Engineering A Scalable Machine Learning Framework

Stephen Elliott[1]

[1]**Student at UNSW, Sydney**

ABSTRACT

This article explores the design and implementation of efficient, scalable frameworks for modern machine learning workflows. Key topics include the mathematical structure of neural networks, their optimisation via gradient descent, and the translation of these models into computational graphs for hardware acceleration. It analyses the backends of the industrial machine learning frameworks PyTorch, TensorFlow, and Horovod. It discusses OpenCL as a low-level framework for parallel computing and introduces the author's Catalyst machine learning library, a hardware-agnostic reimplementation of PyTorch. Catalyst's design emphasises modularity, extensibility, and integration with distributed computing frameworks. The article outlines plans for improvements to Catalyst and the development of Bonzo-HPC, a complementary library for distributed computing with Catalyst.

Keywords: Machine Learning, Deep Learning, High Performance Computing, Distributed Computing

INTRODUCTION

Machine learning is a field of computer science which studies the theory and practice of empirically calibrated pattern recognition algorithms. Deep neural networks form the backbone of many recent advances in the field, including in image segmentation (29), computational biology model (16), and natural language processing (36). The most popular model architecture at present is the multi-head attention transformer, which has been successfully applied to language and vision tasks (5).

Calibrating models involves iteratively optimising non-convex target functions, which is computationally intensive. Therefore, successful machine learning architectures are heavily informed by hardware considerations (11). This is true for neural networks, which can be represented, trained and accelerated easily on modern hardware. It is also true for transformers, which are designed specifically to exploit hardware parallelism (36).

Computational cost is a constraint on training ultra-large models, affecting convergence and therefore model quality. Significant advancements in transformer architecture have come from hardware-aware algorithms. FlashAttention implements efficient GPU memory access patterns to triple transformer execution speed (4). Ring Attention uses block-wise execution and modifies inter-device communication in distributed hardware setups to scale up context length, with no extra compute (20). These advances emphasise the need for a strong understanding of the low-level implementation of machine learning models.

Hence, my project focuses on the design and implementation of a modern machine learning framework (Stephen Elliott / one-2). Initially, I wanted to build a framework and train a functioning generative pre-trained transformer (GPT) with it. I researched transformers and ML libraries during the term 2 holidays. I spent the first weeks of term 3 learning the basics of C++, and cemented my knowledge implementing its public interface. I named the framework Catalyst to represent the accelerating effect I hoped it would bring to my learning. Indeed, it has dramatically boosted my understanding of computational algorithms, architectures, and high-performance machine learning backends. It has ignited a keen curiosity to learn more about these technologies.

As I implemented my backend design, however, I realised it was hopelessly minimal, missing many key components. My design doubled or tripled in size as I came to a fuller understanding of the requirements for an extensible and scalable system. The current design is reasonably well-specified, but is only partially implemented, due to this unexpected explosion in complexity. I will finish implementing the backend during the summer holidays.

Furthermore, as my understanding has grown, I have come to understand that a multi-device setup will be necessary to train even a simple single-attention-head, decoder-only language model at home. My goal has expanded over the course of the project, from a simple GPT implementation to a focus on a deep understanding of core low-level ML implementations. Hence, I have studied distributed and parallel computing theory, and designed a high-performance computing library. I will implement this library after Catalyst, and finally use the pair to train a GPT, from scratch, on home hardware, as originally intended.

This report will examine how neural networks work, and how their mathematical expression is translated to efficient hardware implementations. We will discuss the theory of distributed computing and how hardware acceleration is used to speed up model execution. We will discuss the ML frameworks PyTorch and TensorFlow, and the distributed computing library Horovod. We will examine relevant aspects of the author's Catalyst ML implementation, and future plans for Catalyst and the Bonzo high-performance computing implementation.

NEURAL NETWORKS

0.1 Neural Network Model Structure

Deep learning refers to machine learning using the deep neural network model architecture. Deep neural networks are highly flexible, piecewise nonlinear function approximators. They can be represented as a directed acyclic graph (DAG) G in Figure 1.

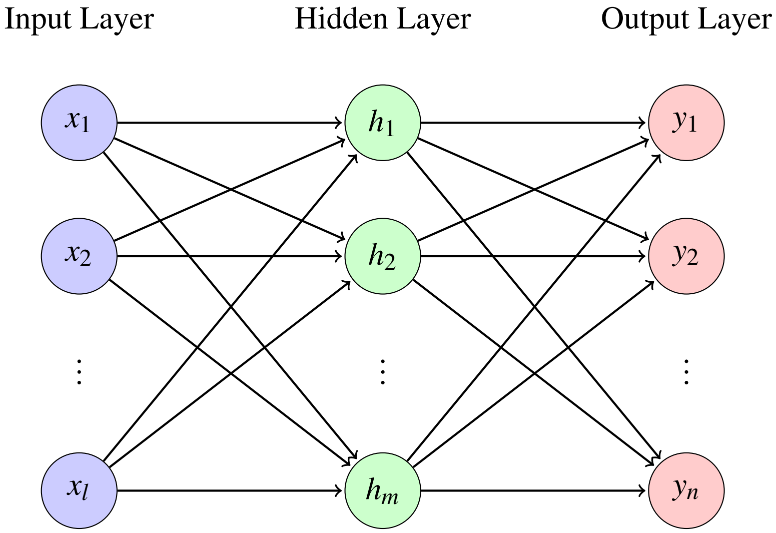

Figure 1: A DAG representation of a neural network with one hidden layer.

The input layer takes an observation set of different input variables x_1, x_2, and so on. A hidden layer receives the inputs from the previous layer, sums them, and transforms them. The transformations at the hidden layers are learned. The output layer receives the outputs of the previous layer, sums them, and transforms them to a desired unit.

The structure of each node is more apparent from the equivalent functional representation of the network:

$$h_j = \sigma\left(\sum_{i=1}^{n} w_{ij}x_i + b_j\right), \quad j = 1, \dots, n$$

$$y_k = \text{out}\left(\sum_{j=1}^{n} v_{jk}h_j + c_k\right), \quad k = 1, \dots, m.$$

The hidden node h_j computes the activation function σ of the sum of weighted values $w_{ij}x_i$ from the previous layer. A separate weight is applied to each incoming value; that is, each node computes a different weighted sum of the incoming values, $w_{ij}x_i$. However, all nodes in a layer share the same bias term b_j, which is added to the weighted sum of inputs. This linear transform of inputs is then ”activated” by the function σ (10, p.394).

0.2 Non-linearity in Neural Networks

The activation function σ is non-linear, introducing complexity into the model. Commonly used σ functions, the rectified linear unit (ReLU) and the hyperbolic tangent (tanh) are plotted in Figure 2 below. The ReLU has a simple derivative, training less expensive, as will become apparent in the section on backpropagation; whereas the tanh function is a more traditional choice, rooted in neuroscience.

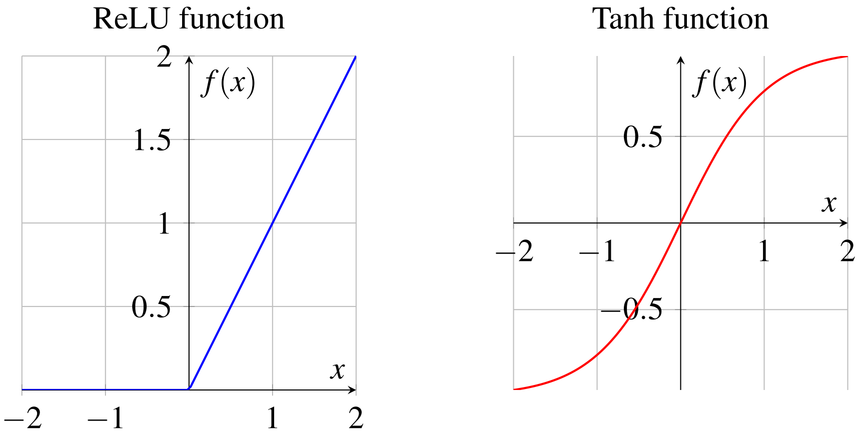

Figure 2: Common activation functions.

In any case, activation functions have a ”switching” effect. If the sum of weighted inputs and bias is less than zero, ReLU will send an activation of zero to the next layer, effectively ”switching off” the linear transformation performed at the node. If the sum of weighted inputs and bias is greater than zero, the node will ”switch on”, sending the value of the processed inputs to the next layer. This complexity is the enhancement of a neural network over a linear model. While each node indeed computes a linear regression on its inputs, the activation function switches (ReLU) or scales (tanh) the linear regression according to its prediction for the current input value.

The output of a single ReLU-activated hidden layer is shown in Figure 3. It is a piecewise linear function, with blue segments representing the idealised scenario where only one node is active at a time. These linear segments are divided by thresholds, where the weighted sum of inputs for one node drops below zero, while another node's weighted sum rises above zero. At these points, the nodes ”swap.”

The red segment illustrates the more realistic, though less intuitive, case, where the output function becomes curved. This curvature emerges when multiple nodes are active simultaneously, with their activations changing at different rates.

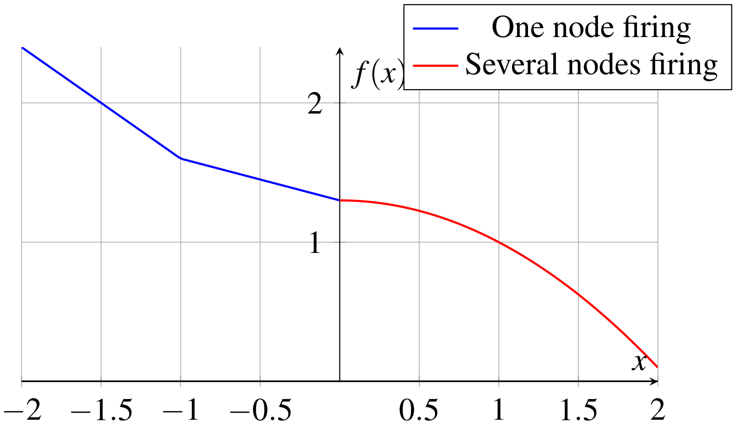

Figure 3: Output of a neural network with a single hidden layer and ReLU activation.

Non-linear activation functions dramatically increase the representational capacity of what would otherwise be no different to a linear regression model. With a linear activation function, the model will always be a linear regression, no matter its width or depth. However, with non-linear activations, the class of functions we can represent expands dramatically.

Deeper networks are exponentially more expressive than shallow networks (26), further enlarging the class of functions we can model using neural networks.

0.3 Calibrating Neural Network Models By Iterative Optimisation

0.3.1 Online Gradient Descent

Critical to the utility of our model is the observed non-linearity in output comes from trained linear transformations. Model weights can therefore be selected such that the sequence of operations in the model transforms inputs into a useful prediction.

Neural network parameters are selected by iterative optimisation. We define the target function for our optimiser to be the generic loss function,

$$\mathscr{L} = \ell\left(y_i, f(x_i; \mathbf{w})\right).$$

We have i as the index of the input-output pair we are evaluating, and $\mathbf{w}$ is the matrix of model weights. The semicolon notation specifies input variables x_i on the left, followed by trainable parameters $\mathbf{w}$. The target function is a function of the predicted output which measures the error our model made in its prediction. In a regression task, for example, squared error is a typical choice:

$$\mathscr{L} = \frac{1}{2}\left(y_i - f(x_i; \mathbf{w})\right)^2.$$

The half term is included for convenience when working with the derivative of this loss, which is important for the computational implementation discussed in the backpropagation section.

The loss target function is optimised by gradient descent. For the training example i, we first perform a forward pass of the inputs through the model's transformations, producing an output prediction $f(x_i; \mathbf{w}_i)$. We then calculate the gradient of the loss with respect to each of the model's parameters. We adjust each parameter according to its contribution to the loss, as reflected by the partial derivative of the loss with respect to that parameter. The parameter update rule can be expressed as

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}^{(t)}),$$

where η is a rate parameter, which scales down parameter adjustments to allow convergence of the model.

For example, if the gradient of the target function is negative with respect to some parameter, increasing that parameter value will reduce the loss. Conversely, if the gradient is positive, decreasing that parameter's value will reduce the loss. This process of adjustment edges the model's parameters towards an overall transformation which accurately reproduces the labeled output data from the associated inputs.

Values of the learning rate vary with the data characteristics, but 0.01 is a typical starting point. Sophisticated tuning techniques exists for the learning rate parameter, such as Adaptive Method of Moments (ADAM); however, this technique has a trivial implementation over and above this simple technique, so we will not discuss it here. We instead focus on methods which have strong educational value in a low-level implementation.

The complete online (single datum) gradient descent algorithm is specified as Algorithm 1 below:

Algorithm 1 Online Gradient Descent Algorithm

```
Initialise model parameters $\mathbf{w}_i$ (typically randomly).
Initialise a learning rate $\eta$.
repeat
    for each training example $(x_i, y_i)$ do
        Perform a forward pass to compute the model's output: $f(x_i; \mathbf{w}^{(t)})$.
        Calculate the loss: $\mathscr{L}(y_i, f(x_i; \mathbf{w}^{(t)}))$.
        Compute the gradient of the loss with respect to each weight: $\nabla_{\mathbf{w}}\mathscr{L}$.
        Update the parameters:
            $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}}\mathscr{L}$.
    end for
until convergence or for a fixed number of iterations.
Output the optimised parameters $\mathbf{w}$ and $\mathbf{b}$.
```

Gradient descent performs well in practice, though it requires careful tuning to navigate complex loss landscapes. There are many methods to measure and encourage convergence to useful parameters, whereby the model will provide good prediction accuracy on unseen data. However, these are secondary to the purpose of the project, so we will not consider them in detail.

0.3.2 Gradient Descent by Backpropagation of Loss

The algorithm presented above is a compelling solution to our problem of weight updates with a simple mathematical implementation, though we still have not provided a concrete method of computing the required derivatives. Consider a functional representation of a two layer network, with one node at each layer:

$$f(x; w_1, b_1, w_2, b_2) = w_2 \cdot \sigma(w_1 \cdot x + b_1) + b_2$$

Let the loss function be squared loss,

$$\mathscr{L}(y, f(x)) = \frac{1}{2}\left(y - f(x; w_1, b_1, w_2, b_2)\right)^2.$$

The loss function depends on nested functions of the inputs. Hence, using the chain rule of calculus, we can compute the derivative of the loss function with respect to any of its dependencies (10; 27). We simply follow the dependencies backwards from the output. Then the gradients of our loss function, with respect to the trainable parameters, are

$$
\begin{aligned}
\frac{\partial \mathscr{L}}{\partial w_2} &= -\left(y - f(x; w_1, b_1, w_2, b_2)\right) \cdot \sigma(w_1 \cdot x + b_1), \\
\frac{\partial \mathscr{L}}{\partial b_2} &= -\left(y - f(x; w_1, b_1, w_2, b_2)\right), \\
\frac{\partial \mathscr{L}}{\partial w_1} &= -\left(y - f(x; w_1, b_1, w_2, b_2)\right) \cdot w_2 \cdot \sigma'(w_1 \cdot x + b_1) \cdot x, \\
\frac{\partial \mathscr{L}}{\partial b_1} &= -\left(y - f(x; w_1, b_1, w_2, b_2)\right) \cdot w_2 \cdot \sigma'(w_1 \cdot x + b_1).
\end{aligned}
$$

Using ReLU activations, these partial derivatives simplify to

$$
\begin{aligned}
\frac{\partial \mathscr{L}}{\partial w_2} &= -\left(y - (w_2 \cdot \max(0, w_1 \cdot x + b_1) + b_2)\right) \cdot \max(0, w_1 \cdot x + b_1), \\
\frac{\partial \mathscr{L}}{\partial b_2} &= -\left(y - (w_2 \cdot \max(0, w_1 \cdot x + b_1) + b_2)\right), \\
\frac{\partial \mathscr{L}}{\partial w_1} &= -\left(y - (w_2 \cdot \max(0, w_1 \cdot x + b_1) + b_2)\right) \cdot w_2 \cdot \mathbb{1}_{w_1 \cdot x + b_1 > 0} \cdot x, \\
\frac{\partial \mathscr{L}}{\partial b_1} &= -\left(y - (w_2 \cdot \max(0, w_1 \cdot x + b_1) + b_2)\right) \cdot w_2 \cdot \mathbb{1}_{w_1 \cdot x + b_1 > 0},
\end{aligned}
$$

where $\mathbb{1}$ is the indicator function. These functions are obviously trivial to compute. Hence, in gradient descent optimisation with backpropagation of loss, we have a method of fitting neural network parameters which is both mathematically justified and computationally efficient. A pseudocode representation of the gradient descent by backpropagation of loss algorithm follows as Algorithm 2 overleaf.

0.3.3 Backpropagation as Automatic Differentiation

Backpropagation is a type of computed differentiation called automatic differentiation (Autodiff). Autodiff involves explicitly specifying derivatives of functions before execution. Derivatives are chained together at runtime using the chain rule of calculus. Autodiff is fast, since chained derivatives are formed simply as a composition of predefined operators; and precise, since its form is specified directly from mathematical differentiation. Autodiff is amenable to optimisation by operator fusion ((2)), which we discuss under the section on computational graphs.

Other computed differentiation strategies include numerical approximation methods and symbolic methods. Numerical methods are useful when the functional form of the derivative is not known, but its complexity grows with required precision. Symbolic methods are useful when the derivatives required are not known ahead of time. Both are more computationally intensive than automatic differentiation and their advantages are not useful in the computational graph application.

0.3.4 Batched Losses and Stochastic Gradient Descent

The surface expressed by our single-datum sum-of-squares target function (used in online optimisation),

$$
\mathscr{L} = \frac{1}{2}\left(y_i - f(x_i; \mathbf{w})\right)^2,
$$

Algorithm 2 Gradient Descent with Backpropagation

```
Initialise model parameters $\mathbf{w}_i$.
Choose a learning rate $\eta$.
repeat
    For each training example $(x_i, y_i)$:
    for each training example $(x_i, y_i)$ do
        Perform a forward pass to compute the model's output: $f(x_i; \mathbf{w}^{(t)})$.
        Calculate the loss: $\mathscr{L}(y_i, f(x_i; \mathbf{w}^{(t)}))$.
        Compute the gradient of the loss with respect to each weight: $\nabla_{\mathbf{w}} \mathscr{L}$.
        Perform a backward pass to compute gradients:
            $\frac{\partial \mathscr{L}}{\partial w_{ij}} = \frac{\partial \mathscr{L}}{\partial f} \cdot \frac{\partial f}{\partial \sigma_j} \cdot \frac{\partial \sigma_j}{\partial w_{ij}},$
            $\frac{\partial \mathscr{L}}{\partial b_j} = \frac{\partial \mathscr{L}}{\partial f} \cdot \frac{\partial f}{\partial \sigma_j} \cdot \frac{\partial \sigma_j}{\partial b_j},$
    where $j$ indexes the $j$th hidden unit.
    end for
    Update the weights and biases:
        $w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathscr{L}}{\partial w_{ij}}, \quad b_j \leftarrow b_j - \eta \frac{\partial \mathscr{L}}{\partial b_j}.$
until convergence or a fixed number of iterations.
Output the optimised parameters $\mathbf{w}$ and $\mathbf{b}$.
```

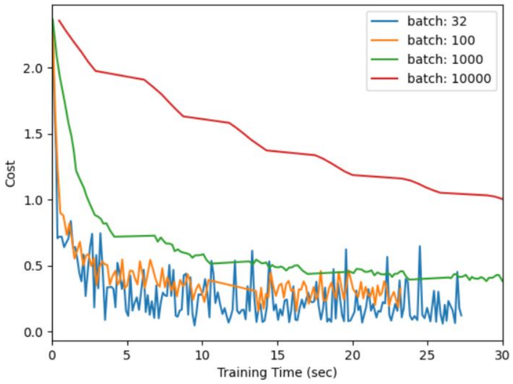

Figure 4: Losses with varying batch sizes. (1)

is likely to be highly irregular due to noise in the information generating process. Consider the neural network regression problem as an attempt to estimate the function f which maps inputs to outputs. Then the true process is

$$Y = f(X) + \varepsilon,$$

where ε represents noise in the outputs. Noise arises due both to randomness inherent to the data generating process and, in practice, to random errors in the measurement of the inputs.

The loss surface generated by online learning includes sharp changes in value, reflecting local variations in noise and signal in the inputs. A basic approach to smoothing this noise out of the loss landscape is batched learning. Batched learning is a generalisation of online learning to N data points, where N is greater than 1 and less than (mini-batch) or equal to (batch) the size of the dataset. For example, the mean squared error target function is

$$\mathscr{L} = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i; \mathbf{w}))^2 .$$

Many other batch weighting schemes exist. Mean squared error is preferred ((15)) for its simplicity and corresponding efficiency.

Optimising using batched losses results in the stochastic gradient descent algorithm, which computes a random approximation to the true least-gradient path. We wish to tune the optimiser by selecting a batch size for which the loss stabilises around a global minimum, among many other possible methods. We see in Figure 4 below that the loss (cost) target function becomes smoother as batch size increases. Small batch sizes vary with local minima, making it difficult to assess convergence. Very large batch sizes take much longer to converge, increasing training expense. Optimal batching depends on the characteristics of the dataset, and is determined empirically.

Expressing the mean-squared error loss in vector notation,

$$\mathscr{L} = \frac{1}{2N} \|\mathbf{y} - f(\mathbf{X}; \mathbf{w})\|_2^2 ,$$

we see that training on batches involves vector/matrix/tensor multiplications. Batching motivates training on vector-optimised hardware, which can execute a function on entire batches of data at once. We further discuss parallel execution strategies such as ”single instruction, multiple data” (SIMD) in the section on parallel and distributed architectures.

0.4 Neural Networks as Computational Graphs

A computational graph is a graph data structure which holds data and methods for efficient execution of neural networks. The directed acyclic graph structure is well-suited to representing neural networks, as suggested by Figure 1 above. Dependencies all flow in one direction in a neural network. Hence, a topological sort of the graph provides an execution order for forward passes of the network. Similarly, a reverse topological sort provides an execution order for the backpropagation of loss.

Furthermore, the DAG structure lends itself naturally to parallelisation of node execution. A (reverse) topological sort with pointers to the first node in each layer creates a priority queue for execution. When a layer has finished execution, all nodes on the next layer can immediately be distributed for execution, with insubstantial scheduling overhead. Each node is an object, capable of holding its weights; callbacks to its sum and activation operators; the values of its inputs; the values of its activations; and the partial derivative of its parameters with respect to the loss. The special properties of input and output nodes are easily incorporated.

Design patterns on the node allow for hardware-agnostic execution, improving potential for parallelisation. The DAG structure neatly collects everything required to execute and scale a neural network.

0.5 Industrial Implementations of Neural Networks

PyTorch and TensorFlow are the major industrial machine learning frameworks we will consider. They are highly flexible, catering to a variety of use cases and model architectures. Both offer extensive customisation. The principal difference between these frameworks lies in their approach to the computational graph.

0.5.1 PyTorch’s Dynamic Computational Graph

PyTorch’s computational graph is not compiled, so it can be updated at any time during execution. It uses a dynamic dispatch pattern to select an execution strategy at runtime (6). This flexible design allows researchers to experiment with unorthodox architectures, such as architectures which conditionally change their structure. It may also be useful for tasks where the input sequence changes length, since the graph can be rebuilt for different input widths; though this is also easily handled by zero-padding inputs in compiled graphs. This flexibility comes at the expense of optimisations which would improve the execution performance of the model. Hence, PyTorch is more popular in research and development. Production models are reconstructed in TensorFlow.

0.5.2 TensorFlow’s Compiled Computational Graph and Optimisations

TensorFlow’s computational graph is compiled by default. Graph compilation allows optimisation of the computation pattern specified by the user. The user specifies the computational graph and compiles it by calling $model.compile()$. Graph-level optimisation prior to compilation is useful because these optimisations are easily expressed at the graph level and are independent of the compilation strategy selected by the user (18, s.10). TensorFlow’s graph optimiser, Grappler, pre-processes the graph, performing

”high-level optimizations,” and, ”optimizing the mapping of graph nodes to compute resources.” Grappler optimisations occur at the level of the Python graph object (34), including:

- Constant folding and arithmetic optimiser: collects constants and simplifies arithmetic, decreasing node counts and thereby reducing scheduling overhead.

- Tensor layout optimiser: adds tensor reformat operations to improve memory access patterns for data layout-dependent nodes. Tensors are represented as sequences of values in memory, and recalled in a multidimensional matrix structure using an offset function. The offset function computes the value of the tensor at a position in n-dimensions by offsetting memory addresses from the start of the array (23). Switching offset functions (for example, from the NCHW format to the NHWC format, for 4D tensors) (18, s.21) can improve performance on some operations.

- Function and loop optimisers: inline function bodies and unrolls loops to keep more of the graph code in the graph execution code, avoiding calls to other address spaces and expensive conditional execution.

- Operator fusion: fuses common operator chains, such as $Matmul + BiasAdd + Activation$. This reduces operator scheduling overhead, improving throughput. Condensing the graph passed to the compiler also improves the compiler’s optimisations (18, s.22).

- Automatic mixed precision optimiser: implements automatic mixed precision (AMP) on the graph. AMP quantises parts of the model from the standard 32-bit types to lower-precision 16-bit types, improving throughput and decreasing memory usage (35).

- Transitive reduction: removes redundant control edges (18, p.28) to reduce redundant message passing (33) in distributed training setups.

Using Grappler on transformer models reduced node count by 55% and improves on-device processing time for an average training step (one labeled input-output pair) by 18% in published results (18, s.33).

TensorFlow compiles the simplified graph in Accelerated Linear Algebra (XLA) (18, s.9), an open-source numerical linear algebra compiler for machine learning (24). PyTorch also uses XLA, specifically to enable training on specialised tensor processing unit (TPU) hardware (28). XLA uses the LLVM compiler infrastructure to compile computational graphs to machine code (25), regardless of the target architecture (LLVM Project). XLA performs common optimisations, then optimises specifically for the target architecture, and finally generates target-specific machine code (25).

0.6 Catalyst Implementation

The Catalyst computational graph is initialised with the Model object, the core of the library’s public interface. The CGGraph itself holds an input operator object, an activation operator object, a matrix of its inputs, and a vector of its activations. The matrix and vector data structure will be illuminated in the section on batch training. Please access the link at the bibliography reference for (31) to see a working copy of the class diagram.

The CGBuilder class builds layers on the CGGraph by adding layers, which simply calls an *add_node* function on the CGGraph many times with the same specification. Layer construction is thereby abstracted away from the CGGraph, keeping the interface clean.

As nodes are added to CGGraph, they fetch callbacks from the TensorOps class. The callbacks represent mathematical operations on tensors. Tensors are generalised resizeable containers which can hold scalars, vectors, matrix, and higher-dimensional data. Using a TensorOps callback abstracts the execution strategy away from the computational graph itself.

With this design decision, the framework becomes hardware-agnostic. Tasks can be sent for execution on the CPU or GPU depending on device utilisation; or sent to a distributed computing framework for large-scale acceleration.

TensorOps takes a user configuration and decides to execute the mathematics on the CPU or GPU by creating a class implementing the TensorOpsStrategy interface, either CPUStrategy or GPUStrategy. CPU operations use third-party methods at this stage. GPU operations are manually implemented manually in OpenCL and handled by the KernelManagerSingleton.

KernelManagerSingleton parameterises, compiles, caches, and launches GPU kernels, as well as handling enqueue and dequeue of kernels to GPUs. Upon creation, the KernelManagerSingleton queries available devices and computes optimal workgroup sizes for each. Upon receiving a request to execute a kernel, the manager first checks the cache to determine whether the needed kernel has already been compiled. If it has not, the manager uses its device queries to parameterise a new kernel, compiles it, caches it, and queues it. If the kernel does exist in the cache, it queues the kernel for execution.

The Catalyst framework also has public logging and data input-output classes. The logger may interact with the backend through the core Model interface to checkpoint (back up) the model's state during training, or write diagnostic information like loss values. The logger also has some resources monitoring capability, allowing basic diagnostics on device utilisation. The input-output classes consist of Dataset and Dataloader. Dataset holds the information, while Dataloader provides an interface for the model execution process to access batches of data.

0.6.1 Future Improvements to Catalyst

I would like to implement many improvements to Catalyst. However, I must prioritise to the highest-impact options:

- Multi-device scheduler. The current design specifies a device before execution and queues all tasks onto that device. My first extension task will be to modify this design to schedule tasks across several devices as they become available. The design is extensible to accommodate this.

- Graph optimisation. The current graph is dynamic, and unoptimised. The first step will be to create a rule-based operator fusion mechanism to execute just before the graph begins to execute. Fusing common GPU operators, such as matrix multiplication and ReLU activation, could save substantial time in launching kernels, improving utilisation. The next goal is to create a simple graph optimiser, like TensorFlow's grappler, follow by a simple intermediate representation language. This will have great educational benefit, as I will be learning about compilers and ML optimisation at once. I will study PyTorch's just-in-time compiled package, Torchscript, and TensorFlow's XLA compiler for inspiration.

- Transformer architecture. Implementing a transformer involves several new components over the top of my current design. I have constructed the design to be modular, so it should accept them without much trouble. However, the new components have very different characteristics, and will require different implementations and optimisation strategies. It also will likely require multi-device training to achieve convergence, even on small models, so this may wait until after the distributed computing project. This is an especially exciting target due to transformers' recent popularity in research.

- Distributed execution of computational graphs. This is elaborated in the next section.

CONCURRENT, PARALLEL, AND DISTRIBUTED COMPUTING

0.7 Concurrency and Parallelism

Fitting deeper and wider models by iterative optimisation requires scaling up computational resource use. Concurrency describes the ability of a system to manage multiple tasks simultaneously by interleaving their execution, potentially overlapping progress on several tasks at once. Parallelism is a special case of concurrency wherein several tasks are executed at the same time. Parallel programming is essential for high-performance ML and scientific computing applications. It may be implemented as multi-threading, on-device task parallelism, or distributed computing.

Each represents an increase in computational power, but also introduces new design challenges. A core problem introduced by concurrent is memory contention, wherein several tasks access or modify the same information at once. Concurrent programs may be encapsulated and isolated from each other to address memory issues. However, to pool computing resources effectively, programs must be able to communicate, introducing new design challenges. Furthermore, errors in concurrent processing may be difficult to diagnose, due to the essentially unpredictable nature of concurrent behaviour.

0.8 Multi-Threading

Multi-threading is a basic form of multiprocessing, involving resource sharing at the processor level. Threads in a multi-threaded process share a memory space and are interleaved for execution on the same core. Each thread has its own stack. However, all threads on a process share a heap and global variables. The operating system allocates the parent process one or more processor cores for execution of its threads. Threads exist purely as subordinates to the parent process, which is ultimately controls its memory space and processor resources. The process can kill its threads, but the threads cannot kill the process (14, p.23).

Multi-threading allows several procedures to share processor resources, reducing latency on urgent processes (14, p.3). Reduced latency comes at the expense of increased software complexity from implementing thread-safe design. Unexpected behaviour arises due to concurrent memory access, and must be carefully managed using synchronisation objects.

0.8.1 Concurrent Data Access Paradigms

There are four paradigms for concurrent access to data. We have exclusive read, exclusive write (EREW); exclusive read, concurrent write (ERCW); concurrent read, exclusive write (CREW); and concurrent read, concurrent write (CRCW) (14, p.9).

When a task is initialised, the operating system adds it to a priority queue and schedules it for execution on a processor. If the operating system implements multi-threading, several tasks will share execution time on a single processor. A thread will pause execution when a higher-priority task is scheduled for execution on the same processor (14, p.45-46).

For example, threads running GUI tasks, which require low latency to ensure a good user experience, may interrupt, or preempt, a low-priority resource monitoring task. Similarly, operations waiting for data input or output may release the CPU to allow computation-heavy threads to run while the data is loaded to CPU memory, improving throughput.

0.8.2 Shared Address Spaces of Multi-Threaded Programs

Threads on the same processor share an address space. Threads may therefore access common data without overhead from inter-process communication (14, p.40). Threads can read and write to global variables, shared data structures, or to dynamically allocated memory (the heap) without overhead from communicating with other threads.

An interesting enhancement is the use of virtual RAM. Using an abstract data structure called a page table, the operating system extends addressable memory onto the hard disk. Virtual RAM thereby creates a larger address space for processes, allowing them to spawn more threads, of greater size (14, p.41).

0.8.3 Design Challenges in Multi-Threaded Programs

However, multi-threading also introduces some design challenges. Volatility issues are unexpected behavior related to concurrent access to the same memory resources. Volatility problems at the thread level include race conditions, where two threads attempt to access and modify the same data simultaneously, leading to inconsistent results. Deadlock occurs when two or more threads wait indefinitely for each other to release resources, due to a cyclic dependency between the threads. For example, Thread A may be waiting for Thread B to release, while Thread B is waiting for Thread C to begin, while Thread C is not initialised until Thread A continues execution. Finally, advanced memory access issues like unnecessary cache invalidation due to false sharing are also a concern, but outside the scope of this report (14, p.25-26).

Non-deterministic bugs, caused by unpredictable thread execution orders, may result in errors that are difficult to reproduce. I have encountered these in developing Catalyst. They are difficult to diagnose and require specialised test design to catch. Bugs may go undetected due to their unpredictable trigger conditions.

0.8.4 Robust Concurrent Programming

Robust concurrent programming can be achieved using a combination of resource-aware operations and thread-safe design. Primitive synchronisation operators, such as mutexes, control access to shared resources by allowing only one thread to read or write to a resource at a time. Atomic operators are instructions which cannot be interrupted, enforcing data consistency for operations sensitive to data volatility. Their implementation varies with processor architecture (14, p.82-84). They may implemented, for example, by blocking memory transfers during execution of the atomic operator. Lock-free data structures are a design pattern which use atomic operations instead of locks, avoiding deadlock the risk of deadlocks and reducing overhead from managing resource locks.

0.9 Multi-Processing and Distributed Computing

Multi-processing is a generalisation of multi-threaded parallel computing, where tasks are created as new processes. Each process has its own address spaces, encapsulated from other processes. This enables simultaneous execution of tasks without interruption. Multi-core architectures are common on modern hardware, and the line between on-device parallelism and inter-device parallelism is indistinct.

The concepts discussed in this section can be applied as much to multi-core processors as to large-scale distributed computing clusters. There are four paradigms of concurrent execution which are useful to grasp the problem space. Execution of tasks in independent memory spaces requires new architectures, to efficiently handle device responsibilities and communication. In general, each process or device handles a portion of the problem space and contributes iteratively without full visibility of the problem it is cooperating to solve. Such programs must carefully manage memory transfers between processes to maximise device utilisation - the percentage of time spent executing operations - and throughput - the amount of data processed over a given period.

0.9.1 Concurrent Execution Paradigms

The two most important paradigms for execution of operations on parallel hardware are Single Instruction, Multiple Data Streams (SIMD) and Multiple Instruction, Multiple Data Streams (MIMD). SIMD executes the same instruction across several data points at once, making it appropriate for batched training of machine learning models. SIMD is useful for vector operations, and is implemented on GPU, and specialised ML accelerators like TPUs. MIMD, on the other hand, executes different instructions concurrently on different data. We have already discussed the MIMD paradigm in the context of multi-threading on multi-core processors. MIMD is common on all general purpose processors today. Multiple Instruction, Single Data Stream (MISD) is less common, used in applications requiring reliable fault tolerance through redundant error-checking. Single Instruction, Single Data Stream (SISD) is the paradigm implemented on single-core processors, where all operations must execute in sequence on a single stream of data (7).

0.9.2 Design Patterns for Multi-Processing and Distributed Programs

Tasks executed in independent memory spaces require specialised software design patterns, to handle device responsibilities and communication. These architectures must specify patterns for task scheduling and data sharing between processes, to allow many independent processes to cooperate on large tasks. Architectures relevant to ML implementations include the host-device pattern, the parameter server pattern, and the decentralised communication pattern. In any case, each device in a distributed computing setup is known as a node, having its own compute which can execute asynchronously of other nodes (contingent on task and data availability). The architectures are distinguished in how they manage task allocation and distribute changed model states between devices.

The host-device model assigns a central host, typically a CPU, to manage branching procedures, and sends computationally intensive vectorised operations to GPUs for execution. The host schedules tasks, launches kernels on subordinate devices, manages memory allocation on devices, and collects/redistributes data to devices. This model is effective in small-scale applications, but scales poorly. Centralised, SISD coordination of devices bottlenecks utilisation as device count grows. In large clusters, the host may still be allocating tasks and redistributing data to workers for a new round of training when the first device finishes executing (3).

The parameter server model creates a server to hold and manage the current model state, and workers to compute updates to the parameter set. Workers compute gradient updates and pass them to the parameter server. The server aggregates updates from its workers, updates the model state, and redistributes the parameters to worker devices. This architecture suffers the same bottlenecking problem as the simple host-device model, though it operates through a network interface abstraction and is therefore more amenable to multi-processing (MIMD) acceleration on the server node (19).

The decentralised communication model does away with the central coordinator entirely. Each device communicates only with its neighbors, sending and receiving partial results in a pipelined manner until all devices have the complete set of data. This reduces network contention and scales well with the number of devices, making it suitable for large-scale training where the previously discussed models carry unacceptable communication overhead (37).

0.10 Industrial Parallel Computing Using OpenCL and Horovod

0.10.1 OpenCL for Host-Device Architectures

Open Computing Language (OpenCL) is a hardware-agnostic standard for parallel computing. It enables hardware acceleration on CPUs, GPUs, TPUs, and other specialised accelerators. It is an open standard, and is therefore portable across hardware vendors (Group).

OpenCL provides a granular API for interacting with heterogenous devices, including memory buffers, synchronisation operators (semaphores), kernels, command buffers, and virtual memory shared between device and host (17, p.125). OpenCL operates implements a host-device pattern (17, p.2), as defined in the section above. It implements execution on kernels by defining work items, which are execute in groups on device (17, p.19). The memory context of an OpenCL program is split between device and host (17, p.30), and the engineer defines memory management for the instance on the host program. This includes allocating memory on the host and devices; moving data between them; and synchronising memory between devices (17, p.20).

The standard's direct control over low-level memory gives it a steep learning curve, but allows fine-grained optimisation across diverse hardware. For this reason, I have used OpenCL for kernel management in Catalyst. Low-level control forces a familiarity with low-level concepts like execution paradigms, memory management across the memory hierarchy, and optimisation techniques.

0.10.2 Horovod for Decentralised Communication Architectures

Horovod is an open-source framework for distributed deep learning which implements the Message Passing Interface (MPI) standard for distributed devices (30). MPI defines procedures for processes to cooperatively exchange information in parallel computing environments. Inter-process communication requires writing to, and reading from, shared memory. MPI specifies operations for node-to-node communication, such as send and receive (22, p.31).

It also specifies operations for communication across groups of nodes, such as broadcasting to groups of nodes and gathering information from groups of nodes (22, p.189). MPI organises processes in groups, which can then be organised into encapsulated communication contexts (22, p.308), to define communication patterns between devices.

Horovod builds on MPI by providing specialised methods for deep learning applications, notably ring-AllReduce (12). Ring-AllReduce arranges devices with shared context into a ring topology, and passes gradient updates around the ring. Each device thereby sends and receives partial updates to the weight space at each iteration, progressively building the complete gradient across all devices.

This method eliminates bottlenecking from central host devices or parameter servers, but requires efficient coordination between devices in the ring. Communication overhead increases sub-linearly with the number of devices, since each device communicates only with its neighbours, making this operation highly scalable. This is an improvement over simpler AllReduce implementations, where communication overhead scales linearly with node count (8)

To decrease bottlenecking from communication, Horovod runs message passing asynchronously. Communication executes while gradient updates are being computed. Devices spend less time waiting for data, improving throughput. Horovod's ”tensor fusion” optimisation improves on this again by pooling several gradient updates into a single larger message (13).

0.10.3 Bonzo-HPC Distributed Computing Implementation

To support training large-language models on Catalyst with home hardware, I will engineer Bonzo-HPC, a basic distributed computing framework implementing a simplified message passing interface (MPI). At present, Catalyst operates on a host-device model. Bonzo will use a simple MPI to explore advanced concepts in distributed ML. Bonzo will take over from OpenCL, handling device querying, memory management, task scheduling, and inter-device communication. The design is in draft, and will continue to improve as I complete Catalyst.

CONCLUSION

The development of scalable and efficient machine learning frameworks is critical for addressing the increasing computational demands of modern deep learning workflows. This report has examined the mathematical structure of neural networks, the principles of their optimisation, and their implementation as computational graphs. It has discussed the theory of concurrent programming. It has discussed concepts and architectures from the level of multi-threading to large-scale distributed computing frameworks. It has analysed industrial machine learning frameworks, particularly PyTorch and TensorFlow, and the distributed computing solution Horovod.

The work-in-progress Catalyst framework combines modularity, extensibility, and hardware-agnostic design, building a robust base for future enhancements, including support for transformers, computational graph optimisation, and distributed training. Catalyst will be complemented by Bonzo-HPC, a high-performance computing library which will enable distributed execution of Catalyst workloads. Together, these projects have developed, in the author, a foundational understanding of the design and implementations of machine learning frameworks.

As the author continues to iteratively expand these tools, this project aims to achieve a deeper understanding of machine learning systems, bridging the gap between theory and practice of high-performance machine learning.

ACKNOWLEDGMENTS

Thank you to my Advanced Algorithms Professor, Dr Aleksandar Ignjatovic, for encouraging me to tackle this project when I was beginning to think that I was not cut out for computer science.

Thank you to my mum for supporting me on this endeavour, despite her natural disinclination towards digital technology.

REFERENCES

[1] AI Stack Exchange User (n.d.). Plotting loss vs number of updates made and plotting loss vs run time. https://ai.stackexchange.com/questions/22691/plotting-loss-vs-number-of-updates-made-and-plotting-loss-vs-ru

[2] Boehm, M., Reinwald, B., Hutchison, D., Evfimievski, A. V., and Sen, P. (2018). On optimizing operator fusion plans for large-scale machine learning in systemml.

[3] Corporation, I. (2023). Gpu execution model overview.

[4] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. (2022). Flashattention: Fast and memory-efficient exact attention with io-awareness.

[5] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale.

[6] Ezyang, E. (2019). Pytorch internals. http://blog.ezyang.com/2019/05/pytorch-internals/.

[7] Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909.

[8] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2020). Applications of ring-allreduce in distributed deep learning. *arXiv preprint arXiv:2003.03009*, page 13.

[Group] Group, K. Opencl overview.

[10] Hastie, T., Tibshirani, R., and Friedman, J. (2013). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, 2nd edition. Chapter title: Neural Networks.

[11] Hooker, S. (2020). The hardware lottery.

[12] Horovod Team (2024). Horovod concepts. https://horovod.readthedocs.io/en/stable/concepts_include.html.

[13] Horovod Team (n.d.). Horovod: Tensor fusion optimization. https://horovod.readthedocs.io/en/stable/tensor-fusion_include.html.

[14] Hughes, D. and Hughes, P. (2004). *Distributed and Parallel Programming*. Springer.

[15] Jadon, A., Patil, A., and Jadon, S. (2022). A comprehensive survey of regression based loss functions for time series forecasting.

[16] Jumper, J., Evans, R., Pritzel, A., et al. (2021). Highly accurate protein structure prediction with alphafold. *Nature*, 596:583–589.

[17] Khronos Group (2024). The opencl api specification: Chapter 5 - the opencl runtime. https://bashbaug.github.io/OpenCL-Docs/pdf/OpenCL_API.pdf.

[18] Larsen, R. M., Shpeisman, T., and Contributors, G. O. S. (2021). Tensorflow graph optimizations. https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf. Presented in the Principles of Data-Intensive Systems course, Winter 2021, Stanford University, Slides 9, 10, 21, 22, 28, 33.

[19] Li, M., Andersen, D. G., Smola, A. J., et al. (2014). Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598. USENIX Association.

[20] Liu, H., Zaharia, M., and Abbeel, P. (2023). Ring attention with blockwise transformers for near-infinite context.

[LLVM Project] LLVM Project. Llvm features. https://llvm.org/Features.html.

[22] MPI Forum (2023). Mpi: A message-passing interface standard, version 4.1. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[23] oneDNN Team (n.d.). Understanding memory formats. https://oneapi-src.github.io/oneDNN/dev_guide_understanding_memory_formats.html. oneAPI Deep Neural Network Library Team.

[24] OpenXLA Team (2024a). Xla: Accelerated linear algebra by openxla. https://openxla.org/xla.

[25] OpenXLA Team (2024b). Xla architecture by openxla. https://openxla.org/xla/architecture.

[26] Peng, J., Cui, X., and Hu, Y. (2018). Optimizing distributed training by extending horovod with new features. In *Workshop on Systems and Machine Learning, NeurIPS*.

[27] PyTorch Team (2021). Overview of pytorch autograd engine. https://pytorch.org/blog/overview-of-pytorch-autograd-engine/.

[28] PyTorch Team (2024). Pytorch/xla release r2.5. https://pytorch.org/xla/release/r2.5/index.html.

[29] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection.

[30] Sergeev, A. and Balso, M. D. (2018). Horovod: fast and easy distributed deep learning in tensorflow.

[31] Stephen Elliott (2024). Catalyst dev class diagram. https://www.figma.com/design/UXEFP7oxLK4gug4gUgBLbK/Catalyst-Project?

node-id=0-1&t=wnh4z1ZfZPlW6jwb-1.

[Stephen Elliott / one-2] Stephen Elliott / one-2. Catalyst: A neural network framework in c++. https://github.com/one-2/catalyst. Open-source project under the MIT License.

[33] TensorFlow Team (2017). Tensorflow: Control flow implementation. http://download.tensorflow.org/paper/white_paper_tf_control_flow_implementation_2017_11_1.pdf. Page 7.

[34] TensorFlow Team (2024a). Tensorflow guide: Graph optimization. https://www.tensorflow.org/guide/graph_optimization.

[35] TensorFlow Team (2024b). Tensorflow guide: Mixed precision. https://www.tensorflow.org/guide/mixed_precision.

[36] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

[37] Yu, M., Ji, B., Rajan, H., and Liu, J. (2022). On scheduling ring-all-reduce learning jobs in multi-tenant gpu clusters with communication contention. *arXiv preprint arXiv:2207.07817*.