution units and b. loading from r9[-1] returns 0 (there's extra memory before the array storing the SDR, initialized to zero). But due to data dependency, we find it impossible to make evaluation a 3 packet process even with a infinite wide VLIW processor. The only solution making synaptic evaluation faster after this point besides introducing custom hardware is to go SIMT style parallelism.

[4]Basically VLIW-ifed ARM/RISC-V assembly

[5]Maybe VLIW isn't that bad after all

[6]Calculated using hand-wavy math. No simulation is used and no careful analysis is done.

```
# r4 holds the offset of the current calculations
n:      ldr r0, r5[r4]          # r5 stores where the synapses are
        ldr r1, r6[r4]          # r6 stores the permence
        cmp r10, r4, n_synapse-2# weather we should end the loop
                                # in the next iteration

n+1:    cmp r3, r0, -1          # -1 indicates a invalid link
        lt r1, r1, min_perm     # is r1 lesser then min_perm
        add r4, r4, 1           # increment loop counter


n+2:    bt r3, next_cell        # if we have a invalid link
                                # we are at the end of synapses
        bt r1, n                # if permence is too low, we
                                # go to the next synapse
        mov r7, r11             # should we end the loop?

n+4:    ldr r1, r9[r3]          # r9 stores where the SDR is
        mov r11, r10            # turn "we should end the next time"
                                # to "we should end now"

n+5:    add r8, r8, r1          # r8 is activation counter
        bt r7, next_cell        # loop ends
        bf r7, n
```

Figure 15: Optimal algorithm to evaluate HTM cell activation in a 3 wide VLIW

```
n:      ldr r0, r5[r4]          # r5 stores where the synapses are
        ldr r1, r6[r4]          # r6 stores the permence
        cmp r10, r4, n_synapse-2# weather we should end the loop
                                # in the next iteration
        add r4, r4, 1           # increment loop counter

n+1:    cmp r3, r0, -1          # -1 indicates a invalid link
        lt r1, r1, min_perm     # is r1 lesser then min_perm
        mov r7, r11             # should we end the loop?


n+2:    bt r3, next_cell        # if we have a invalid link
                                # we are at the end of synapses
        bt r1, n                # if permence is too low, we
                                # go to the next synapse
        mov r11, r10            # turn "we should end the next time"
                                # to "we should end now"
        ldr r12, r9[r3]         # r9 stores where the SDR is

n+4:    add r8, r8, r12         # r8 is activation counter
        bt r7, next_cell        # loop ends
        bf r7, n
```

Figure 16: Optimal algorithm to evaluate HTM cell activation in a 4 or above wide VLIW and negative indices results in a 0

4.4.6 OpenCL 2.0 SVM for fast data migration

Currently for Etaler to access data on the GPU/sending data to GPU requires the CPU actively coping all data and sending/reviving them via PCIe. OpenCL 2.0's SVM (shared virtual memory) feature can mitigate the need of a CPU copying everything and even allowing GPUs sending data from one to another without touching the main memory.

4.4.7 Lazy evaluation

Currently we employ eager evaluation for the tensor system. Every operation is evaluated as soon as they are called. Lazy evaluation does things the opposite way; only evaluating operations when we absolutely need them. Opening the chance for merging operations and further optimizations.

4.4.8 Mirrored Synapses

Sungur implements mirrored synapses in his thesis[6]. Unlike our approach of storing synapses each cell has, his method additionally stores which cells the current cells is connecting to. Using at minimal 2 times more memory in exchange for less operation when inference. This method also makes managing synapse and wrights more difficult. As now whenever a synapse is modified, we also need to update the mirrored one. Among other complexity revolving memory management.

We decide it is not worth the trouble to have it in Etaler. Especially the amount of memory management needed is far too complex for OpenCL. (We are not suggesting it can't be done.) But the resulting performance increase looks promising and is worth further exploration.

5 Agent Design and Evaluation

5.1 Neural Architecture

The network architecture we used, shown in Figure 17, is a modified version of what is used in Sungur's thesis[5]. [7]

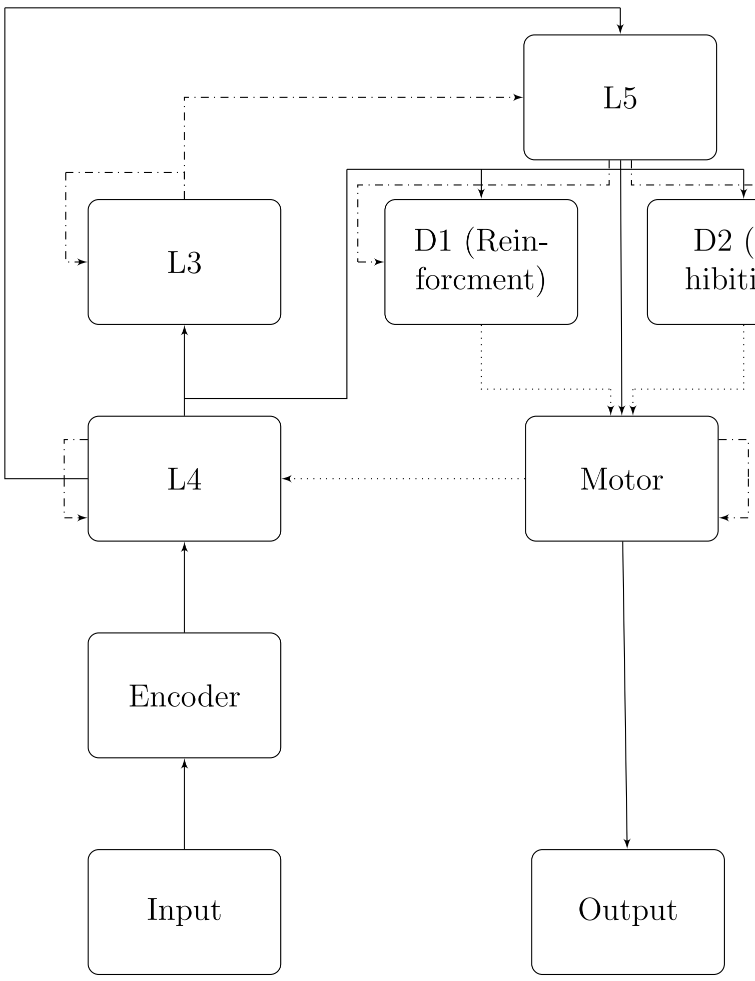

Figure 17: The architecture of our agent

Solid lines in Figure 17 denotes distal synapses, the dash lines denotes proximal synapses and the dotted lines denote apical projection. [8]

The only change we did was to remove the L2 layer and use a different learning strategy. We find asking a Temporal Memory layer predicting another TM layer's output (both without external modulation) makes not much sense. In a more formal expression, a TM layer predicts x_{t+1} based on x_t. Stacking two TMs together is predicting x_{t+2} from x_t in the best case or degrading the performance (since learning isn't perfect) in the worst case. And while using TD-error like previous works for learning is the most reasonable approach. The TD infused Temporal Memory is an experimental implementation and not documented in any way. So instead apply positive learning when met with positive reward. Vise versa. (The architecture either performs reinforcement or inhibit the neurons.)

[7]Please forgive my poor still of building $\LaTeX$graphs.

[8]We can't find a reasonable way to add a legend to a graph.

The functionality of each block is as follows: The encoder is used to convert the environment observations into SDRs that HTM can handle. We uses two pairs of 2 dimensional Grid Cell encoders in our agent. There's no particular reason doing so but Grid Cell based encoders generally outperform the basic Scalar and Random Distributed Scalar encoders in our experience. L4 attempts to predict the next state of the agent. L3 generates a high level representation. L5 integrates information from L3 and L4 to generate possible actions. D1/D2 infuses information for L4 and L5 to generate "good" and "bad" ideas that the agent should or shouldn't take. Then the motor reason combines L5, D1 and D2 to generate motor commands.

5.2 Agent and Evaluation

To bring HTM out of theoretical use cases. We decide to evaluate our learning agent in a standard Reinforcement Learning environment. Namely OpenAI Gym[21] and it's CartPole-v1 environment. Then we record the average reward given to the agent.

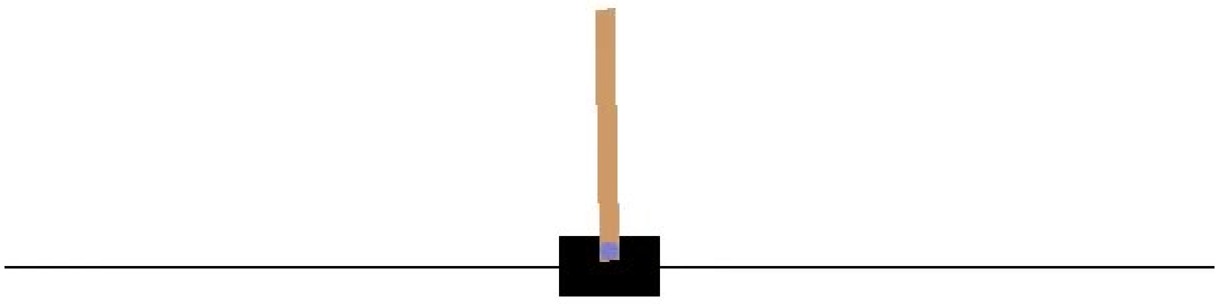

Figure 18: The CartPole-v1 environment

As the name indicated. CartPole-v1 is composed by a cart on a 1 dimension line and a pole which an end is loosely attached to the cart. The cart can move along the line and the pole will be pulled and rolls along. When the environment is initialized, the cart is set to be at the center of the screen and the pole set to an random (tho small) angle. Then the agent is rewarded whenever the pole doesn't fall over and the cart is kept onscreen. When the condition failed, the agent is punished and the environment resets. Thereby tasking the agent to carefully balance the pole and not doing so using a large movement. From the agent's point of view, the environment provides the agent with 4 variables. Namely the pole's angle, the angular velocity of the pole, the speed of the cart and the location of the cart.

6 Results

Figure 19 shows a plot of the reward the agent received over time. We discovered sometime in the development process that random action is a somewhat good solution for this environment. So we decide to include a non-learning agent to simulate the default behavior of out agent as a reference.

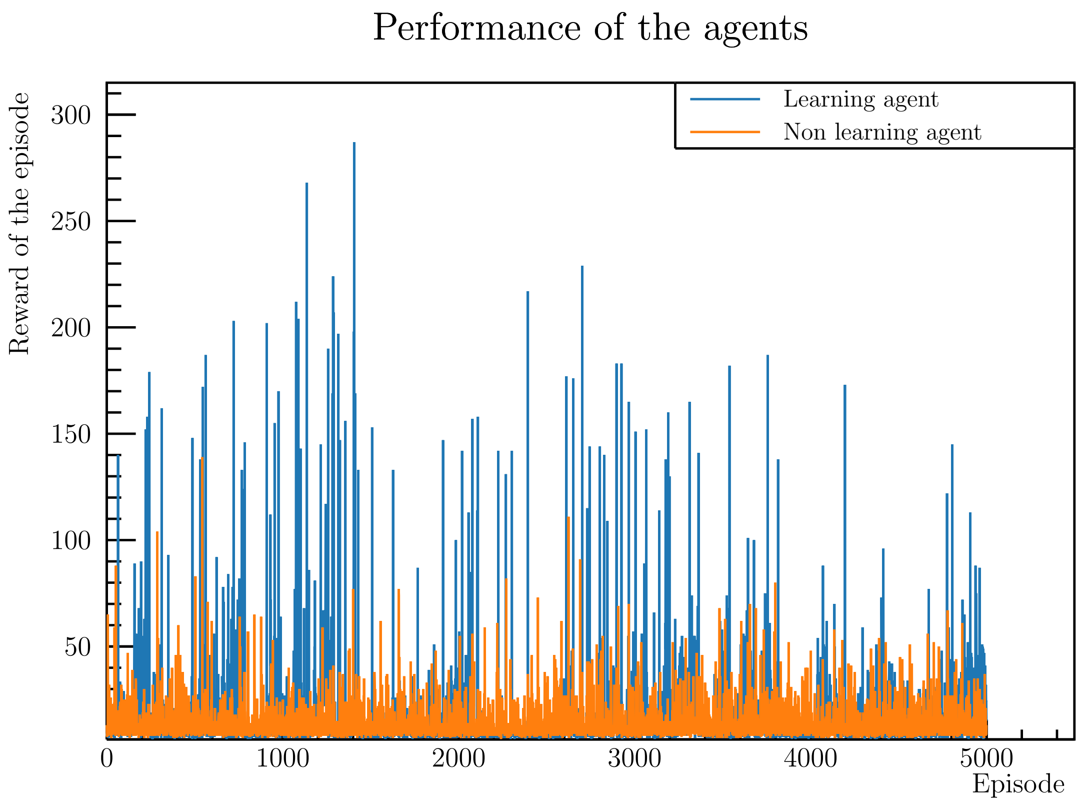

Figure 19: The performance of our agents

We can see the learning agent does perform a lot better than the non-learning one. But despite that, the learning agent is unstable and cannot balance the pole for a prolong period of time nor doing it constantly. As of the time of writing this report, we have no evidence to point to any particular reason causing this. But we think it is caused by not using TD Error for learning thus the agent is just evaluating the expected reward of the intimidate next time step. Leading to the agent not capable of seeing further consequence of it's actions. we also notices in Figure 20 that the learning agent although performing better overall. It performs worse than the non learning one more often. We are not sure why this occurs but our best guess is that it is related to the previously described issue.

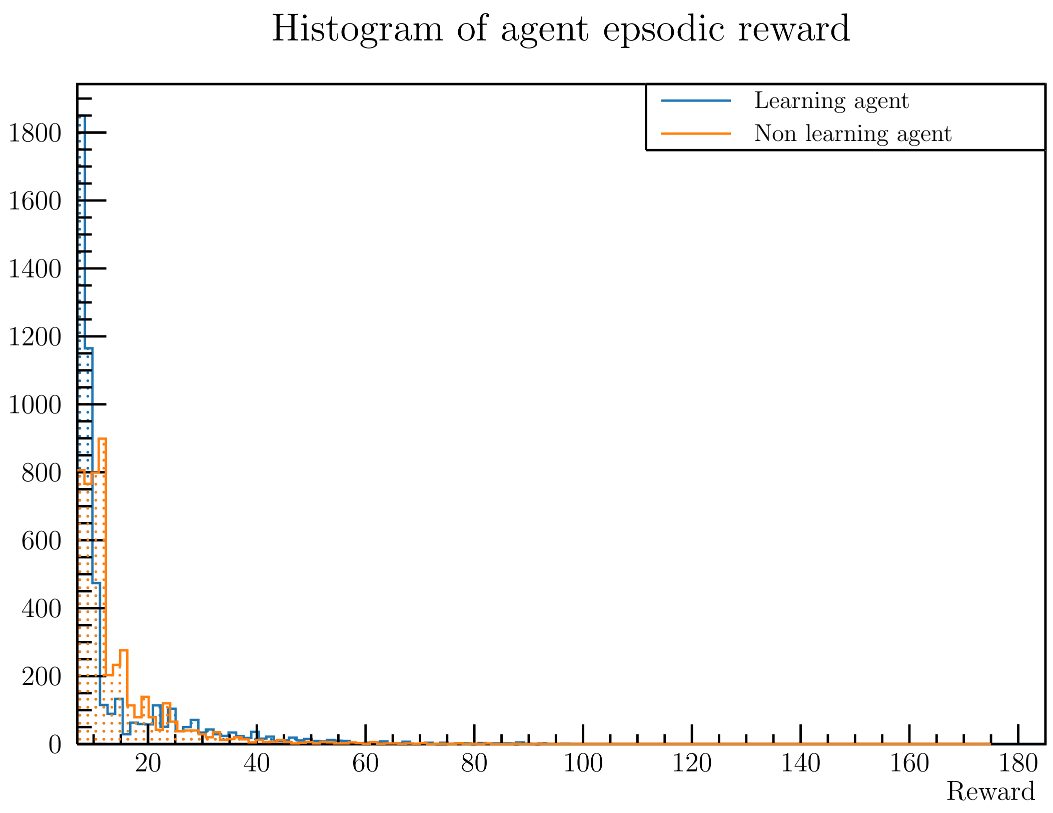

Figure 20: The histogram of rewards. Zoomed into 0 100.

7 Discussion

The discussion will be slipped into several parts. Mostly due to this report covers several directions of HTM implementation and reinforcement learning.

7.1 Efficient HTM Implementation

As our work shows. Running HTM on any modern computer slow and mainly bottled-necked by how fast can the processor access memory. Further more, hand wavy calculation shows even HBM will be saturated under the worst (or best, depending on how you see things) real world situation - a 64 compute unit, 4 cycle per instruction 16-wide SIMD processor.[9] Thus, instead of trying to execute more operations per cycle. We should aim for getting more effective bandwidth out of memory.

That is. From a software perspective, how could we not go through the processor's caches for one-off data, how to keep reusable data in cache and how to keep the access pattern linear. And from a hardware/ASIC point of view, how could the DRAM bandwidth be maximized. Thus we suggest to have a NoC where each node is a simple processor with their own memory controller attached in order to maximize the overall bandwidth. Then the issue will be about how to manage the NoC traffic instead of not having enough DRAM bandwidth. (For all those paper out there implementing HTM on FPGAs using on-chip memory. **No, no, no.** Using the on-chip memory is not a solution. You get single cycle access. But there's only 54MB of them in a state-of-the-art FPGA. That's far from enough for any possible use cases.)

7.2 The possibility of intelligent biological-like system

Though the definition of "intelligence" is vague even in the ML Community. It should be reasonable to state that the agent in our work is not an intelligent system. However we hope we have demonstrated that building a agent in HTM is doable and can be expended upon. In the project we apply an brain-like circuit to the agent. While we also expect this is not the only working configuration. Like how brains of different species works differently. We expect varied combinations of layers reacting differently but still showing traits of a learning system.

7.3 Open Questions

7.3.1 The credit assignment problem and generalization

Out agent architecture relies on the D1 and D2 region to learn what's good and bad behavior given the agent's past and current states. Which can be analog to what Q and Deep Q learning does. But this strategy learns very slowly and does not generalize well. It can't distinguish between the positive and negative cause of the reward. Thus for it to learn the outcome of all possible states and actions, the agent must have tried all possible combination in the search space. Which is very inefficient. We hope the Thousands Brains Theory[22] can provide a solution.

[9] Yes, I'm talking about the AMD GFX9 chips. aka Radeon Vega64

7.3.2 Better handling of TD Error

Based on what we know about the TD Error based Temporal Memory algorithm.[10] The algorithm can only handle a TE Error in the range of $[-1, 1]$. Thus it made tacking long term rewards more difficult. For example, an error of 0.01 over 1000 time steps accumulates to an error of 10. Which is not traceable by the current algorithm.

7.3.3 The relation to Q learning

Given there's not a mathematical framework to reason about HTM and our agent's objective function is vague at bestZ4 . It is quite difficult to pin point what our agent is trying to do nor why it does it. Still, we think our agent can be compared to Q learning where the sum of D1, D2 and it's interaction with the motor region is the Q table. If such proof can be provided, we believe it can be a key to improving the agent's performance.

[10]That is to say - not much. There's no documentation of the algorithm nor we used it in our project. Our information comes from reading the source code.

8 Conclusion

The project started out aiming to experiment HTM in a standard Reinforcement Learning environment. To achieve our original goal, we came up with an efficient HTM implementation, explored various optimizations and purposed unexplored optimizations. Showing current HTM implementations like NuPIC can be vastly optimized. On the other hand, our experiment of applying HTM in standard Reinforcement Learning environment shows that HTM is a valid option for such tasks.

8.1 Future research

I certainly want HTM to success. I'll keep maintaining Etaler and see how much further the performance can be pushed. The memory bandwidth barrier is a hard one to break. We also have plan to develop a HTM accelerator for FPGA. No one knows how far we can take that; we still run into memory bandwidth issues. But I optimistically hope the project can further accelerate HTM's development.

I also want to spend more time to investigate computational neural science and the old HTM-Zeta1[23] model. As lacking a proper hierarchy is a daunting problem in HTM (people have ideas on how they are structured and what they do, but don't know how is it actually done). I hope further work can provide a solution to the current situation.

Finally, as usual, I'll be an active member of the HTM community. Joining discussions and sharing my finding with everyone.

Acknowledgment

The author would like to thank all contributors whom contributed to the Etaler project. The contributions are important parts of the project.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484 EP –, Jan 2016. Article.

[2] Y. Jin, J. Zhang, M. Li, Y. Tian, H. Zhu, and Z. Fang, "Towards the automatic anime characters creation with generative adversarial networks," 2017.

[3] J. Hawkins, *On Intelligence: How a New Understanding of the Brain Will Lead to the Creation of Truly Intelligent Machines.* Times Books, oct 2004.

[4] M. Otahal, *Architecture of Autonomous Agent Based on Cortical Learning.* PhD thesis, 12 2013.

[5] A. K. Sungur and E. Surer, "Voluntary behavior on cortical learning algorithm based agents," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–7, Sep. 2016.

[6] A. K. Sungur, *HIERARCHICAL TEMPORAL MEMORY BASED AUTONOMOUS AGENTFOR PARTIALLY OBSERVABLE VIDEO GAME ENVIRONMENTS.* PhD thesis, 8 2017.

[7] J. Heyder, *Hierarchical Temporal Memory Software Agent : In the light of general artificial intelligence criteria.* PhD thesis, 2018.

[8] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin, "Biological and machine intelligence (bami)." Initial online release 0.4, 2016.

[9] G. N. Elston, "Cortex, Cognition and the Cell: New Insights into the Pyramidal Neuron and Prefrontal Function," *Cerebral Cortex*, vol. 13, pp. 1124–1138, 11 2003.

[10] D. Debanne, B. H. Gähwiler, and S. M. Thompson, "Asynchronous pre- and postsynaptic activity induces associative long-term depression in area ca1 of the rat hippocampus in vitro," Feb 1994.

[11] Y. Cui, S. Ahmad, and J. Hawkins, "The htm spatial pooler—a neocortical algorithm for online sparse distributed coding," *Frontiers in Computational Neuroscience*, vol. 11, p. 111, 2017.

[12] Numenta, "The official implementation of Hierarchical Temporal Memory." `https://github.com/numenta/nupic`, 2019.

[13] M. Otahal, D. Keeney, D. McDougall, *et al.*, "HTM.core implementation of Hierarchical Temporal Memory." `https://github.com/htm-community/htm.core/`, 2019.

[14] A. P. Chang, "Etaler implementation of Hierarchical Temporal Memory." `https://github.com/etaler/Etaler`, 2019.

[15] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[16] "Opencl 1.2 reference pages."

[17] D. E. Knuth, *The Art of Computer Programming: Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, may 1998.

[18] *IEEE standard for binary floating-point arithmetic*. New York: Institute of Electrical and Electronics Engineers, 1985. Note: Standard 754–1985.

[19] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software, Articles*, vol. 8, no. 14, pp. 1–6, 2003.

[20] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, pp. 752–785, Oct 2015.

[21] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[22] J. Hawkins, M. Lewis, M. Klukas, S. Purdy, and S. Ahmad, "A framework for intelligence and cortical function based on grid cells in the neocortex," *Frontiers in Neural Circuits*, vol. 12, p. 121, 2019.

[23] J. Hawkins and D. George, "Hierarchical temporal memory concepts , theory , and terminology," 2006.