StarBoard: A Structural Stream Processing Language for

Online Learning

# Introduction

1. Machine learning is starting to become a major technological player.
2. The success and popularity of ML can largely to attributed to:
   1. Vast improvements in computational capability.
   2. Great methods of abstracting complexity in computational systems.
   3. Increasing role of computing devices in daily life.
   4. The ongoing information revolution.
   5. Large scale adoption of automation.
   6. Exploration of non-symbolic, highly parameterized learnable functions.
   7. Theoretical research in probability, statistics and analysis.
   8. Findings from psychological and neurological research.
   9. Intelligent engineering choices deviating from biological models.
   10. Composability of ML models and algorithms.
3. Predominant practice: Collect data, Design model, Train model, Iterate, Deploy.
4. Works within a problem setting distinct from the traditional CS problem setting.
   1. Traditional CS problem setting:

Given a description of a computational framework and a list of operations with certain constraints on their behavior with respect to a set of actions, find implementations of the operations satisfying the constraints, while maximizing a desirable property of the implementation if multiple solutions exist.

* 1. Machine learning problem setting:

Given an external process that generates input data, and an external process that assigns scores to output data, create a system that utilizes a sample stream of input data and a proxy metric related to the true generating and scoring processes, to eventually produce output data such that the score of the data is maximized under the external scoring process.

# Major Challenges

1. The external generating and scoring processes typically both exist outside a computational system, so their throughput and quality does not improve with technological advancement. Therefore, the sample data stream is not made out of on-demand observations from the external generating process; part or all of this data stream is created ahead of time.
2. Higher order information such as gradients cannot be computed for the external processes.
3. It can be difficult to generalize the capabilities of machine learning techniques from well-understood stand-in data and metrics to poorly-understood generating and scoring processes.
4. The iteration step of the dominant practice relies heavily on experimentation and experience. It is difficult to create a well-performing solution from inspection without comparison between multiple models or direct experience with the external processes (or their stand-ins).
5. Many solutions assume that the generating and scoring processes do not change over time. Indeed, the predominant practice freezes a trained system after the iteration step. Even with perpetual training on an infinite stream of input data, the behavior of many systems can change in irreversible ways over time, which affect their ability to adapt to changing data.
6. The existence of the training step hinders patching and updating. Current models are treated like black boxes after training, therefore architectural updates cannot be made on a deployed system without a retraining step and a total replacement of the old system.
7. The training process forces tight coupling between components of a system, such that components of a trained model depends strongly on the exact behavior of other components. Updating one component requires retraining numerous other components.
8. Rare events of high importance may exist in the generation process, which might not appear in the sample data. Even with a regular process of total system updates, a plan must be made to detect and store rare events during deployment, and represent these events in the sample data when training all future versions of the system.
9. Systems performing separate tasks are isolated from one another. In theory, several systems can cooperatively build a shared abstract knowledge base about the world. However current methods do not do this, so multiple systems need to independently learn their own representations of this abstract knowledge base, likely with much lower quality.

# Organizing Principles for a New Methodology

It is proposed that a hypothetical new methodology (hereby referred to as M) can be created to further advance the field of machine learning and address many of the challenges current techniques face. Care must be taken to ensure that M does not undo the many factors that have contributed to the success of machine learning. The above sections can be used as a reference to define several organizing principles of M, which will help guide its future design and implementation.

1. As per 1.2.a), M must be able to take advantage of our computational resources. Therefore, its design cannot fundamentally limit its performance. In the current era, this implies that systems created out of M must be able to make scalable use of parallel processing.
2. As per 1.2.b), M cannot undo the progress that has been made in abstracting system complexity. This means M must be able to take advantage of mature libraries, frameworks and programming languages, lest the users of M be forced to unwrap layers of abstraction to implement a feature.
3. As per 1.2.e), M must be well suited for automatic operation.
4. As per 1.2.f), it seems reasonable to specialize M for highly parameterized non-symbolic learning algorithms. In fact, as per 1.2.g) M should build on top of the many existing machine learning techniques that have a strong basis in mathematical and statistical research.
5. As per 1.2.i), M must allow users to experiment with many engineering choices. In other words, M should not unnecessarily constrain the family of solutions that can be created from it.
6. As per 1.2.j), components and features of M should be highly composable.
7. M is not forced to follow the predominant practice shown in 1.3. It must be capable of working within the problem setting detailed in 1.4.b), though M does not have to completely limit itself to said problem setting.
8. To help with 2.3, M should weaken the barrier between external processes and the sample data / proxy metrics used by a system, though to avoid worsening 2.1 it still must be able to handle surrogate data that has been collected ahead of time.
9. To help with 2.4, M must make it possible to create a system that eventually performs well on a ML problem with much less experimentation and domain-specific experience.
10. To help with 2.5, systems made in M must be able to perform well when used on external processes that change over time, with minimal effects from irreversible internal changes.
11. To help with 2.6, M should make it easier to create patchable systems that do not require explicit retraining or total system replacement when updated.
12. To help with 2.7, M should try to reduce the extent of coupling between trained components.
13. To help with 2.8, systems in M should be able to hold important pieces of input data for long periods of time, in order to avoid regressing to a state where such data is handled improperly.
14. To help with 2.9, M should facilitate the cooperation of systems that perform distinct tasks, such that abstract knowledge can be shared between systems despite their differences in purpose.

# High Level Design

Considering the organizing principles, it’s clear that M should make use of online learning. Online learning refers to systems that run continuously and do not have distinct training and deployment stages. If care is taken to ensure that an online learning system does not undergo irreversible changes, it should be able to satisfy principle 3.10 as long as the system can receive and process data from the external generating process (more discussion on this later).

To satisfy principle 3.4, we need to avoid creating a whole new family of online algorithms that directly compete against existing techniques. The best approach is to design methods that will help adapt current machine learning techniques to work in an online learning setting. In particular, M should advocate the use of general purpose wrappers that convert algorithms optimized for batch processing into online learning algorithms.

This wrapper approach also helps promote modularity and the exploration of engineering choices, since users of M can test new learning algorithms without worrying about their behavior under online learning environments, or create new wrappers that improve the responsiveness of existing learning algorithms without worrying about implementation details. This works toward principles 3.5 and 3.6.

Machine learning algorithms are only required to optimize on the parameter level, and do not necessarily optimize on the model level. There is no fundamental reason why a machine learning algorithm cannot use data to inform its choice of model architecture or hyperparameters. The main reason why this is usually frowned upon though is because every model change will induce a need for retraining - a manifestation of issue 2.6. The problem is compounded due to issue 2.7, where a model change in one component of a system requires retraining on many other components. Online learning can alleviate this issue, since all affected components will automatically update after the change anyway by the very nature of the training mechanism. Online learning or not, the impact of the model update can be minimized by pretraining the updated model on the behavior of its predecessor. Online learning, however, allows the full process of pretraining, replacement and retraining occur seamlessly without interrupting the rest of the system. If M is powerful enough to realize such functionality, then many of the model choice and hyperparameter tuning experiments can occur automatically, allowing it to satisfy principle 3.9 and make progress toward principle 3.3.

Building on this robustness to model updates, an online learning system is also capable of receiving incremental patches without total system replacement. If needed, a fine-grained parallel adoption system can be used to minimize the impact of the patch, essentially by running the modified component alongside its older version before ultimately phasing out the older variant. In addition, unmodified components of the system can remain idle while its dependencies are changed or replaced, and after the change the other components will automatically update to accommodate the updates. In doing so, M can satisfy principles 3.11 and 3.12.

Knowing that the training process continues forever, an online learning system can potentially utilize the external generating process to create new samples to mix into its sample data stream. There are difficulties that must be overcome to do this, however. In many machine learning problems, the input data contains labels that are required by the proxy metric to produce a score. The system not only would have to learn how to produce outputs that score well with the proxy metric; it would also need to have a second input stream containing feedback related to the external scoring process, which can be used to deduce labels or otherwise update its behavior to improve its performance. Though this approach only makes sense in a different problem setting, M can still be made compatible with this new setting according to principle 3.7. Semi-supervised methods can also be used, where the system that has high confidence in one of its outputs can assert that output to be the ground truth. Both strategies should be implementable in M to satisfy principles 3.8. Note that even without this capability, an online system is still useful if there are external agents who can inject new labeled training data, which may become increasing available due to 1.2.d) and would help the system satisfy 3.13. Relying less on such external agents will then work toward principle 3.3.

Much of machine learning lends itself well to data-parallel stream processing, which can be done very efficiently in parallel and allows for pipelining in a computation that contains many stages. However, machine learning also depends heavily on data mutation, which clashes against stream processing’s assumption that all computation is performed with stateless functional kernels. The result is that machine learning tends to alternate between a pure computational phase that computes a number of important values, and a mutative update phase which uses the computed values to improve the model. In this approach, stream processing must first clear the pipeline and run to completion in the pure phase before the mutative phase can run, which reduces the performance benefits. To satisfy principle 3.1, a looser variant of stream processing can be proposed where components of a system are allowed to be updated at any point during the computation, allowing stream processing to continue indefinitely. This change to the computational model is acceptable as long as individual updates do not significantly affect the behavior of a component, though it could make the system non-deterministic.

There is a second, more subtle issue however. Machine learning components often depend on values produced by other components in order to perform an update. Due to the tight coupling of components mentioned in 2.7, components often need to update in unison before they are allowed to process more inputs. Even if components can update independently, the dependencies may force them all to update at the rate of the slowest component. M should facilitate the creation of systems with more loosely coupled components, not only because it would work toward principle 3.12 but also because this will reduce the performance penalty of the dependency effect. The decoupling must be done in a way that doesn’t hinder the expressiveness of M too much, in order to satisfy principle 3.5.

When it’s possible to modularize machine learning systems into loosely coupled components, then it will be easier to abstract groups of simpler components into individual, more advanced components. This would work toward principle 3.6. In addition, modularization lends well to task specialization. With a standardized interface, one can use different specialized programming tools to implement the central features of each component. Therefore, M should allow its components to offload its computation to external handlers as to satisfy principle 3.2. Lastly, decoupling allows components to communicate with a variety of very different other components. M can allow big systems to use this capability to create shared abstract knowledge bases in order satisfy principle 3.14.

# StarBoard Design

StarBoard is an experimental framework designed to work under methodology M. At the moment, the below list describes the overall design of StarBoard.

* The behavior of a system is specified with a hierarchical structural design. The system is a single module that contains input and output ports, made out of other modules.
* The structures of the higher level modules are specified in a unique StarBoard module description language (SBDL). The description specifies ports, interior modules, connections, and the behavior of the module upon each data receipt event.
* The StarBoard compiler (SBC) parses SBDL to create an intermediate representation (IL). The StarBoard runtime (SBR) reads the IL to create one or more worker processes (WP), which execute the logic specified in the module descriptions.
* The lowest level leaf modules can be created in any language. To do this, a module environment process (MEP) is created in that language, which instantiates modules written in the same language and handles communication between it and the WP’s.
* Modules written in a small subset of languages can be specified as inlineable. If so, the logic of the module is executed by the WP directly, without needing inter-process communication between a WP and an MEP.
* When data is to be communicated between two modules within an MEP, an optimization can be used to reduce communication costs.
* Ports come in two forms. Message ports transfer discrete messages with some value as payload, and continuous ports constantly output a value at all times. Each module contains a distinct message receipt handler for each input message port, and a data source for each output continuous port.
* Output message ports cannot act without a corresponding event from an input message port. Inputs can be periodic timer pulses if autonomous behavior is needed.
* Each message port behaves as if it contains a finite length buffer. The buffering behavior of a message port is implementation defined, and is not specified in the SBDL. Message ports may do one of the following when attempting to add a message when its buffer is full:
  + Stall the module that is trying to output the message until the buffer is no longer full.
  + Replace an existing message in the buffer.

Note that the message itself cannot be discarded.

* An output continuous port can display a different value from a corresponding input continuous port. However, the input continuous port must show a value from the output port taken from a bounded amount of time in the past.
* Each module can periodically save its state. There is a mechanism to load checkpoints for all modules in a system, as a well as a mechanism to modify select modules upon loading.
* Raw data types include unit, boolean tensors, float tensors, integer tensors, lengths, indices, tuples, arrays and interfaces.
  + Each dimension of a tensor type may either have a fixed size or a variable size. Modules can be polymorphic on the length of the tensor for fixed sized dimensions.
  + Lengths are integers, and indices contain both a length and an integer. They are used to create / access arrays, and cannot be used for other purposes.
  + Tuples can contain any types, but have compile-time fixed structure.
  + Arrays have any number of elements of the same type, but cannot be resized after creation.
  + Custom hierarchies of type-safe aliases can be defined for any of the given types. Polymorphic wrapper types are also possible.
  + The SBC runs type-checking before creating the IL.
* A set of ports from a single module can be considered as a single interface. All connections between modules are specified via pairs of interfaces.
* When sending a message, a module associates it with a return address - an input message port from the same module. If the message is not expected to return, the return address is void. If the return address is not void, the module can choose to add tag-along data if it wishes. The tag-along data can only be read from the return address.
* Input ports are also annotated with a forward list - a list of possible output ports for an incoming message. If the input message is to be consumed, then the forward list is empty. The SBC uses this to ensure that messages with a specified return address will eventually reach the return address (provided that it isn’t lost in the process).
* A module can be specified to be closely coupled. If so, all of its component modules must run in the same MEP. Messages passed between closely coupled modules can contain any data that can be defined in the language in which the MEP is written (or perhaps tuples or arrays containing elements of arbitrary types). This data cannot be accessed outside. It is also possible to force internal message ports to use the blocking wait method, or force internal continuous ports to always read the latest value.
* A module can send interface templates through messages. There are two types of such messages. One contains calls for a new module to be created. The other is used to reference an existing module. A module receiving the interface template can instantiate it, thereby creating an external interface linked to a module (which may have been newly created). This external interface can be thought of as an interior module.
* Messages sent through an external interface cannot have a void return address. This is needed to ensure that messages sent through external interfaces cannot leave the module permanently.
* External interfaces can either be singular or list-like. Singular external interfaces begin with ports to nowhere, but the interface can be replaced at any time. A specific position in a list-like external interface can be accessed via an index. List-like external interfaces contain a continuous port showing the length, and all received values from them contain a tag-along showing the index.

# StarBoard Example

Here’s an overview of an example system, and how it could be implemented in StarBoard. This system takes in audio data from three channels and can perform two tasks. One channel contains data with sub-phoneme labels for each frame. The other two are unlabeled. The first function of the system is it can learn a classifier from frames to sub-phonemes. The second is that it can take in audio signals from a voice donor and a speech donor (the other two channels), and produce new speech whose words matches the speech donor but voice characteristics match the voice donor. The two tasks complement one another, demonstrating the cooperation touched on in 3.14.

## Converging Online Mean

We create a module that takes in i.i.d. input data. It finds the mean of the input data and typically increases the sample size by one (and displays both values in a continuous port). When the input data changes distribution, the sample size goes down. In this way, the output mean value converges toward the population mean and the sample size tends to infinity so long as the input distribution doesn’t change.

## Data Buffer

A second module is used to store incoming data into a buffer, and randomly samples from the buffer when another module needs the data. The decay behavior of the samples in the buffer can be engineered so that the distribution of age vs. frequency can follow a number of different distributions. For example, we can make a fat-tailed distribution where a large number of old samples can be found.

## Classifier

A closely coupled module contains the modules needed to train the classifier. Ports are available for training and inference. There will be a module that makes non-local updates to the various internal components, but this is allowed within the closely coupled module. An extra value continuous input port is used to tune the training behavior of the module - the higher this input value, the more fine-tuned the training will be. This input essentially behaves like a time stamp; the module acts as if it’s approaching convergence when the input is large.

## VAE

A similar approach can be used to create a VAE. Ports are available for training, encoding and decoding. Though much of the data that flows through this module will be unreadable from the outside, ports can be included to translate the data back into readable forms. The unreadable forms can then be optimized for GPU training, or just simply chosen to match an existing framework.

## Assembly

Data from the labeled channel enters a data buffer. Data from the buffer is extracted to train a classifier. Data from all three channels enter a second buffer, sans labels. A VAE is trained on data from this second buffer. In both cases, a converging online mean takes in the data from the buffers and produces a sample size value, which is passed into the classifier and VAE as a time stamp value.

The classifier is used to classify each frame from both the voice donor and the speech donor. Online converging mean modules use the labeled data buffer to find the mean encoding for the frames separately for each of the possible labels (either ground truth or from our classification). We subtract the label-specific mean encoding from the encoded frames of the voice donor, and add the label-specific mean encoding from the labels computed for the speech donor. The resulting stream of frames is sent out of the system (possibly to some voice synthesizer), and is added to the labeled data buffer.

# A few loose ideas

## Clustering

We can implement a GMM module, where “soft assignments” of input data points to clusters can be computed, and the cluster means and covariances can be updated using converging online mean modules. This acts like an online version of the EM algorithm.

## Grid search

We can create a module that links to three other modules with different hyperparameters, specifically three consecutive values of a monotonically increasing sequence. The middle module is the active one, but we can accept performance measurements from all three. When one of the two is statistically significantly better than the middle, then the group of three modules shifts over by one.

Extra logic can be added to deal with cases where all three modules perform equally poorly, or to explicitly measure when a newly added module is approaching convergence (e.g. statistically significantly behaves the same as an older version).

This can also generalize to sequences of models, ranging from a family of high bias models to a family of high variance models.

## Structural modifications

A module can be created for a neural network, which includes ports for adding new layers or changing the size of one of the layers. Newly added layers can be designed to initially act like the identity function, and new nodes in a layer can be designed to initially cause minimal effect to the behavior of the network.

## Shifting Experts

A module can be created where given an input, it finds a probability distribution over several models. The distribution is trained to maximize the performance of the overall system, where the predicted distribution is used to choose the model that will actually handle the input. This can be done with an online learning system and an EM-like algorithm even when the behavior of the models is constantly evolving.

## N choose K functions

A module can be created so that it makes predictions for N values given an arbitrary subset of K values. This can be implemented as a collection of one-way networks that depend on one another during training. For example, a 2 choose 1 network can be thought of as a pair of functions that are trained to be inverses of one another, such as an auto-encoder.

N choose K functions are well suited for data generation. A passive process of learning a function can be turned into a method of generating novel data from internally generated labels, encodings or other simpler forms. This opens the function to be used by other systems. For example, a speech recognition function can be adapted into a 2 choose 1 function, and a separate textual language model can use the backward direction to vocalize some internally generated text.

# Implementation Plan

1. Design a subset of the IL, which allows for a simpler language runtime for creating the WP’s. This subset assumes that only one WP exists and only can specify statically defined systems with no external interfaces.
2. Design the communication interface between WP’s and MEP’s.
3. Implement the runtime for the subset of the IL.
4. Test a few static systems with hand-crafted IL representations.
5. Design a subset of SBDL, leaving out aliases and static analysis.
6. Implement the SBC for the SBDL subset.
7. Test the systems in 4.) through compilation with the SBC.
8. Expand SBDL to support type aliases / return addresses / forward lists, and implement static analysis in the SBC. Polymorphism can be left out temporarily.
9. Add support in the IL for external interfaces.
10. Test a few dynamic systems with hand-crafted IL representations.
11. Expand SBDL to support external interfaces.
12. Test the systems in 10.) through compilation with the SBC.
13. Expand SBDL to support polymorphic types and modules.
14. Implementing inlining and communication between multiple WP’s.
15. Create a library of useful modules and experiment with various designs.