Ogma Corp Project Documentation

OgmaNeo 2

Eric Laukien

December 24, 2019

Exploring Sparse Predictive Hierarchies (SPH)

Contents

1 Introduction

1.1 Sparse Predictive Hierarchies

Our brains, as well as those of many mammals, are capable of efficiently controlling organisms with low resource requirements. We wish to reproduce these capabilities in computer software.

Instead of blindly guessing as to what kind of algorithms the brain might use, we try to operate within biological limitations in order to ensure that we are exploring in the right areas. While we don't need to reproduce biological algorithms exactly, we wish to capture the essence of the algorithms the brain implements and reduce them to programs with low complexity that can exhibit similar properties.

Sparse Predictive Hierarchies (SPH) are a general model of how we think the human brain functions. It can perform tasks involving sequence prediction, world model building, and reinforcement learning. It is based on online and incremental learning algorithms developed at Ogma Corp. SPH is:

- **S**parse: It uses only small fractions of the available resources at a time.
- **P**redictive: It models the world as sequences of events, where previous events are causally related to subsequent events.
- **H**ierarchies: It consists of several layers of abstraction, both spatially and temporally.

While SPH is "deep" in the sense that it uses these multiple layers of abstraction in a hierarchy, we do not consider it to be "Deep Learning" (DL). Deep learning is not biologically plausible, mostly due to continuous, dense (not sparse) representations and the use of the backpropagation algorithm.

1.2 Historical Context

Artificial intelligence (AI), and in particular Artificial General Intelligence (AGI) has been a common goal of many computer scientists since before computers were even built. AI research has recently enjoyed a reinvigoration due to the advent of low-cost compute in the form of graphics cards (GPUs), far more data, and several algorithmic improvements. Prior to this, however, there was a period of low research activity, often referred to as the "AI winter", which began in the 1970s. The cause of this stall in research was thought to be that of over-promise and hype with poor results. However, interestingly, the algorithms pioneered then - in particular the backpropagation algorithm - remain the basis of most modern artificial intelligence methods. Our method is an exception to this.

In terms of architecture, we adopt the notion of "repeated computational units" - named columns as a general term for an abstract model of the cortical column as discovered by Vernon Mountcastle in 1957. While there exist many brain regions that are specialized to various tasks, the portions of the brain thought to be more general-purpose in function, such as the "newly evolved" neocortex, are surprisingly uniform in structure. The neocortex in particular consists of several layers of column-like structures. These are organized into minicolumns and hypercolumns. These are groups of neurons that are repeated throughout the neocortical sheet.

As described first by Mountcastle, these "vertical" columns are further divided horizontally into 6 layers (5 of which are visually distinct). The neurons that make up the cortical columns average round 80% excitatory ("positive") neurons, and 20% inhibitory "negative" neurons. The inhibitory neurons regulate the activity of the excitatory neurons to produce a sparse code. They do so by asynchronously decorrelating the excitatory neurons, forming a sort of "one hot" or "winner takes all" (WTA) circuit.

Sparse coding is an important aspect of SPH, as it provides the sparse representations used by the majority of the algorithm. Sparse coding is a field in itself, with two fundamental flavors of sparse coding algorithm: "Neural" algorithms (biologically plausible), and more classic sparse coding algorithms that relate to the mathematical field of matrix factorization. Neural sparse coding was first theorized by Olshausen and Field in 1996. From then on, many biologically plausible algorithms were developed, with many varieties and amounts of realism. We have experimented with many kinds of sparse coding algorithms of both types (bio-plausible and not), and conclude that both can provide excellent sparse codes if done properly. We find that what is important is not the details of the algorithm, but rather that it is efficient and provides an *end result* that is biologically plausible.

1.3 Motivation

Currently, Deep Learning methods are requiring vast amounts of compute, and suffer from various algorithm problems such as catastrophic interference (forgetting), inability to learn online/incrementally, and little biological relevance. SPH attempts to bridge the gap between:

- Highly realistic implementations of small portions of neural circuitry.

- More general, practical, and immediately applicable as well as justified algorithms.

We find that by embracing the sparsity found in biology and finding practical algorithms that use it, we can achieve vast performance increases over Deep Learning, as well as avoid several of the aforementioned algorithmic flaws associated with it.

From a practical perspective, these improvements allow us to apply intelligent algorithms to previously infeasible applications, in particular those involving embedded systems and robotics.

2 Our Model

2.1 Notation

In the following, we use the subscript t to specify time index. However, sometimes it is left out, in which case the described process is implicitly assumed to occur at the current time t.

Generally, we use upper case letters to describe either matrices, vectors, and tensors. We use lower case letters to denote scalar variables.

Finally, a hat indicates an approximated or alternate version of an existing variable. For instance, $\hat{x}$ refers to some alternate or closely related version of x.

2.2 Encoders and Decoders

An encoder is a mapping and associated states that transform from some representation X to some "hidden" representation H, where both X and H may be forced to have certain properties.

A decoder is the reverse mapping of an encoder. Given H, we can map back to some approximation of the original X, $\hat{X}$. Often, an encoder's parameters can be re-purposed to implement a decoder.

2.3 Predictor/Predictive Decoder

A predictor (or predictive decoder) is similar to a decoder, but instead of outputting $\hat{X_t}$ given some H_t, it outputs $\widehat{X_{t+1}}$ given H_t. In SPH, prediction refers to the process of predicting the next timestep $(t+1)$ of the targets as opposed to the original targets.

2.4 SPH

In SPH, a layer contains an encoder and a predictive decoder (encoder-decoder pair).

SPH consists of a hierarchy of layers in which each layer is strictly above or below another (Which implies that the layers form a stack). For each layer above the first layer, the layer's encoder's input tensor X is the output tensor H of the encoder from the layer below the current one. For each layer below the last layer, the layer's decoder's feedback input tensor $\hat{X}$ is the output of the decoder from the layer above the current one.

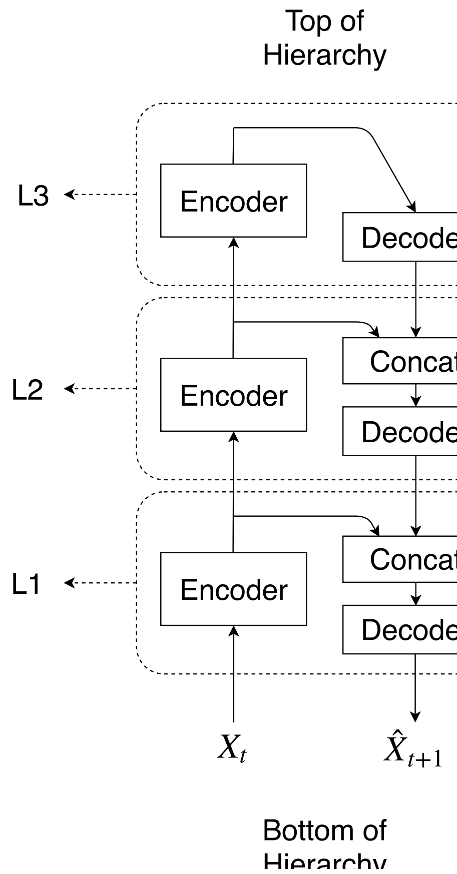

Figure 2.1: An example of SPH with three layers.

Processing proceeds in both directions vertically (top-bottom, bottom-top). First, there is an "up" pass - layers are iterated sequentially from bottom to top, in which each encoder encodes the hidden layer state of the encoder directly below itself. Then, there is a "down" pass, in which each decoder predicts the *next timestep (t+1)* of the hidden state of the encoder directly below itself by using both the current layer encoder hidden state and top-down (previous decoder) feedback. This describes a synchronous process, although it is also possible to run the layers asynchronously. For simplicity, we assume this synchronous processing.

A key portion of this algorithm is that each encoder-decoder pair predicts (through the predictive decoder) the next timestep (or timesteps, as we will see when we add "Exponential Memory") of the input/target of that encoder-decoder pair. We are essentially using "time as the supervisor" for each layer. The next timestep(s) prediction provides the output of the algorithm at the bottom-most layer, and provides useful feedback to the next lower layer for all other layers.

Each layer uses predictions of its own hidden state from layer(s) above to refine its own predictions, which in turn are used by the layer(s) below. What is interesting about this method is that no error propagation is necessary. "Time as the supervisor" essentially means that targets are provided by waiting for time to elapse at each layer. This elapsed time provides a useful, informative signal, and is a good heuristic for producing accurate predictions. This method is completely local in both space and time.

We will now discuss how to implement the encoders and decoders individually.

2.5 The Encoder

The encoder can be basically any sort of nonlinear mapping from some input vector X to some hidden vector H. While it may be tempting to simply use an autoencoder, doing so would sacrifice the online learning capabilities of SPH. So, instead we turn to the field of sparse coding, and implement an approximation of the sort of sparse coding thought to occur in the brain. We have tried many different sparse coding algorithms, two of which stand out among the rest: the exponential reconstruction encoder (ERE) and the two-stage inhibition explaining-away encoder (just referred to as 2SI).

In our setup, each hidden state vector H is really a 3D tensor. They are treated as vectors (raveled) in our algorithms, though. While the setup as a whole can be any dimension, we wish to use local, sparse weight matrices to conserve performance and memory, but would also like to be able to handle image-like data (2D). We also decide that sparsity should occur across a separate dimension. We therefore introduce the notion of "columns" - binary one-hot vectors that make up one of the dimensions of H. Each node in a column is called a cell. H is therefore a 2D grid of columns, giving it an overall 3D grid-of-cells shape. The choice of this particular encoder hidden state representation has several advantages:

- It is sparse.
- It is binary (normalized).
- Each column may have a single receptive field shared among all "cells" - this ensures that during the sparse coding process, all cells compete for the same input. We call this "fair cell competition".
- Local receptive fields will function well with images.
- It is straightforward to compute indexed locations of cells, since the tensor represents a square 3D grid (as opposed to an irregular structure).

We call this shape Columnar Sparse Distributed Representation (CSDR).

Let us assume that X also has a similar CSDR shape, but possibly with different dimensions. This consistency makes writing an optimized version of the algorithm much simpler, but does require us to add an explicit "pre-encoder" that converts the (possibly continuous, dense, any-dimensional) data to the CSDR shape. This is a topic of its own, and will be covered later. For now, we assume that the encoder hidden state H as well as the input X are both in CSDR format.

2.5.1 ESR

Exponential Sparse Reconstruction (ESR) is a particularly simple to implement encoder.

ESR falls into the category of reconstruction error optimization sparse coding algorithms. However, it only performs a single sparse code solving iteration. We define some dictionary D (a "weight" matrix), which can also be thought of as the "synapses". In practice, D is a sparse weight matrix with local receptive fields shared among cells in a column.

We find the sparse code by simply "activating" the "hidden" nodes, followed by column-wise one-hot inhibition (argmax).

$A \leftarrow D \cdot X$
$H \leftarrow ColumnWiseOneHot(A)$

The $ColumnWiseOneHot$ function sets the largest value in each column to 1 and the rest to 0.

In order to update the dictionary, we first reconstruct the hidden state H:

$R \leftarrow e^{D^T \cdot H}$

Where we multiply H by the transpose of D and perform an element-wise exponentiation. The exponentiation is vital in order to have the algorithm perform well, and will be explained later.

Next, can update the dictionary like so:

$E \leftarrow X - R$
$D \leftarrow D + \alpha(E \otimes H)$

The dictionary update occurs after the sparse code has been found. Learning is simply the delta rule that attempts to better reconstruct the input given the current code.

The reason for the exponentiation (which gives the algorithm its name) is due to a method found empirically to perform better. We initialize D to small negative noisy values. This means that initially, the reconstruction R will by close to 1. Since X is binary, E will negative. This means that D is "initialized high" and decreases over time. This is a method for avoiding the "dead unit" problem of sparse coding - when certain cells are never used and can no longer learn as they are never selected. By initializing such that weights start by decreasing, the less often a neuron is selected for learning, the more likely it will be selected in the future as other neurons that were selected before will have lower weights (and therefore lower activations A).

As an improvment, we find that adding some form of "early stopping" can be quite helpful. That is, when the reconstruction is already correction (the CSDRs match), then no updates to the dictionary are performed. This is used in some variants.

2.5.2 2SI

While ESE works well in many instances, it is not without issues. In particular, it has problems differentiating between certain patterns, where one pattern is a subset of another. This problem is mitigated by only taking other CSDRs as input, but it does not go away completely.

Another encoder we have tried that does not suffer from this issue is the two-stage inhibition explaining-away encoder (2SI). This encoder is more closely tied to biology than ESE, and does not require reconstruction. Further, it can be implemented with little more than populations of excitatory and inhibitory neurons with Hebbian learning rules. For simplicity, we do not model the inhibitory neurons directly; rather, we model inhibition as negative connections coming from excitatory neurons.

2SI has the same feed-forward weight matrix, or dictionary D, as ESE. However, it also has another matrix (also with sparse, local connectivity) of lateral connections. These lateral connections L serve to decorrelate the behavior of columns. The lateral connections connect to all cells in a radius around a column, except for itself (as we do not want self-decorrelation, which is meaningless).

2SI proceeds in two phases:

- Iterative Sparse Coding: Find the CSDR state that best represents the input X with given D and L.
- Learning: Update D and L to improve the code quality.

Note that the first phase is iterative - it requires a few "solving" iterations to find the appropriate representation. This process is referred to as "explaining-away", as the code settles on a value with minimum correlation between states.

For solving the sparse code, we first compute a stimulus S, an activation A, and an initial state for H:

$S \leftarrow \frac{D \cdot X}{|X|}$
$A \leftarrow S$
$H \leftarrow ColumnWiseOneHot(A)$

Where $|X|$ indicates the L1 normal of X. Since X is typically also a CSDR, this value is usually just the number of active input units.

Next, we iterate for some number of "solving" iterations. We typically only need around 3 solving iterations to get a decent quality code.

$I \leftarrow \frac{L \cdot H}{|H|}$
$A \leftarrow A + S * (1 - I)$
$H \leftarrow ColumnWiseOneHot(A)$

Where I is the total lateral inhibition, and $*$ is element-wise multiplication.

In order to learn D and L, we simply use Hebbian rules for both.

For each active cell in H:

$D_H \leftarrow D_H + \alpha(X - D_H)$
$L_H \leftarrow L_H + \beta(H - L_H)$

Where α and β are learning rates.

For 2SI, we generally initialize D to random values close to 1, and L to random values close to 0. The reasoning for this is similar to that of ESE, this initialization avoids the "dead unit" problem.

2.6 The Predictor/Predictive Decoder

The decoder portion of the layer is typically far simpler than the encoder. In our implementation, this is simply a logistic regression that outputs the one-hot columnar prediction. It takes as input a concatenation of H from the encoder, as well as some additional feedback F (prediction from decoder above, if there is one).

The decoder therefore predicts $\hat{X}_{t+1}$ given $C_t = concat(H_t, F_t)$ as input. To do this, it uses weight matrix W.

Variable names are reused from the encoder section, but they are not the same ones as used there.

The decoder proceeds by first making a prediction:

$A_{X_{t+1}} \leftarrow W \cdot C_t$
$\hat{X}_{t+1} \leftarrow ColumnWiseOneHot(A_{X_{t+1}})$

We introduced a new activation vector similar to that of the encoder, but this time in the shape of X. We then wait a timestep, and before performing the next prediction, we update the weights to point towards the "true" X that was just now observed:

$W \leftarrow W + \beta(X_t - \sigma(A_{X_t})) \otimes C_{t-1}$

Where σ is the logistic sigmoid function.

2.7 Combined Encoder and Decoder

Together, and encoder and a decoder form a single layer. Since in our particular implementation the state of the input and the layer is in a 2D CSDR format, our layer looks roughly as follows:

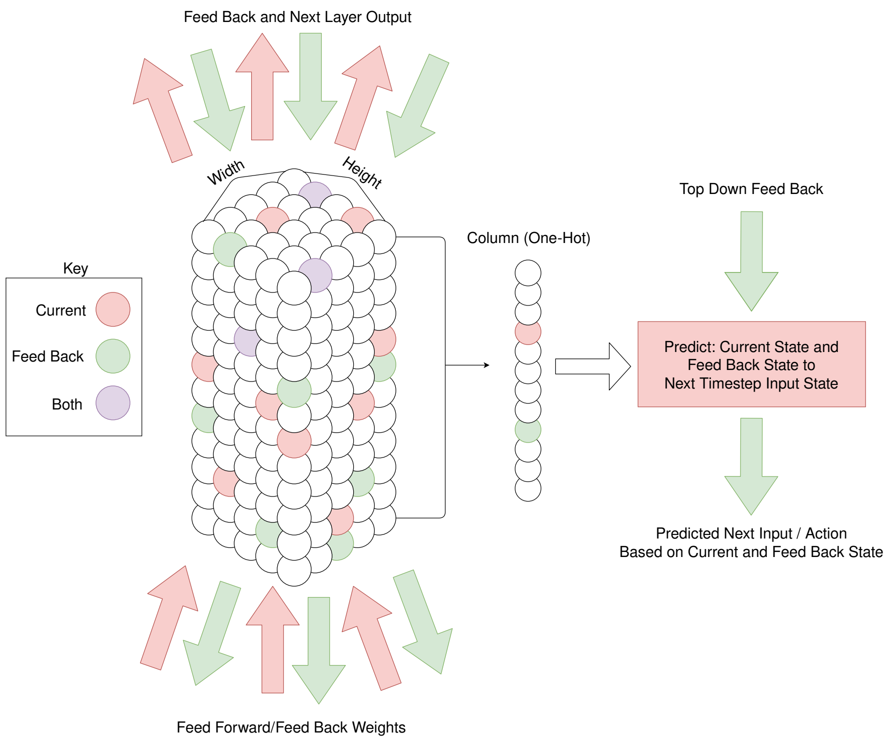
Feed Back and Next Layer Output
Width
Height
Top Down Feed Back
Column (One-Hot)
Key
Current
Feed Back
Both
Predict: Current State and Feed Back State to Next Timestep Input State
Predicted Next Input / Action Based on Current and Feed Back State
Feed Forward/Feed Back Weights

Figure 2.2: Single Layer

Note that we draw the *H* as overlapping with the predicted *X* of the next higher layer (current, feed back, both).

2.8 Exponential Memory

While we predict the next timestep of input at each layer, the hierarchy of layers doesn't actually have any memory up until now. It only predicts off of the current input, and does not take past inputs into consideration. It therefore cannot operate in partially observable environments. However, we have a method for not only giving the hierarchy memory, but doing so elegantly and with large capacity. We will now introduce the notion of Exponential Memory (EM).

While we could just add recurrent connections to each layer's encoder, there is a better way. We instead have each higher layer run slower than the previous. This means that each layer going up will cover a larger and larger temporal horizon. This is sort of like a bi-directional clockwork recurrent neural network.

We typically have each layer N times slower than the last, and N is almost always 2. This means that if we have L layers, we can represent N^L timesteps of memory. This is a finite horizon, but one that grows exponentially with respect to L, giving it the name "Exponential Memory".

Note that it is possible to have N differ between layers. We could, for instance, have $N = 1$ (no memory) but every third layer $N = 2$ (double). This allows for more fine grained layer count to memory horizon ratios.

Since each layer is N times slower, it also receives N times the amount of inputs, and predicts N times the amount as well. We refer to this as "striding". Each layer therefore groups together N time steps, and predicts the next N subsequent timesteps. Those subsequent timesteps are split and fed individually to the layer below (or as a prediction) based on the current clock tick. We call this "de-striding" - it essentially assures that each layer still receives a prediction of its own hidden state H one timestep into the future, at each timestep.

In an image, exponential memory looks like this:

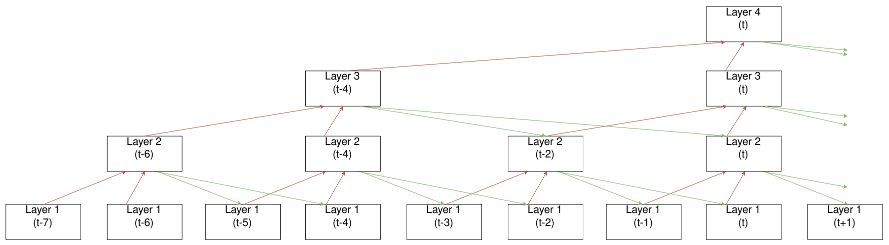

Figure 2.3: Exponential Memory

Red arrows indicate striding (with encoder dictionary D) and green arrows indicate de-striding (with predictor weights W). Note that we label an encoder-decoder pair simply as "Layer" in this image.

2.9 Pre-Encoders and Pre-Decoders

Pre-encoders are used to convert from arbitrary data to the CSDR format. We need these since the regular encoders (at least in our implementation) only accept other CSDRs as input, and predict CSDRs as well.

Pre-decoders are simply the inverse of the corresponding pre-encoder. This allows us to retrieve predictions in the same format as the original data.

Often, we can simply bucket-encode (and decode) values like individual scalars into columns. For instance, if we have a value we know to be in the range [a, b], we can simply rescale it into the range of a column [0, columnSize), and assign it to the next closest cell index in the column.

However, we often need more complex pre-encoders/decoders. For images for instance, we typically want some sort of sparse coding algorithm that works with the dense 2D color (3 channel) image data. For things like sound waves, we might want to find a way to convert MFCCs to CSDRs and then CSDRs back to MFCCs or the waveform directly. Pre-encoding and pre-decoding is highly dependent on the application, and custom pre-encoders/decoders will typically perform better in their niche scenarios than more general-purpose ones.

From a biological perspective, pre-encoders are very prevalent in the brain, since neurons don't accept raw e.g. visual or auditory input. The visual stream performs a large amount of pre-processing before the image even leaves the eye. The reason for this pre-processing occuring in the eye itself is that the optical nerve cannot handle the bandwidth of the full image. Eventually, images have some sort of sparse coding applied to them in V1, producing the familiar Gabor wavelets.

Audio is also interesting from a biological perspective. Neurons can only fire a thousand times a second or so, but audio requires sensitivity to changes in pressure in the ear at timescales of over 10000Hz. Therefore, a lot of pre-processing occurs here as well before the data even reaches the neurons. Audio features such as MFCC (Mel Frequency Cepstrum Coefficents) attempt the human ear in the way it transforms audio waveforms into a small set of features. From there, these features can be further transformed into CSDRs (for our particular use).

2.10 Reinforcement Learning

It isn't immediately obvious how to perform reinforcement learning with SPH. While still one of our primary research areas, there are generally three categories of algorithm that we investigate:

- Unsupervised Behavioral Learning (UBL) (most biologically plausible)
- Agent swarm learning (medium biological plausibility)
- Routing (not biologically plausible, in the current form)

In the first two of these, we use the concept of "action-prediction" - a prediction of X_{t+1} may contain both input data as well as action data generated by the SPH itself from the previous prediction (added in to the new input data through concatenation). This is mostly just a different way of interpreting predictions in the context of agents.

2.10.1 UBL

This is possibly the most elegant and simple solution for reinforcement learning, but also performs the worst at the moment.

The idea is to modify each layer's predictive decoder's learning phase. Instead of both inferring and learning from the same concatenation of current state H_t and feedback F_t, we instead replace F_t with H_{t+1} for learning only. This requires that the predictor is activated again (twice per timestep). However, this change essentially conditions the predictor differently - it now associates a transition $H_t \to H_{t+1}$ with the corresponding "action-prediction" X_{t+1}. Now during inference (prediction), we can replace the H_{t+1} with F_t again. These essentially makes each layer treat F as a "goal" state - one that the system wishes to transition to. This isn't really reinforcement learning, but can achieve similar results. With this system, each predictor sets the goal of the predictor below in such a way that it tries to move towards its own goal. It does this entirely through association of transitions with their corresponding action-predictions.

In an abstract sense (not SPH specific), the algorithm looks as it does in figure 2.4.

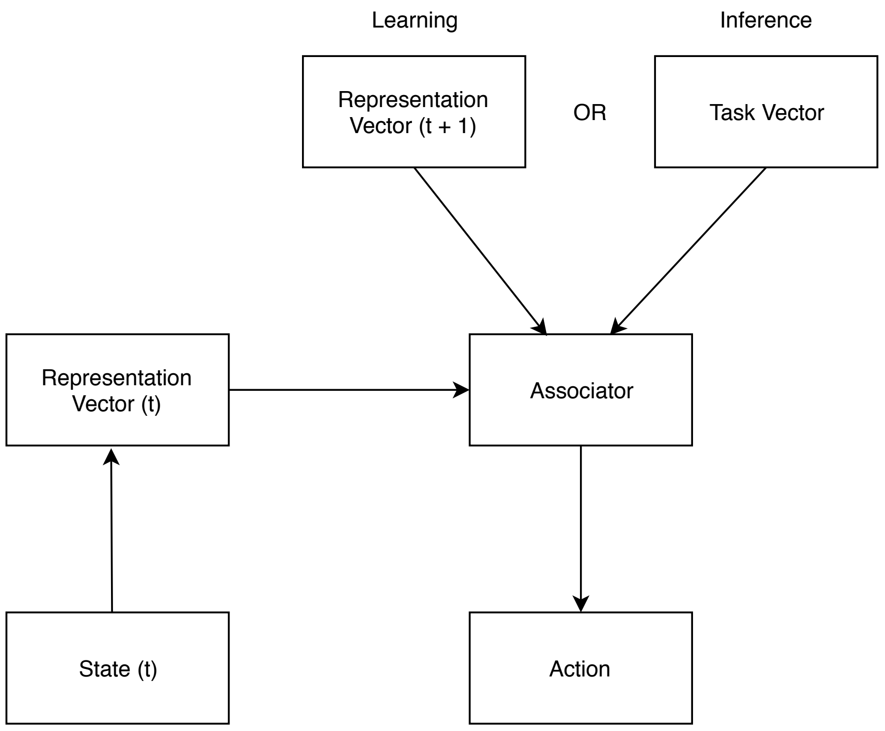

Figure 2.4: Unsupervised Behavioral Learning

Without exponential memory, this method only works for 1-step transitions. However, with exponential memory, we can set goals for policies that span several timesteps. As a user of this system, with only do three things: Provide data from the environment together with the last action taken, set the highest layer goal state (feedback) F, and read out the actions.

2.10.2 Agent Swarm

Another approach, which is more like classical reinforcement learning, is to replace the columns in the output CSDR of some (or all) predictors with a reinforcement learning agent. Each agent selects a single integer action in the range [0, columnSize). Each layer receives a reward (averaged over multiple timesteps for slow-clocking layers), which is used to update this swarm of agents.

For the agents themselves, we have tried several approaches. All of them involves a simple linear policy or value function.

- Q learning
- SARSA
- Actor-Critic SARSA
- Advantage Learning
- Finite horizon reward prediction

We have tried several more, but these are the approaches that seem to work best. We also find that only making the first (bottom most) layer have reinforcement learning instead of predictors performs better than having every layer perform reinforcement learning. This is due to the stability of the predictors as opposed to the reinforcement learning. First-layer reinforcement learning is sufficient in order to use all layers of the hierarchy, as the next higher (second) layer provides meaningful context as predictions to the reinforcement learner.

2.10.3 Routing

Routing is more general than reinforcement learning, but we have only implemented it thus far to perform reinforcement learning. Routing allows one to use a regular deep neural network (DNN) together with a SPH.

In routing, the SPH is unmodified, but we tack on a DNN that is "routed" through the active cells in the SPH. The DNN has the same shape as the SPH, but the SPH selects sub-networks for the DNN. This is a general method that allows us to perform supervised learning (and therefore also reinforcement learning) with SPH.

Since the DNN is sparsified, it learns online along with the SPH. Further, the DNN can actually be linear, and the system as a whole will not lose representational power. This is because the routing itself imbues the DNN with nonlinearity.

This method is useful, although generally not biologically plausible as the DNN still uses backpropagation.

We have used this approach for reinforcement learning by simply predicting Q values for multiple winner-takes-all actions.

3 Implementation

For the GPU implementation of OgmaNeo2 we use OpenCL. OpenCL can also run on the CPU as well as other devices, but we will generally refer to it as "the GPU version".

3.1 Performance Considerations

The combination of the ability to learn online/incrementally and the low clock rates of exponential memory already allow the system to operate very quickly. However, we can do better. We wish to exploit the sparsity of the model to only perform updates that are needed. For instance, during encoding and decoding, most entries that are the input to the encoder or decoder will be zero. Since the encoder and decoder perform a matrix multiplication, we can simply ignore all of these 0 elements.

In general, when performing the sparse matrix multiply, we can ignore all elements which are 0 in the input vector. This reduces the complexity of the algorithm substantially, essentially going from quadratic complexity (with respect to number of cells) to linear complexity. While not quite this extreme in practice, mostly since the sparse weight matrix is also a source of complexity reduction already, this is a highly worthwile optimization to perform.

There are two methods we know of for implementing this optimization:

1. Iterate through the input vector, find which elements are nonzero, and then propagate those to the output.
2. (CSDR only) Iterate through the output vector, and find the input columns the output cell addresses. Since we are using the CSDR format, we know that each column has exactly one active unit in it. If we represent CSDRs through lists of integers (index of the active cell in a column), we can index into the sparse weight matrix only for the active cell in the input.

Method 2 only works when using CSDRs. CSDRs are generally stored as lists of integers, where each element represents the index of the active cell in the column. If we simply maintain these indices at all times, we never have to search a column for the active cell, we always know its index. Method 2 exploits this by simply addressing the input and associated weight using only the index of the active unit.

While both methods have theoretically the same performance, we prefer method 2 despite its seemingly more complex nature. This is because it is easier to parallelize - it operates from the "perspective" of the hidden state, which is the output of the computation, meaning there is never a conflict or race condition. Method 1 on the other hand requires atomic operations since it operates from the perspective of the input cells, which are not the output of the computation. It requires the ability to asynchronously accumulate results in the hidden state, and addressing conflicts will occur in an undefined manner.

3.2 Data Structures

Originally, in order to transpose the encoder dictionary D using our original sparse matrix format, we had to perform a complex "reverse projection" algorithm. However, we have since switched to using compressed sparse row (CSR) matrices. We therefore generate the matrix shape once (on the CPU), and then all following operations occur using sparse matrix operations on the GPU (or CPU as well, if using the CPU-only version).

4 Example Usage

Some examples of SPH in action. This list is not comprehensive, but gives some general idea of what kind of projects can be done with OgmaNeo2.

4.1 Model Self-Driving Cars

One of our older demonstrations, where we modified a model car to have simple self-driving capabilities.

The unique aspect of our particular car was that all training and inference happened on-board in real-time. First, the user demonstrates driving, during which the car learns to predict the steering angle given by the user. Then, when in self-driving mode, the predicted steering angles is looped back in as input (as if it were user input data) and learning is disabled. This leads to the car driving itself.

Our first car used a stock model RC car as the basis of the project. For processing, it used a Raspberry Pi 3 model B. For the camera, we tried variations with both a Pi Camera 2 and a third-party fish-eye camera. The fish-eye camera generally performed better, as it could better see the edges of the path.

4.1.1 Micro Car 1 and 2

We later produced "Micro-SDCs" (self-driving cars). The first one used a Raspberry Pi Zero for processing. It functioned similarly to the larger model, but was able to fit on a toy track intended for very small (5cm long) cars. The Raspberry Pi Zero was especially limited in terms of processing power, and ran at around 10fps.

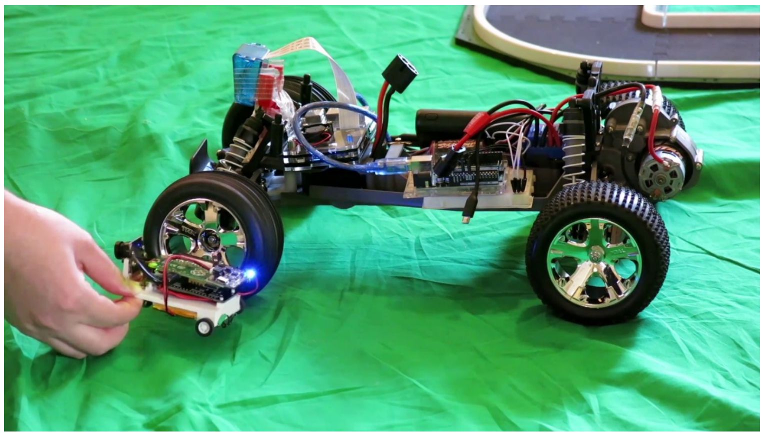

Figure 4.1: Original, larger SDC and the first Micro-SDC

We later created a second version with a NanoPi Duo. The NanoPi Duo is not only smaller than a Pi Zero, but also is vastly more powerful in terms of both memory and processing power. With this computer, we could get back up to 60fps. It also was able to perform some other "user mimicry" tasks, aside from following paths, such as returning to known locations.

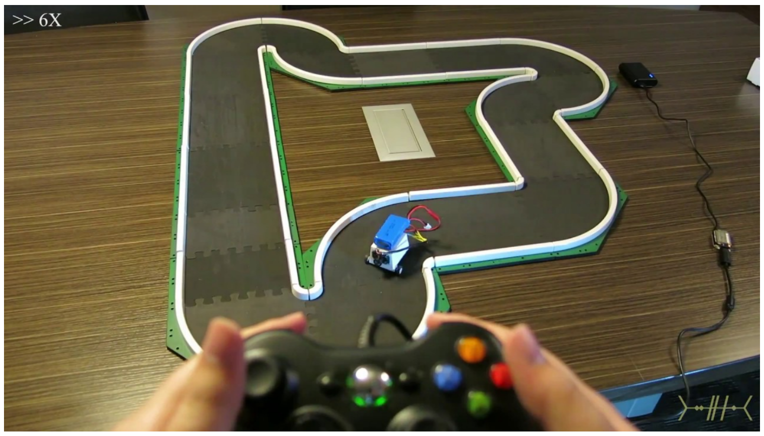

Figure 4.2: Micro-SDC 2

4.2 Reinforcement Learning

We ran the reinforcement learning versions of OgmaNeo2 on several reinforcement learning tasks - many from the OpenAI Gym.

4.2.1 Lunar Lander

The Lunar Lander environment involves a landing module which the agent must land on a landing pad in the center of the screen. The starting configuration is randomized, so the agent must learn to compensate for a variety of trajectories.