Etaler

Jan 07, 2020

INSTALLATION

Etaler in an high-performance implementation of Numenta's HTM algorithms in C++. It is diesigned to be used in real world applications and research projects.

Etaler provides:

- HTM algorithms with modern API
- A minimal cross-platform (CPU, GPU, etc..) Tensor implementation

Etaler requires a modern C++ compiler supporting C++17. The following C++ compilers are supported

- On Windows, Visual C++2019 or better
- On Unix systems, GCC 8.2 or a recent version of clang

CHAPTER 1

Licensing

Etaler is licensed under BSD 3-Clause License. So use it freely!

Be aware that Numenta holds the rights to HTM related patents. And only allows free (as "free beers" free) use of their patents for non-commercial purpose. If you are using Etaler commercially; please contact Numenta for licensing. (tl;dr Etaler is free for any purpose. But HTM is not for commercial use.)

1.1 Building with Visual Studio 2019

This is a step by step instruction detailing a proven way to build Etaler with Visual Studio 2019.

1.1.1 Install dependencies

- Install CMake
- Install Git Bash
- Using a Git Bash terminal, perform the following commands:

```
cd Etaler/3rdparty
git clone https://github.com/imneme/pcg-cpp.git
git clone https://github.com/USCiLab/cereal.git
```

- Download the latest Intel TBB zip file.
- Extract it into `C:\Program Files\Intel\TBB`
- Update PATH to include `C:\Program Files\Intel\TBB\bin\intel64\vc14`
- Setup `TBBROOT` environment variable to point to `C:\Program Files\Intel\TBB`

To compile the tests you need to:

- Download standalone catch.hpp into the `Etaler/tests` directory.

If you intend to use Etaler with a GPU:

- Install an OpenCL SDK appropriate for your GPU (e.g. NVidia CUDA SDK).
- Download Khronos cl.hpp into `Etaler/3rdparty/CL` directory.

1.1.2 CMake

The CMake application is used to generate the project and solution files for MSVC.

Set the "Where is the source code:" text box to point to the root of your cloned Etaler repository. And set the "Where to build the binaries:" text box to point to a `build` subdirectory off of your Etaler repositories.

For example:

- Source: `C:/Github/Etaler`
- Binaries: `C:/Github/Etaler/build`

Press the `Configure` button and choose the "Visual Studio 16 2019" generator. Finally press the `Generate` button.

Visual Studio 2019 can then be used to load the `Etaler/build/Project.sln` solution file, and build the library, tests and examples.

By default the solution builds a **dynamic** linked Etaler library. Therefore, the `Etaler/build/Etaler/Release` (or `Debug`) directory needs to be in your `PATH`. Or the `Configuration Properties -> Debugging -> Environment` entry can use "PATH=%PATH%;C:/Github/Etaler/build/Etaler/Release".

1.2 Building on OS X

This is a step by step instruction detailing on a proven way to build Etaler on OS X

1.2.1 Install dependencies

```
brew install tbb cereal cmake
```

1.2.2 Install catch2

If you want to runs tests

```
git clone https://github.com/catchorg/catch2
cd catch2
mkdir build && cd build
cmake ..
make -j4
sudo make install
```

1.2.3 Get the OpenCL C++ wrapper

If you want OpenCL support. Download and install the offical OpenCL C++ wrapper

```
sudo wget https://www.khronos.org/registry/OpenCL/api/2.1/cl.hpp -P /System/Library/
↪Frameworks/OpenCL.framework/Headers/
```

1.2.4 Build Etaler

```
git clone https://github.com/Etaler/Etaler --recursive
cd Etaler
mkdir build && cd build
cmake ..
make -j4
```

Please add `-DETALER_BUILD_TESTS=off` to cmake if you don't want to build the tests or don't have catch2 installed. Add `-DETALER_ENABLE_OPENCL=on` for OpenCL support.

1.2.5 Older OS X sysetms

If you are running OS X < 10.14. Please install `gcc` from homebrew and add `-DCMAKE_CXX_COMPILER=gcc-9` (Or whatever the GCC version you installed) to build Etaler. Apple doesn't ship the full C++17 features Etaler needs on the older systems.

1.3 Building on Linux

Building Etaler on Linux should be easy as it is mainly developed on Linux.

1.3.1 Using Docker

Etaler's repo ships with a Docerfile in the `.devcontainer` directory. You can copy the file into the docker folder and utilize docker for an easy build.

```
cd Etaler/docker
cp ../.devcontainer/Dockerfile .
# Build the library
docker -D build   --tag etaler:latest .
# Run the container
docker run --rm -it -e DISPLAY=:0 --cap-add=SYS_PTRACE --mount source=etaler-volume,
↪target=/home etaler:latest
```

1.3.2 Building locally

If you are like me - want to use the library locally on the system and/or want to deploy it to an embedded system, Docker may not be an option for you. No worries, building locally is also very easy.

Here I show how to setup your system. You'll need to adapt the code if you are not using Arch Linux.

Installing dependency

```
sudo pacman -S gcc cmake catch2 cereal intel-tbb opencl-headers
```

Clone and build

```
git clone https://github.com/Etaler/Etaler --recursive
cd Etaler
mkdir build && cd build
cmake ..
make -j4
```

With a lack of writing ablilty. The introduction will be a clone of PyTorch's introduction document. This is also a good chance to see how well Etaler is doing at it's tensor operators.

1.4 A quick introduction to Etaler

At Etaler's core are Tensors. They are generalized matrix in more then 2 dimentions; or called N-Dimentional Arrays in some places. We'll see how they are used in-depth later. Now, let's look what we could do with tensors

```
// Etaler.hpp packs most of the core headers together
#include <Etaler/Etaler.hpp>
#include <Etaler/Algorithms/SpatialPooler.hpp>
#include <Etaler/Encoders/Scalar.hpp>
using namespace et;

#include <iostream>
using namespace std;
```

1.4.1 Creating Tensors

Tensors can be created from pointers holding the data and an appropriate shape.

```
int d[] = {1, 2, 3, 4, 5, 6, 7, 8};
Tensor v = Tensor({8}, d);
cout << v << endl;

// Create a matrix
Tensor m = Tensor({2, 4}, d);
cout << m << endl;

// Create a 2x2x2 tensor
Tensor t = Tensor({2, 2, 2}, d);
cout << t << endl;
```

Out

```
{ 1, 2, 3, 4, 5, 6, 7, 8}
{{ 1, 2, 3, 4},
 { 5, 6, 7, 8}}
{{{ 1, 2},
  { 3, 4}},

 {{ 5, 6},
  { 7, 8}}}
```

Just like a vector is a list of scalars, a matrix is a list of vectors. A 3D Tensor is a list of matrices. Think it like so: indexing into a 3D Tensor gives you a matrix, indexing into a matrix gives you a vector and indexing. into a vector gives you a scalar,

Just for clearifcation. When I say "Tensor" I ment a `et::Tensor` object.

```
// Index into v wii give you a scalar
cout << v[{0}] << endl;

// Scalars are special as you can convert them into native C++ types
cout << v[{0}].item<int>() << endl;

// Indexing a matrix gives you a vector
cout << m[{0}] << endl;

// And indexing into a 3d Tensor to get a matrix
cout << t[{0}] << endl;
```

Out

```
{ 1}
1
{ 1, 2, 3, 4}
{{ 1, 2},
 { 3, 4}}
```

You can also create tensors of other types. The default, as you can see, is whatever the pointer is point to. To create floating-point Tensors, just point to an floating point array then call `Tensor()`.

You can also create a tensor of zeros with an supplied shape using `zeros()`

```
Tensor x = zeros({3, 4, 5}, DType::Float);
cout << x << endl;
```

Out

```
{{{ 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0}},

 {{ 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0}},

 {{ 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0},
  { 0, 0, 0, 0, 0}}}
```

1.4.2 Operation with Tensors

You can operate on tensors in the ways you would expect.

```
Tensor x = ones({3});
Tensor y = constant({3}, 7);
```

(continues on next page)

(continued from previous page)

```
Tensor z = x + y;
cout << z << endl;
```

Out

```
{ 8, 8, 8}
```

One helpful operation that we will make use of later is concatenation.

```
// By default, the cat/concat/concatenate function works on the 1st axis
Tensor x_1 = zeros({2, 5});
Tensor y_1 = zeros({3, 5});
Tensor z_1 = cat({x_1, y_1});
cout << z_1 << endl;

// Concatenate columns:
Tensor x_2 = zeros({2, 3});
Tensor y_2 = zeros({2, 5});
Tensor z_2 = cat({x_2, y_2}, 1);
cout << z_2 << endl;
```

Out

```
{{ 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0}}
{{ 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0}}
```

1.4.3 Reshaping Tensors

Use the `reshape` method to reshape a tensor. Unlike PyTorch using `view()` to reshapce tensots. view in Etaler works like the one in xtensor; it performs indexing.

```
Tensor x = zeros({2, 3, 4});
cout << x << endl;
cout << x.reshape({2, 12}) << endl;  // Reshape to 2 rows, 12 columns
```

Out

```
{{{ 0, 0, 0, 0},
  { 0, 0, 0, 0},
  { 0, 0, 0, 0}},

 {{ 0, 0, 0, 0},
  { 0, 0, 0, 0},
  { 0, 0, 0, 0}}}
{{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
```

1.4.4 HTM Algorithms

HTM algorithms, like what the name indicates. Implements HTM related algorithms. They are the sole perpose why Etaler exists.

Without getting too deep. Typically the first thing we do in HTM is encode values into SDRs. SDRs are sparse binary tensors. i.e. Elements in the tensor are either 1 or 0 and most of them are 0s.

```
Tensor x = encoder::scalar(/*value=*/0.3,
        /*min_val=*/0.f,
        /*max_val=*/5.f);
cout << x << endl;
```

Out

```
{ 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0}
```

Spatial Pooler is a very commonly used layer in HTM. Trough an unsurpervised learning process, it can extract patterns from SDRs fed to it.

```
SpatialPooler sp(/*input_shape=*/{32}, /*output_shape=*/{64});
// Run (i.e. inference) the SpatialPooler.
Tensor y = sp.compute(x);
cout << y.cast(DType::Bool) << endl;
// If you want the SpatialPooler to learn from the data. Call the `learn()` function.
sp.learn(x, y);
```

Out

```
{ 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,␣
↪0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0}
```

After traning, you might want like to save the Spatial Pooler's weights for use in the future.

```
auto states = sp.states();
save(states, "sp.cereal");

//Alternativelly save as JSON
save(states, "sp.json");
```

1.5 Introduction to HTM Algorithms

Hierarchical Temporal Memory is a biologically possible machine learning algorithm developed by Numenta. Biological possiblity means the algorithms has to simulate (or approximate) behaivour and limitations of real neurons. i.e. Neurons has to speak binary (spikes) and there's no global learning (no backpropergation). In HTM Theory, this means HTM operates on Sparse Distributed Representation (SDR, basically binary tensors) and every layer operates indipendently.

This document will introduce HTM algorithms as a black box. It is highly reommended to watch the HTM School series on YouTube for deeper understanding of the material.

1.5.1 Encoders

Encoders are how data interact with HTM. As mentioned earlier, HTM can only accepts binary as input and only generates binary output. So a method of converting data into SDRs is required. There are several encoders avaliable by default.

All encoders (for now) are plan function calls in the `et::encoder` namespace. And every of them returns a tensor of Booleans; you can inspect the result of any encoder by printing the returned value.

```
auto sdr = encoder::gridCell1d(0.5);
cout << sdr << endl;
```

Out

```
{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
 0, 0, 0, 0, 0, 0}
```

The rule of thumb for an encoders is that for any given pair of value encoded. The farther the values are, the less 1 bits in the resulting SDR overlaps. The short ROOT script demonstrates this property (for a grid cell encoder).

```
root [] g = TGraph(100);
root [] x = encoder::gridCell1d(0.5);
root [] for(int i=0;i<100;i++) {
root []     float p = (float)i/100;
root []     auto s = encoder::gridCell1d(p);
root []     int overlap = sum(x && s).item<int>();
root []     g.SetPoint(i+1, p, overlap);
root [] }
root [] g.Draw();
root [] g.SetTitle("# of overlapping bits");
```

Out

number_of_overlapping_bits

Extending encodings

Unless in a very simple case, you might have multiple fields or non-standard types in your data. In such case, you might need to come up with your own encoder or use multiple of them together. Having your own encoder is easy - the only requirment of an encoder is **a)** returns a fixed length binary tensor **b)** the tensor have to be sparse and **c)** values futher apart should have less and less 1 bits overlapping. **a and b** are requirments. Failing to fulill will lead to a program crash. While **c** is nice to have but not having it will only causing bad learning results or not learning at all.

For example. We can come up with an basic scalar encoder that encodes values from 0~1.

```
Tensor dumb_encoder(float v)
{
    intmax_t start = v*(64-4);
```

(continues on next page)

(continued from previous page)

```
    // Create a tensor of length 64.
    // Then set 4 of the bits to 1
    Tensor sdr = zeros({64}, DType::Bool);
    sdr[{range(start, start+4)}] = 1;
    return sdr;
}

cout << dumb_encoder(0.1) << endl;
```

Out

```
{ 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

It is quite hard to build a robust encoder for complex data and in most cases the data are not related to each other. So pratically we would apply encoding to each field then concatenate all SDRs together.

```
Tensor complex_encoder(float v1, float v2, float v3)
{
        Tensor sdr1 = encoder::scalar(v1);
        Tensor sdr2 = encoder::scalar(v2);
        Tensor sdr3 = encoder::scalar(v3);
        return concat({sdr1, sdr2, sdr3});
}

cout << complex_encoder(0.1, 0.2, 0.3) << endl;
```

Out

```
{ 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

1.5.2 Spatial Pooler

Spatial Pooler is a dimention reduction/clustering algorithm. As a black box, a Spatial Pooler learns the associations between bits (learning which groups of bits frequently becomes a 1 together) and merges them together. The process removes reducdent information within the SDR and enforces a stable density (number of 1 bits in the SDR). Improving Temporal Memory's performance.

```
auto sp = SpatialPooler(/*input_shape=*/{256}, /*output_shape=*/{64});
auto y = encoder::gridCell1d(0.5);
cout << sp.compute(y) << endl;
```

Out

```
{ 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1}
```

An untrained Spatial Pooler is pretty much useless; knowing nothing about the input SDR and only guess the relation among bits. We need to feed data to it and allow it to learn the relations itself - by feeding the results back into the `learn` method.

```
auto sp = SpatialPooler(/*input_shape=*/{256}, /*output_shape=*/{64});
for(const auto& sample : dataset) {
    auto x = encode_data(sample);
    auto y = sp.compute(x);

    sp.learn(x, y); // Ask the SP the learn!
}
```

Like any machine learning algorithm, the Spatial Pooler have multiple hyper-parameters. The important ones include:

```
sp.setGlobalDensity(density); // The density of the generated SDR
sp.setBoostingFactor(factor); // Allows under-performing cells to activate
sp.setActiveThreshold(thr);   // How much activity can lead to cell activation

sp.setPermanenceInc(inc);     // These are the learning rates
sp.setPermanenceDec(dec);     // For both reward and punish
```

1.5.3 Temporal Memory

As the name implied, Temporal Memory is a sequence memory. It learns the relations of bits at time `t` and `t+1`. For a high level view, given a Temporal Memory layer is trained on the sequence A-B-C-D. Then asking what is after A, the TM layer will respond B.

```
// Some setup is requied for a TM
// In this example, the input shape is {256} and there are 16 cells per column
// last_active and last_pred are the stats of the TM
// This is annoning and will be fixed in a future release
Tensor last_active = zeros({256, 16}, DType::Bool);
Tensor last_pred = zeros({256, 16}, DType::Bool);

auto tm = TemporalMemory(/*input_shape*/{256}, /*cells_per_column=*/16);
auto [pred, active] = tm.compute(encoder::gridCell1d(0.5), last_pred);
cout << pred << endl;
```

Out

```
{{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 ....
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
```

The Temporal Memory has some very good properties. First, a Temporal Memory will respond "nothing" when it has no idea what is going on (unlike neural networks, they return at least something when met with unknown). Like in this case, we asked a Temporal Memory that has been trained on nothing to predict what's after 0.5. The layer responded with a zero tensor of shape `{input_shape, cells_per_column}`. Indicating nothing.

In most applications. We don't care about what's happning within each column but weather a column contains a cell that is a 1. This can be extracted with a `sum()` call.

```
// Continuing from the previous block
auto pred_sdr = sum(pred, 1, DType::Bool);
cout << pred_sdr << endl;
```

Out

```
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0}
```

Traning a Temporal Memory involved basically the same process as the Spatial Pooler. But with slightly more steps.

```
// Continuing from the previous block
for(const auto& sample : dataset) {
    auto x = encode_data(sample);
    auto [pred, active] = tm.compute(x, last_active);
    // You can you the predictions here
    ...
    // Now update the states
    tm.learn(active, last_state);
    tie(last_pred, last_active) = pair(pred, active);
}
```

Idealy, unless your input data is very simple. We recommend to pass the encoded through a SpatialPooler to reduce useless information.

```
for(...) {
    auto x = encode_data(...);
    auto y = sp.compute(x); // Use the SP!

    auto [pred, active] = tm.compute(y, last_active);
    ...
}
```

After traning, the TM should be able to predict what is possible in the next time step based on current input and the state. A Temporal Memory layer can gracefully deal with ambiguous situaction. When trained on the sequence A-B-C-B-C-D then asking what's after C without a context(past state), the TM will respond both B and D.

Detection anomaly

One of HTM's main use is to perform anomaly detection. The method is stright forward. Given a well trained Spatial Pooler, Temporal Memory and a cyclic signal. The only cause for the TM to not predicting well must be an anomaly in the signal. The TM's property ties in very well with the application. A TM will resolve ambiguous states by predicting everything and predicts nothing when it don't know.

We can compute the anomaly score using the TM's prediction from the last time step and the actual input from the current time step.

```
for(...) {
    auto x = encode_data(...);
    auto [pred, active] = tm.compute(y, last_active);

    float anomaly_score = anomaly(sum(last_pred, 1, DType::Bool), x);
    ...
}
```

Do note that detecting anomaly at `t=0` will always give you an anomaly score of 1 due to having no past history to rely on.

1.5.4 Classifers

It's good to have the ability to predict the future. But sometimes it's hard to make sense of the predictios in the SDR form. Etaler provides a biologically possible classifer to categorize results back into human readable information.

Note: The API of classifers are pron to change. And more classifers will be added.

```
auto clf = SDRClassifer(/*input_shape=*/{256}, /*num_class=*/10);

//Add SDRs to the classifer
for(int i=0;i<10;i++)
    clf.addPattern(encoder::gridCell1d(i/10.f), i);

cout << "The value 0.8 is close to "
    << clf.compute(encoder::gridCell1d(0.8))/10.f << endl;
```

Out

```
The value 0.8 is close to 0.8
```

1.6 Tensor

Tensors are how Etaler stores data. They are a minimal NDArray implementation. Thus it is currently lacking some features. But they should be enough for HTM.

For now content type of `int`, `bool`, `half` and `float` are supported.

1.6.1 Creating a Tensor

It is easy to create a Tensor

```
Tensor t = Tensor(/*shape=*/{4,4}, DType::Float);
```

At this point, the contents in the Tensor hasn't been initalized. To initalize, call `Tensor()` with a pointer pointing to a plan array. The Tensor object will copy all of the provided data and use them for initalization.

```
float data[4] = {3,2,1,6};
Tensor t = Tensor(/*shape=*/{4}, DType::Float, data);
```

1.6.2 Copy

Tensors holds a `shared_ptr` object that points to a actual implementation provided by the backend. Thus, copying via `operator =` and the copy constructor results in a shallow copy.

```
Tensor t = Tensor({4,4});
Tensor q = t; //q and t points to the same internal object
```

Use the `copy()` member function to perform a deep copy.

```
Tensor t = Tensor({4,4});
Tensor q = t.copy();
```

1.6.3 Accessing the raw data held by the Tensor

If the implementaion allows. You can get a raw pointer pointing to where the Tensor stores it's data. Otherwise a `nullptr` is returned.

```
Tensor t = Tensor({4,4}, DType::Int32);
int32_t* ptr = (int32_t*)t.data();
```

1.6.4 Copy data from Tensor to a std::vector

No matter where or how a Tensor stores it's data. the `toHost()` method copies everything into a std::vector.

```
Tensor t = Tensor({4,4}, DType::Int32);
std::vector<int32_t> res = t.toHost<int32_t>();
```

Be aware that due to `std::vector<bool>` is a specialization for space and the internal data cannot be accessed by a pointer, use uint8_t instead.

```
Tensor t = Tensor({4,4}, DType::Bool);
std::vector<uint8_t> res = t.toHost<uint8_t>();
//std::vector<bool> res = t.toHost<bool>();//This will not compile
```

Also `toHost` checks that the type you are stroing to is the same the Tensor holds. If mismatch, an exception is thrown.

```
Tensor t = Tensor({4,4}, DType::Float);
std::vector<int32_t> res = t.toHost<int32_t>(); //Throws
```

1.6.5 Indexing

Etaler supports basic indexing using Torch's syntax

```
Tensor t = ones({4,4});
Tensor q = t.view({2, 2}); //A vew to the value of what is at position 2,2
std::cout << q << std::endl; // Prints {1}
```

Also ranged indexing

```
Tensor t = ones({4,4});
Tensor q = t.view({range(2), range(2)}); //A view to the value of what is at position␣
↪2,2
std::cout << q << std::endl;
// Prints
// {{1, 1},
//  {1, 1}}
```

The `all()` function allows you to specsify the entire axis.

```
Tensor t = ones({4,4});
Tensor q = t.view({all(), all()});//The entire 4x4 matrix
```

1.6.6 Writing data trough a view

And you can write back to the source Tensor using `assign()`

```
Tensor t = ones({4,4});
Tensor q = t.view({2,2});
q.assign(zeros({2,2}));
std::cout << t << '\n';
//Prints
// {{ 0, 0, 1, 1},
//  { 0, 0, 1, 1},
//  { 1, 1, 1, 1},
//  { 1, 1, 1, 1}}
```

The Python style method works too tanks to C++ magic

```
Tensor t = ones({4,4});
t.view({2,2}) = zeros({2,2});
std::cout << t << '\n';
//Prints
// {{ 0, 0, 1, 1},
//  { 0, 0, 1, 1},
//  { 1, 1, 1, 1},
//  { 1, 1, 1, 1}}
```

But assigning to an instance of view doesn't work. Jusk like how things are in Python.

```
Tensor t = ones({4,4});
Tensor q = t.view({2,2});
q = ones({2,2});
std::cout << t << '\n';
//Prints
// {{ 1, 1, 1, 1},
//  { 1, 1, 1, 1},
//  { 1, 1, 1, 1},
//  { 1, 1, 1, 1}}
```

Technical note

Numpy uses the `__set_key__` operator to determine when to write data. If the operator is not called. Python itself handles object reference assignment and thus data is not written. However thers is no such mechanism in C++. So Etaler distingishs when to copy the reference it holds and when to write data using `operatpr= ()&` and `operator= ()&&`. When writing to an l-valued Tensor, the reference is copied. While assigning to an r-value, actual data is copied trough the view.

Which works in most cases, but there are caveats.

```
Tensor foo(const Tensor& x)
{
        return x;
}

foo(x) = ones({...}); //Oops. Data is written to x even tho it is passed as const!
```

1.6.7 Add, subtract, multiply, etc...

Common Tensor operations are supported. Incluing +, -, *, /, exp, log(ln), negation, Tensor comparsions, and more! Use them like how you would in Python. The comparsion operators alowys return a Tensor to bools. The others return a Tensor of what you get in plan C/C++ code.

```
Tensor a = ones({4,4});
Tensor b = a + a;
```

1.6.8 Brodcasting

Etaler supports PyTorch's brodcasting rules without the legacy rules. Any pair of Tensors are bordcastable if the following rules holds true.

(Stolen from PyTorch's document.)

- Each tensor has at least one dimension.
- When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

For example:

```
Tensor a, b;

//The trailing dimensions match
a = ones({2, 2});
b = ones({   2});
std::cout << (a+b).shape() << std::endl;
// {2, 2}

//But not necessary
a = ones({6, 4});
b = ones({1});
std::cout << (a+b).shape() << std::endl;
// {6, 4}

//This fails
a = ones({2, 3});
b = ones({   2});
std::cout << (a+b).shape() << std::endl;
//Fails
```

Unlike PyTorch and NumPy, Etaler does not support the lagecy brodcasting rule. It doesn't allow certain tensors with different shapes but have the same amount of elements to brodcast together.

```
//This would get you a warning in PyTorch and works in NumPy.
//But not in Etaler
a = ones({4})
b = ones({4, 1})
std::cout << (a+b).shape() << std::endl;
//Fails
```

1.6.9 Copy Tensor from backend to backend

If you have multiple backends (ex: one on the CPU and one for GPU), you can easily transfer data between the backends.

```
auto gpu = make_shared<OpenCLBackend>();
Tensor t = zeros({4,4}, DType::Float);
Tensor q = t.to(gpu);
```

1.6.10 Catch-yas

Using the Tensor() constructor to create a Tensor of 1 dimentions in facts creates a Tensor of the given value.

```
Tensor t = Tensor({4});
std::cout << t << std::endl;
//Prints {4} instead of {x, x, x, x}
```

1.7 Python bindings

1.7.1 PyEtaler

PyEtaler is the offical binding for Etaler. We try to keep the Python API as close to the C++ one as possible. So you can use the C++ document as the Python document. With that said, some functions are changed in the binding to make it more Pythonic.

```
>>> from etaler import et
>>> et.ones([2, 2])
{{ 1, 1},
 { 1, 1}}
```

1.7.2 ROOT

If cppyy is not avaliable to you for any reason. You can use Etaler in Python via ROOT and it's automatic binding generation feature.

```
# Load ROOT
import ROOT
gROOT = ROOT.gROOT
gSystem = ROOT.gSystem

# Include Etaler headers
gROOT.ProcessLine("""
#include <Etaler/Etaler.hpp>
#include <Etaler/Algorithms/SpatialPooler.hpp>
#include <Etaler/Algorithms/TemporalMemory.hpp>
#include <Etaler/Algorithms/Anomaly.hpp>
#include <Etaler/Encoders/GridCell1d.hpp>
""")
# Load the library
gSystem.Load("/usr/local/lib/libEtaler.so");
```

(continues on next page)

(continued from previous page)

```
# make a handy namespace alias
et = ROOT.et
```

Then you can call Etaler functions from Python.

```
s = et.Shape()
[s.push_back(v) for v in [2, 2]] #HACK: ROOT doesn't support initalizer lists.
t = et.ones(s)
print(t)
"""
{{ 1, 1},
 { 1, 1}}
"""
```

1.8 Using with cling/ROOT

cling is a JITed C++ interpter and ROOT is a data analyisis framework by CERN built upon cling (since version 6). Thay are a valuable tool that provides a interactive C++ shell when using Etaler/doing experiments. This file documents how to use Etaler within clang/ROOT.

1.8.1 General solutions

After installing Etaler to system and launching cling or ROOT. You can load Etaler via the `#pragma cling load` command.

By default cling only looks in `/usr/lib`. So you'll need to specisify the full path if the library is not located there.

```
> cling -std=c++17

****************** CLING ******************
* Type C++ code and press enter to run it *
*             Type .q to exit             *
*******************************************
[cling]$ #pragma cling load("/usr/local/lib/libEtaler.so")
[cling]$ #include <Etaler/Etaler.hpp>
```

Or you'll have to specsify an apporate search path.

```
> cling -std=c++17 -L /usr/local/lib
[cling]$ #pragma cling load("Etaler")
[cling]$ #include <Etaler/Etaler.hpp>
```

Otherwise, loading the library can be simplifed:

```
> cling -std=c++17
[cling]$ #pragma cling load("Etaler")
[cling]$ #include <Etaler/Etaler.hpp>
```

The same solution works with ROOT too! By default ROOT is compiled with C++11 only. You'll need to compile your own version of ROOT with C++17 enabled (by using `cmake -Dcxx17=ON`). Or you'll need to download a version with C++17 enabled. Like the one in Arch Linux's repo

```
> root
   ------------------------------------------------------------
  | Welcome to ROOT 6.16/00                  https://root.cern |
  |                               (c) 1995-2018, The ROOT Team |
  | Built for linuxx8664gcc on Jan 23 2019, 09:06:13           |
  | From tags/v6-16-00@v6-16-00                                |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q' |
   ------------------------------------------------------------

root [0] #pragma cling load("/usr/local/lib/libEtaler.so")
root [1] #include <Etaler/Etaler.hpp>
```

The same -L flag can be applyed to ROOT.

```
> root -L /usr/local/lib
root [0] #pragma cling load("Etaler")
root [1] #include <Etaler/Etaler.hpp>
```

1.8.2 For cling

Cling accepts the `.L` command. (ROOT also has a .L coomand, but it is for a different function)

```
> cling -std=c++17 -L /usr/local/lib
[cling]$ .L Etaler
[cling]$ #include <Etaler/Etaler.hpp>
```

1.8.3 For ROOT

If you want load the library programmatically. You can load Etaler via `gSystem`. You'll have to specisify the full path using this method.

```
> root
root [0] gSystem->Load("/usr/local/lib/libEtaler.so");
root [1] #include <Etaler/Etaler.hpp>
```

1.8.4 Using Etaler under an interactive C++ shell

After loading the library. You can use the library as you would normally. (And ROOT imports the `std` namespace by default.)

```
root [2] using namespace et;
root [3] Tensor t = ones({3,3});
root [4] cout << t << endl;
{{ 1, 1, 1},
 { 1, 1, 1},
 { 1, 1, 1}}
root [5]
```

1.9 GUI

This file documents the steps to add GUI features to Etaler running under container.

- On the host run "xhost local:root" to enable X11 connection on localhost from the container
- Build and run the tm_visualize example

1.10 Contribution

There are may different ways to help!

1.10.1 Using Etaler

The easiest and the most direct way to help is by using the library. If you find a bug or a missing feature, please open an issue and let us know how we can improve. If you have built a cool project usign Etaler. Share it!

1.10.2 Developing Etaler

If you want to develop the library it self, you are very welcomed! We are excited to see a new PR pop up.

Deisgn guidlines

- Don't care about x86-32 (Any 32bit architecture in fact). But no intentional breaks
- Use DOD. OOP is slow and evil
- Backend calls should be async-able, allow parallelization as much as possible
- Seprate the compute backend from the API frontend
- Make the design scalable
- See no reason to run a layer across multiple GPUs. Just keep layers running on a single device
- Keep the API high level
- Braking the archicture is OK
- Serealizable objects should reture a `StateDict` (`std::map<std::string, std::any>`) object for serialization
  - Non intrusive serialization
  - Reseves a StateDict object to deserealize
- Language binding should be in another repo
- Don't care about swarmming
- follow the KISS (Keep it Simple Stupid) principle
- Configuration files are evil (Looking at you NuPIC)
- Avoid raw pointers. They are evil

If you are adding a dependency to the library. Please discuss it in a relevant thread or make a new issue. For now, we are actively avoiding Boost and try to stick to STL as much as possible. Using Boost might lead to trouble later on. But we might decide use Boost later on.

1.11 Backend Design

Backends are how Etaler supports computing on different device/processors. They perform the actual computing and memory managment. Currently there are 2 backends avaliable.

- CPUBackend
- OpenCLBackend

They should cover more than 99% of the use cases. When needed, other backends can be added to support new devices.

1.11.1 Total front-end/back-end seperation

Etaler seperated it's frontends and backends. i.e. Calling `Tensor::Tensor()` does not immidatelly create a Tensor object. Instead it calls a backend to create the object for you. Depending on which backend is called. The resulting Tensor can live on CPU's main memory or the GPU's VRAM. (Omiting a lot of details.)

This applies to all Tensor operations and HTM algorithms. All of them are provided by a backend and the Tensor object only calculates what backend-indipendent parameters are needed to generate correct result. Allowing a very simple way to have a Tensor running anywhere.

1.11.2 Backend APIs

By design (might change in the future). All backend-exposed compute API are expecting thair own Tensors being passed in . *View of tensors are not expected* (besides `realize()` and `assign()`. They handle views). If anython besides an actual Tensor is passed in, the **backend aborts**.

The frontend APIs of common functions like `sum`, `cast` automatically handles realizing. But HTM specific APIs don't.

1.11.3 Backend name

You can get the backend's name usging the `name()` method.

```
std::cout << backend->name() << "\n";//prints: CPU
```

1.11.4 Default Backend

Etaler provices a convient function `defaultBackend()` that returns a usable backend. When no backends are provied to algorithms/objects the default backend is used.

By default `defaultBackend()` returns a `CPUBackend`. You can which backend is used by calling the `setDefaultBackend()` function.

Like so:

```
setDefaultBackend(std::make_shared<OpenCLBackend>());
```

1.11.5 Using multiple backends

Using multiple backends should be easy. Just initalize multiple backends and tensors on them! You can even have different threads controlling different backends for maxium performance. The backends are not thread-safe tho. You'll have to handle that yourself.

1.11.6 Tensors and views, how do they work

From a technical point. Each backend implements it own XXXBuffer (ex. CPUBuffer) class, storing whatever is needed. When the backend being requested to create a tensor (the `createTensor` method called). The backend returns a shared_ptr pointing to XXXBuffer, which is then wrapped by a TensorImpl. When the reference counter drops to 0, the `releaseTensor` method is called automatically (Also all TensorImpl holds a shared_ptr to the backend, so you don't need to worry about the backend being destructed before all tensors being destructed).

When creating a view. Like Numpy and PyTorch's implementation we modifies the offset and stride of the tensor.

But not all backend APIs support handling strides. (Espcally HTM algorithms and those modifies data in-place). If a strided Tensor is sent to a API that doesn't support strides. Backend aborts.

1.12 Developer Notes

This file documents some facts about the codebase that might be useful for it's developers and users.

- The `Error.hpp` file provides the `et_assert()` macro for function pre/post condition checking. It should be replaced with C++20 contracts when avaliable.
  - Use it to check for function pre and post condition fails.
  - You can also throw an `EtError` if you like it for recoverable errors.
  - Not disabled in release builds
- `Error.hpp` also provides the `ASSERT()` macro that works exactly like C `assert()`. But
  - Does not generate a unused variable warning in a release build
- Address Sanitizer is your good friend to debug buffer overflows if you are on Linux or OS X.
  - But it might cause Etaler not able to detect OpenCL platforms.
  - Memory Sanitizer will work with OpenCL. But is too sensitive.
- Nvidia's OpenCL implementation although very optimized, has piles upon piles of problems.
  - Use POCL w/ CUDA backend for debugging. POCL is a lot slower, but very stable.
  - Or use Intel/AMD's OpenCL SDK

1.13 About the OpenCL backend

This is a brief document/note about the OpenCL backend.

1.13.1 OpenCL standard version

Currently the backend only requires and uses the features provided by OpenCL 1.2. But OpenCL 2.0 support is planned.

1.13.2 Program caching

The OpenCL backend caches programs automatoically. If a program `cast` already exists in the kernel manager. Asking it to compile another version of `cast` does nothing. Set the `force_override` flag to force a compilation and override.

1.13.3 Local Memory

Due to how HTM work - being very memory bangwidth hungry and the input SDR is relativelly small. The OpenCL backend tries to stores data in the GPU's local(on-chip) memory so more bandwidth can be used for fetching synapse.

However this also poses limitations. Since the input Tensor is copied into the local memory. The size of the input Tensor cannot exceed the size of local memory (48KB on NVIDIA cards and 64KB on AMD cards). If larger inputs are encountered, Etaler simply switches to a version not using local memory. Also some GPUs - all of them are mobile GPUs - don't have such local memory and is emulated using global memroy. Etaler uses kernels without local memory optimization.

1.13.4 Global inhibition

Currently finding the top-k values are done by a quick counting based scanning methoud. Performed by a single work group and stores it's counters in Local Memory. This is fast and accurate as long as the size of the input array is small (the max value is less then the size of the counter). Thus another algorithm might be needed for big arrays.

1.13.5 OpenCL kenrel distribution

The OpenCL backend searchs for the kernels it needs in the following paths: `./kernels/`, `../kernels/`, `/usr/local/share/Etaler/kernels` and `/usr/share/Etaler/kernels/`. If the file is found, then it is read and cached in memory. If not, an exception is raised. The kernel files are installed when installing the library.

If you have your kernels stored at a different location. You can set the `ETALER_KERNEL_PATH` enviroment variable to make your path avavliable.

1.13.6 JIT compiling views

The OpenCL backend generates the OpenCL kernels to copy/write to Tensor views at runtime. Thus copying from a view might be slow. It the problem turns out to be too bug a problem. It will be cchanged.

1.13.7 NVIDIA's OpenCL implementation

NVIDIA's OpenCL implementation can crash without notifing the user. (kerenl can crash without abort, generating error code at the wrong places, etc...). Use POCL's CUDA backend for varification that the kernel is running correctly.

1.13.8 program name mangling

Since the OpenCL backend tracks programes using a key. Name mangling (in Etaler's case, appending the hash of the compiler argumnents to the end of the key) is required to support multiple versions of the same program (with different `-D` defines, etc...).

1.13.9 RPi VC4CL Support

VC4CL is **not suported** for now. The limitation of VC4CL only supporting up to 12 PE per work group is not taken into account in the OpenCL backend. (And VC4 uses global memory to emulate local memory, it is going to be slow),

1.13.10 Altera AOCL / Xiinx SDAccel support

FPGA based OpenCL although interaseting, are not supported now due to the lack of a API callable compiler.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search