AutoCalc

A From-Scratch Autograd Engine in C++ with Python Bindings

Documentation & System Design Guide

February 19, 2026

Contents

Part I

Introduction

Chapter 1

What Is AutoCalc?

AutoCalc is a complete, from-scratch machine-learning framework written in C++17. It contains:

- An **automatic differentiation** (autograd) engine that can compute gradients of arbitrary compositions of mathematical operations.
- A **neural-network module system** (layers, loss functions, optimizers) similar in spirit to PyTorch's `torch.nn`.
- A hand-written **SGEMM kernel** (single-precision general matrix multiply) with cache-aware tiling and multithreading.
- A **thread pool and parallel-for** runtime for data-parallel computation.
- **Python bindings** via pybind11 so the entire engine can be used from Python scripts.
- **Data loading** utilities (datasets, data loaders, transforms) that mirror PyTorch's `torch.utils.data`.

The project is designed to be *educational*: every component is written from scratch so you can see exactly how a modern ML framework works under the hood.

1.1 Who This Document Is For

This guide is written for someone who:

- May not have written C++ before (we explain every C++ idiom we encounter).
- May not be familiar with machine learning math (we derive the key formulas).
- Wants to understand *system design*: why the code is structured the way it is, what trade-offs were made, and what bugs were found and fixed.

1.2 Conventions

- `monospace` denotes code: file names, function names, types.
- "Shape" means the dimensions of a tensor, e.g. `[B, C, H, W]` for a batch of images with B samples, C channels, height H, width W.

- We use 0-based indexing everywhere (as C++ does).
- "Leaf" means a node with no parents in the computation graph (typically a learnable parameter or input data).

Chapter 2

Repository Layout

Path	Purpose
`include/ag/core/`	Core autograd: `Node`, `Variable`
`include/ag/ops/`	Operator declarations (elementwise, linalg, etc.)
`include/ag/nn/`	Neural-network module system (layers, loss, optimizer)
`include/ag/parallel/`	Thread pool, parallel_for, configuration
`include/ag/data/`	Dataset, DataLoader, transforms
`include/ag/sys/`	System queries (cache sizes)
`src/ag/`	Corresponding `.cpp` implementations
`bindings/`	pybind11 C++→Python bridge
`ag/`	Python package shim (`__init__.py`)
`c_tests/`	C++ unit tests
`pytests/`	Python test suite
`c_demos/`	C++ demo programs (MNIST, ResNet, LSTM)
`py_demos/`	Python demo scripts
`CMakeLists.txt`	Build system

Key Idea

In C++ projects, **headers** (`.hpp`) declare interfaces (types, function signatures), while **source files** (`.cpp`) contain implementations. Headers live in `include/` so other translation units can `#include` them; sources live in `src/`.

Part II

The Autograd Core

Chapter 3

Automatic Differentiation: The Big Picture

3.1 What Problem Does Autograd Solve?

Training a neural network requires computing *gradients*: partial derivatives of a scalar loss L with respect to every learnable parameter θ_i. The chain rule of calculus lets us decompose this:

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial z_n} \cdot \frac{\partial z_n}{\partial z_{n-1}} \cdots \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial \theta_i}$$

where $z_1, z_2, \ldots, z_n$ are intermediate results. Computing this by hand for every network architecture would be tedious and error-prone.

Reverse-mode automatic differentiation ("backpropagation") automates this. The idea:

1. **Forward pass**: evaluate the function, recording every operation in a directed acyclic graph (DAG).
2. **Backward pass**: walk the DAG in reverse topological order, applying the chain rule at each node to accumulate gradients.

3.2 Why Reverse Mode?

There are two modes of AD:

- **Forward mode**: propagates derivatives *forward* through the graph. Cost scales with the number of *inputs* (parameters).
- **Reverse mode**: propagates derivatives *backward* from the output. Cost scales with the number of *outputs*.

In ML, we have one scalar output (the loss) and millions of parameters, so reverse mode is dramatically cheaper.

Chapter 4

Node and Variable

These two types are the heart of AutoCalc. Every tensor in the system is represented by a `Variable`, which is a thin wrapper around a heap-allocated `Node`.

4.1 The Node Struct

Defined in `include/ag/core/variables.hpp`:

Listing 4.1: Node struct (simplified)

```
struct Node {
  std::vector<float> value;    // the tensor data, flattened
  std::vector<float> grad;     // gradient, same size
  std::vector<std::size_t> shape; // e.g. {2, 3} for a 2x3 matrix

  bool requires_grad = false;

  std::vector<std::shared_ptr<Node>> parents; // inputs to this op
  std::function<void()> backward;             // the local VJP
};
```

Let us unpack each field:

`value`
: A flat `std::vector<float>` storing the tensor elements in **row-major** order. A shape {2,3} matrix $\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$ is stored as `[a, b, c, d, e, f]`.

`grad`
: Same layout as `value`, but holds $\partial L / \partial \text{this\_tensor}$. Initialized to zeros; accumulated during backward.

`shape`
: A vector of dimension sizes. `numel(shape)` = product of all entries = length of `value`.

`requires_grad`
: If `false`, gradient will not be computed for this node. Input data tensors are typically `requires_grad=false`; learnable parameters are `true`.

`parents`
: Pointers to the input nodes of whatever operation created this node. For a leaf (parameter or data), this is empty.

`backward`
: A `std::function<void()>` closure that implements the *vector-Jacobian product* (VJP) for the operation that created this node. When called, it reads `this->grad` (the upstream gradient) and accumulates into each parent's `grad`.

Key Idea

A `std::shared_ptr<T>` is a C++ smart pointer that automatically frees the object when the last pointer to it is destroyed. Multiple `shared_ptr`s can point to the same object; an internal reference count tracks how many exist. This is how AutoCalc manages the lifetime of Nodes: when no Variable or parent list references a Node, it is freed.

4.2 The Variable Class

Listing 4.2: Variable class (simplified)

```
class Variable {
public:
  std::shared_ptr<Node> n;  // the wrapped node

  Variable();  // creates an empty node
  Variable(const std::vector<float>& value,
           const std::vector<std::size_t>& shape,
           bool requires_grad = true);

  void backward();              // scalar loss -> seed=1
  void backward(const std::vector<float>& seed);
  void zero_grad();             // zero grads in entire subgraph
};
```

`Variable` is a *value type* that holds a `shared_ptr` to the actual data. Copying a `Variable` is cheap: it just increments the reference count. This means that when an operator returns a new `Variable`, the caller gets a lightweight handle, not a copy of the tensor data.

4.3 Grad Mode and NoGradGuard

A global thread-local boolean controls whether new nodes get `requires_grad=true`:

```
inline thread_local bool __grad_enabled = true;
inline bool is_grad_enabled() { return __grad_enabled; }
inline void set_grad_enabled(bool v) { __grad_enabled = v; }
```

`NoGradGuard` is an RAII object that temporarily disables grad:

```
struct NoGradGuard {
  bool prev_;
  NoGradGuard()  { prev_ = is_grad_enabled(); set_grad_enabled(false);
     }
```

```
4     ~NoGradGuard() { set_grad_enabled(prev_); }
5 };
```

Key Idea

RAII (Resource Acquisition Is Initialization) is a C++ pattern where a constructor acquires a resource and the destructor releases it. Because C++ guarantees destructors run when an object goes out of scope (even via exceptions), RAII ensures resources are always released. Here, the "resource" is the grad-disabled state.

Chapter 5

The Backward Pass

5.1 Topological Sort

Before running backward, we need the nodes in an order where every node's parents are processed *before* the node itself (so gradients flow correctly from output to inputs). This is a **topological sort** of the DAG.

AutoCalc uses a recursive DFS:

Listing 5.1: Topological collection

```
void topo_collect(const shared_ptr<Node>& node,
                  vector<shared_ptr<Node>>& order,
                  unordered_set<Node*>& seen) {
  if (!node || seen.count(node.get())) return;
  seen.insert(node.get());
  for (auto& p : node->parents)
    topo_collect(p, order, seen);
  order.push_back(node);  // parents first, then self
}
```

After collection, `order` has parents before children. We iterate in *reverse* to get the backward order (children before parents):

Listing 5.2: Backward loop

```
void Variable::backward(const vector<float>& seed) {
  vector<shared_ptr<Node>> order;
  unordered_set<Node*> seen;
  topo_collect(n, order, seen);

  // Seed the output gradient
  for (size_t i = 0; i < seed.size(); ++i)
    n->grad[i] += seed[i];

  // Reverse iterate: output -> inputs
  for (auto it = order.rbegin(); it != order.rend(); ++it) {
    if ((*it)->backward) (*it)->backward();
  }
```

```
}
```

5.2 Post-Backward Cleanup

After the backward pass completes, intermediate nodes (non-leaf) have their `backward` closure and `parents` list cleared:

Listing 5.3: Graph cleanup after backward

```
for (auto& node : order) {
  if (node->parents.empty()) continue; // leaf -- keep alive
  node->backward = nullptr;
  node->parents.clear();
}
```

This is **critical** for memory management. Without it, the backward closures hold `shared_ptr`s to parent nodes, forming reference cycles that prevent garbage collection. Over many training iterations, this causes unbounded memory growth (the OOM bug that was fixed—see Part IX).

5.3 zero_grad

Before each training step, gradients from the previous step must be reset to zero. `zero_grad()` does a topological walk from the loss node and sets every reachable node's `grad` vector to all zeros.

Part III

The Operator Library

Chapter 6

Tensor Utilities

`include/ag/ops/tensor_utils.hpp` provides fundamental helpers:

`numel(shape)` Returns the product of all dimensions (total number of elements).

`strides_for(shape)` Computes row-major strides. For shape $\{d_0, d_1, \ldots, d_{n-1}\}$, stride $s_i = \prod_{j=i+1}^{n-1} d_j$.

`ravel_index(idx, strides)` Converts a multi-dimensional index to a flat offset: $\sum_i \text{idx}[i] \times \text{strides}[i]$.

`unravel_index(linear, shape)` Inverse of ravel: converts a flat offset back to multi-dimensional indices.

`broadcast_two(A, B)` NumPy-style broadcasting: right-aligns shapes, pads with 1s, and checks compatibility (dimensions must match or one must be 1).

Chapter 7

Elementwise Operations

`src/ag/ops/ops_elmwise.cpp` implements addition, subtraction, multiplication, division, negation, sin, cos, exp, and power.

7.1 Pattern: How an Op Is Built

Every operator follows the same pattern. Let us trace `add` as the canonical example:

1. **Compute output shape** via broadcasting.
2. **Allocate output Node**: set shape, value, grad, parents.
3. **Forward compute**: loop over output elements, map each to the corresponding input elements (handling broadcasting), compute the result.
4. **Attach backward closure**: a lambda that, when called, reads `out->grad` and accumulates into `A.n->grad` and `B.n->grad` according to the local Jacobian.
5. **Return** a `Variable` wrapping the output node.

7.1.1 Forward

For `add(A, B)`, the forward is simply:

$$\text{out}[i] = A[\text{map}(i)] + B[\text{map}(i)]$$

where `map` handles broadcasting (mapping an output index to the corresponding input index, collapsing broadcast dimensions to index 0).

7.1.2 Backward

For addition, $\partial(\text{out})/\partial A = 1$ and $\partial(\text{out})/\partial B = 1$, so:

$$\frac{\partial L}{\partial A[j]} \mathrel{+}= \sum_{i:\,\text{map}(i)=j} \frac{\partial L}{\partial \text{out}[i]}$$

The sum handles broadcast: if A had a dimension of size 1 that was broadcast to size n, the gradient contributions from all n output positions are summed back into that single input element.

7.1.3 Parallelism

When the output has more than 4096 elements (`ELEM_SERIAL_CUTOFF`), the forward and backward loops are parallelized using `parallel_for` with a grain size of 1024 elements per task.

7.2 Multiplication Backward

For out $= A \cdot B$:

$$\frac{\partial L}{\partial A[j]} \mathrel{+}= \sum_{i:\,\mathrm{map}(i)=j} \frac{\partial L}{\partial \mathrm{out}[i]} \cdot B[\mathrm{map}_B(i)]$$

and symmetrically for B. The forward values of the *other* input are needed during backward—this is why intermediate values must be kept alive until backward completes.

7.3 Other Elementwise Ops

Op	**Forward**	$\partial/\partial x$
`neg(x)`	$-x$	-1
`sinv(x)`	$\sin(x)$	$\cos(x)$
`cosv(x)`	$\cos(x)$	$-\sin(x)$
`expv(x)`	e^x	e^x
`pow(x,p)`	x^p	$p \cdot x^{p-1}$ (w.r.t. x)
`div(a,b)`	a/b	$1/b$ (w.r.t. a), $-a/b^2$ (w.r.t. b)

Chapter 8

Activations

`src/ag/ops/activations.cpp` provides:

`relu(x)` $\max(0, x)$. Backward: gradient is passed through where $x > 0$, zeroed where $x \leq 0$.

`logsumexp(x, axes, keepdims)` Numerically stable: $\text{LSE}(x) = m + \log \sum_i \exp(x_i - m)$ where $m = \max(x)$. Used in cross-entropy loss.

Chapter 9

Reduction Operations

`src/ag/ops/reduce.cpp` implements `sum` and `mean` along specified axes (with optional keepdims).

Sum backward: gradient is broadcast back to the input shape. If we summed axis 1 of a $[3, 4]$ tensor to get $[3]$, each gradient element is copied to all 4 positions along axis 1.

Mean backward: same as sum, but divided by the number of elements that were averaged.

Chapter 10

Linear Algebra: Matmul and Transpose

10.1 Matrix Multiplication

`matmul(A, B)` computes $C = A \times B$ where A is $[\ldots, M, K]$ and B is $[\ldots, K, N]$, producing C of shape $[\ldots, M, N]$. Batch dimensions are broadcast.

10.1.1 Forward

The core computation calls `sgemm_f32` (our custom GEMM kernel, described in Part V). For batched inputs, the batch dimensions are iterated and a separate GEMM is dispatched per batch element.

10.1.2 Backward

The gradients for matrix multiplication follow from the chain rule:

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial C} \cdot B^T \qquad \text{(shape: } [M, N] \times [N, K] = [M, K]\text{)} \tag{10.1}$$

$$\frac{\partial L}{\partial B} = A^T \cdot \frac{\partial L}{\partial C} \qquad \text{(shape: } [K, M] \times [M, N] = [K, N]\text{)} \tag{10.2}$$

10.1.3 The weak_ptr Fix

The backward closure originally captured `C.n` (a `shared_ptr`) by value. Since `C.n->backward` *is* that closure, this created a reference cycle: the Node's backward lambda owned a `shared_ptr` to the Node itself. The fix: capture a `std::weak_ptr` instead, and `lock()` it at the start of the backward call. See Part IX for the full story.

10.2 Transpose

`transpose(A)` swaps the last two dimensions of a tensor, producing a new tensor with copied data. For a $[\ldots, M, N]$ input, the output is $[\ldots, N, M]$.

Backward: transposing the gradient is its own inverse, so we transpose the incoming gradient back.

10.3 Slicing: `at(A, begin, end)`

Extracts a contiguous sub-tensor. `begin` and `end` are per-dimension half-open ranges.

Backward: the gradient is scattered back into the corresponding positions of the input's gradient tensor (with zeros elsewhere).

Part IV

The Neural Network Module System

Chapter 11

Module Base Class

`include/ag/nn/module.hpp` defines the abstract base:

Listing 11.1: Module interface (key members)

```
class Module {
public:
  virtual Variable forward(const Variable& x) = 0;

  vector<Variable*> parameters();      // recursive collection
  void zero_grad();
  void train();
  void eval();

  Module& register_module(Module& child);
  Module& register_parameter(const string& name, Variable& v);

protected:
  virtual vector<Variable*> _parameters() = 0; // own params only
};
```

11.0.1 Parameter Collection

`parameters()` is recursive: it calls `_parameters()` on `this` module to get its own parameters, then recurses into every registered child module. This is how the optimizer discovers all learnable weights in a model.

11.0.2 Train vs. Eval Mode

`train()` and `eval()` set a boolean flag that affects layers like BatchNorm (which uses running stats in eval mode) and Dropout (which is disabled in eval mode).

Chapter 12

Sequential

Listing 12.1: Sequential container

```
struct Sequential : Module {
  vector<shared_ptr<Module>> layers;

  void push(shared_ptr<Module> m) {
    register_module(m);
    layers.push_back(move(m));
  }

  Variable forward(const Variable& x) override {
    Variable y = x;
    for (auto& m : layers) y = m->forward(y);
    return y;
  }
};
```

`Sequential` chains layers: the output of one is the input of the next. It has no parameters of its own; all parameters come from the contained layers.

Chapter 13

Layers

13.1 Linear (Fully Connected)

`Linear(in, out)` holds:

- Weight W: shape $[\text{in}, \text{out}]$
- Bias b: shape [out] (optional)

Forward: $y = xW + b$ where x is $[B, \text{in}]$. The bias is broadcast across the batch dimension.

13.2 Conv2d (2D Convolution)

`Conv2d(Cin, Cout, kernel, stride, padding, dilation)` implements convolution via the **im2col** approach.

13.2.1 What Is Convolution?

A 2D convolution slides a small *kernel* (filter) over a 2D input image and computes a weighted sum at each position. If the input has C_{in} channels and the kernel is $K_H \times K_W$, then at each output position (oh, ow) the operation reads a *patch* of size $C_{\text{in}} \times K_H \times K_W$ from the input and dot-products it with the kernel weights. With C_{out} different kernels, we get C_{out} output channels.

The output spatial dimensions are:

$$H_{\text{out}} = 1 + \frac{H + 2P_H - D_H(K_H - 1) - 1}{S_H}, \qquad W_{\text{out}} = 1 + \frac{W + 2P_W - D_W(K_W - 1) - 1}{S_W}$$

where P is padding, S is stride, and D is dilation (spacing between kernel elements).

13.2.2 The im2col Trick: Why and How

A naive convolution loops over every output position and every kernel element inside a 6-deep nested loop. This is slow because:

- The memory access pattern is irregular (strided reads from the input).
- The compiler cannot vectorize the inner loops well.

- We cannot reuse highly optimized GEMM kernels.

im2col (image-to-column) transforms the problem so that the entire convolution becomes a single matrix multiplication:

1. **Reshape the weight tensor** from $[C_{\text{out}}, C_{\text{in}}, K_H, K_W]$ into a 2D matrix W_{col} of shape $[K, C_{\text{out}}]$, where $K = C_{\text{in}} \times K_H \times K_W$. Each column of W_{col} contains one output filter flattened into a vector.
2. **Build the im2col matrix.** For each output position (b, oh, ow) in the batch, we extract the corresponding input patch of size K and lay it out as a *row* of a large matrix X_{col} of shape $[\text{rows}, K]$, where $\text{rows} = B \times H_{\text{out}} \times W_{\text{out}}$.

   Concretely, for output position (oh, ow), the patch covers input positions:

   $$\text{ih} = oh \cdot S_H - P_H + kh \cdot D_H, \qquad \text{iw} = ow \cdot S_W - P_W + kw \cdot D_W$$

   for $kh \in [0, K_H)$, $kw \in [0, K_W)$, across all C_{in} channels. If (ih, iw) falls outside the input bounds, we use zero (this is how zero-padding works).
3. **Multiply**: $Y_{\text{col}} = X_{\text{col}} \times W_{\text{col}}$, producing shape $[\text{rows}, C_{\text{out}}]$.
4. **Reshape and add bias**: scatter the rows of Y_{col} back into the $[B, C_{\text{out}}, H_{\text{out}}, W_{\text{out}}]$ output tensor, then add the bias (broadcast over spatial dimensions).

Key Idea

im2col is the standard trick used by most deep learning frameworks (including cuDNN) to implement convolution efficiently. By rearranging input patches into matrix columns, we convert convolution into GEMM, which has decades of optimization behind it. The trade-off is memory: the im2col matrix has redundant copies of overlapping input elements.

13.2.3 Concrete Example

Consider a $1 \times 1 \times 4 \times 4$ input (1 batch, 1 channel, 4×4) with a 3×3 kernel, stride 1, no padding:

- Output size: $H_{\text{out}} = 1 + (4 - 3)/1 = 2$, $W_{\text{out}} = 2$.
- $K = 1 \times 3 \times 3 = 9$.
- im2col matrix: 4 rows $\times$ 9 columns. Each row is one flattened 3×3 patch from the input.
- Weight matrix: $9 \times C_{\text{out}}$.
- One GEMM call: $[4, 9] \times [9, C_{\text{out}}] = [4, C_{\text{out}}]$.

13.2.4 Blocked im2col in AutoCalc

Building the full im2col matrix for a large input can consume significant memory. AutoCalc mitigates this by processing rows in **blocks** of size `ROW_BLOCK` (default 256, matching the GEMM `MC` tile):

Listing 13.1: Blocked im2col + GEMM (simplified)

```
for each block of ROW_BLOCK rows:
  1. Build im2col_block: [ROW_BLOCK, K]  (small buffer)
  2. GEMM:  im2col_block * W_col -> Y_block: [ROW_BLOCK, Cout]
```

```
    3. Scatter Y_block into output tensor
```

These blocks are processed in parallel via `parallel_for`, so different threads handle different spatial regions simultaneously.

13.2.5 Backward Pass

The backward for Conv2d computes two gradients:

- ∇W: for each output position, the gradient contribution is the outer product of the upstream gradient and the corresponding im2col row. This is again a GEMM: $X_{\text{col}}^T \times \nabla Y_{\text{col}}$.
- ∇X: the upstream gradient is multiplied by the transposed weight matrix ($\nabla Y_{\text{col}} \times W_{\text{col}}^T$), then scattered back to the input positions using the reverse of the im2col index mapping ("col2im").

13.3 BatchNorm2d

Batch normalization over NCHW tensors. Per channel c:

Training mode:

$$\mu_c = \frac{1}{N \cdot H \cdot W} \sum_{n,h,w} x_{n,c,h,w} \tag{13.1}$$

$$\sigma_c^2 = \frac{1}{N \cdot H \cdot W} \sum_{n,h,w} (x_{n,c,h,w} - \mu_c)^2 \tag{13.2}$$

$$\hat{x}_{n,c,h,w} = \frac{x_{n,c,h,w} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} \tag{13.3}$$

$$y_{n,c,h,w} = \gamma_c \hat{x}_{n,c,h,w} + \beta_c \tag{13.4}$$

Running statistics are updated with exponential moving average:

$$\bar{\mu} \leftarrow (1 - m)\bar{\mu} + m \cdot \mu_c$$

Eval mode: uses running mean and variance instead of batch statistics.

The mean and variance are computed using **Welford's online algorithm**, which is numerically more stable than the naive two-pass approach.

Learnable parameters: γ (scale) and β (shift), each of shape $[C]$.

13.4 MaxPool2d and AvgPool2d

`MaxPool2d(kernel, stride, padding)` slides a window over each channel and takes the maximum. Backward: gradient flows only to the position that achieved the max ("argmax routing").

`AvgPool2d` takes the mean instead. Backward: gradient is divided equally among all positions in the window.

13.5 Dropout

During training, each element is independently zeroed with probability p, and surviving elements are scaled by $1/(1-p)$ to preserve the expected value.

The random mask is generated using **SplitMix64**, a fast non-cryptographic PRNG. The seed incorporates a `call_counter` that increments each forward call, ensuring different masks each time.

During eval mode, Dropout is a no-op (identity function).

13.6 LSTM

Long Short-Term Memory. Implements the standard LSTM cell equations:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{(forget gate)} \tag{13.5}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{(input gate)} \tag{13.6}$$

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \quad \text{(candidate)} \tag{13.7}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad \text{(cell state)} \tag{13.8}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(output gate)} \tag{13.9}$$

$$h_t = o_t \odot \tanh(c_t) \quad \text{(hidden state)} \tag{13.10}$$

Chapter 14

Loss Functions

14.1 Cross-Entropy Loss

`cross_entropy(logits, targets)` computes:

$$L = \frac{1}{B} \sum_{b=1}^{B} [\text{LSE}(\mathbf{x}_b) - x_{b,t_b}]$$

where $\text{LSE}(\mathbf{x}) = \log \sum_c \exp(x_c)$ and t_b is the target class for sample b.

Backward: the gradient with respect to logit $x_{b,c}$ is:

$$\frac{\partial L}{\partial x_{b,c}} = \frac{1}{B} \left(\text{softmax}(x_b)_c - \mathbf{1}[c = t_b]\right)$$

This is the classic "softmax minus one-hot" gradient.

Chapter 15

SGD Optimizer

`SGD` implements stochastic gradient descent with optional momentum, Nesterov acceleration, and weight decay:

Listing 15.1: SGD step (pseudocode)

```
for each parameter p:
  g = p.grad
  if weight_decay > 0:
    g += weight_decay * p.value  // L2 regularization
  if momentum > 0:
    v = momentum * v_prev + g
    if nesterov:
      update = g + momentum * v
    else:
      update = v
  else:
    update = g
  p.value -= lr * update
```

Velocity vectors are stored per-parameter (keyed by `Node*` identity) in an `unordered_map`.

Part V

The SGEMM Kernel and Parallelism

Chapter 16

Why a Custom SGEMM?

Matrix multiplication is the computational bottleneck of neural networks. Rather than linking an external BLAS library, AutoCalc implements its own single-precision GEMM (`sgemm_f32`) to demonstrate how high-performance linear algebra works from first principles.

16.1 The Memory Hierarchy Problem

Modern CPUs can perform floating-point arithmetic orders of magnitude faster than they can fetch data from main memory. A naive triple-loop matmul ($O(MNK)$ multiply-adds) spends most of its time waiting for memory.

The solution is **tiling** (also called blocking): break the matrices into small blocks that fit in the CPU's fast caches, and reuse each loaded block as many times as possible.

16.2 Tiling Strategy

AutoCalc uses a three-level tiling scheme:

1. **NC** × **KC** panels of B are packed into a contiguous buffer that fits in L2 cache.
2. **MC** × **KC** panels of A are packed into a buffer that fits in L1 cache.
3. **MR** × **NR** micro-tiles (8×8 in AutoCalc) are computed by a tight inner kernel that maximizes register reuse.

The tile sizes are chosen at runtime based on detected cache sizes:

Listing 16.1: Runtime tile selection

```
GemmTiles pick_tiles_runtime(size_t sizeofT) {
  auto ci = cache_info();  // L1d and L2 sizes in bytes
  int KC = L1 / (2 * NR * sizeofT);    // B panel fits half of L1
  int NC = 0.6 * L2 / (KC * sizeofT);  // B macro-panel fits 60% L2
  int MC = 0.5 * L2 / (T * KC * sizeofT); // A per-thread stripe
  // ... clamp and round to MR/NR multiples ...
}
```

16.3 The 8×8 Micro-Kernel

The innermost computation is an 8×8 outer-product accumulation:

Listing 16.2: Micro-kernel (scalar, unrolled)

```
void microkernel_8x8_f32(const float* Ap, const float* Bp,
                         float* C, int ldc, int kc) {
  float acc[8][8] = {{0}};
  for (int p = 0; p < kc; ++p) {
    // Load 8 elements of A and 8 elements of B
    // Compute all 64 products (8*8 FMAs)
    // ... fully unrolled ...
  }
  // Write acc back to C
}
```

This is a **rank-1 update** kernel: at each step p, we compute the outer product of an 8-element column of A with an 8-element row of B and accumulate into the 8×8 register tile.

16.4 Packing

Before the micro-kernel runs, matrix panels are "packed" into contiguous buffers with a specific layout:

- `packA`: rearranges an (mc × kc) panel of A into MR-wide strips, padded with zeros.
- `packB`: rearranges a (kc × nc) panel of B into NR-wide strips, padded with zeros.

Packing ensures sequential memory access in the micro-kernel, which is essential for cache line utilization.

16.5 Transpose-Aware Overload

A second overload of `sgemm_f32` accepts `Trans::N` or `Trans::T` flags for each input. Instead of materializing transposed copies, it uses stride-aware packing functions (`packA_f32_strided`, `packB_f32_strided`) that read the source matrix with swapped row/column strides.

Chapter 17

Thread Pool and parallel_for

17.1 Why a Custom Thread Pool?

Many operations in a neural network (elementwise ops on large tensors, GEMM tile dispatch, im2col construction) are embarrassingly parallel. Creating a new `std::thread` per task would be far too expensive—thread creation on Linux/macOS takes $\sim$50 μs, while the work per task may be only a few microseconds. A **persistent thread pool** creates workers once and reuses them for the lifetime of the process.

17.2 Thread Pool Architecture

`include/ag/parallel/pool.hpp` implements the pool as a class `ThreadPool` with the following components:

Worker threads (`workers_`)
: A `std::vector<std::thread>` created lazily on first use. The default count equals `std::thread::hardware_c` (i.e., the number of CPU cores), but can be overridden via the `AG_NUM_THREADS` environment variable or `set_max_threads()`. Workers run a `worker_loop()` that sleeps until work arrives.

Task queue (`tasks_`)
: A `std::deque<RangeTask>` where each task is a struct holding a `std::function<void(size_t, size_t)>` plus a `begin` and `end` range. Protected by a `std::mutex`.

Condition variable (`cv_`)
: Workers block on `cv_.wait()` when the queue is empty. `submit()` calls `cv_.notify_one()` to wake exactly one sleeping worker.

Inflight counter (`inflight_`)
: An `std::atomic<size_t>` incremented on submit, decremented when a worker finishes a task. When it reaches zero, the `done_cv_` condition variable is signaled.

Wait mechanism (`done_cv_`)
: The caller of `wait_for_all()` blocks on `done_cv_` until `inflight_` is zero. This is how `parallel_for` waits for all chunks to finish.

Exception propagation (`first_exc_`)

If any worker catches an exception, it is stored in `first_exc_` and the remaining queued tasks are drained. `wait()` then rethrows the exception on the calling thread.

Listing 17.1: Worker loop (simplified)

```
void worker_loop() noexcept {
  for (;;) {
    RangeTask task;
    {
      unique_lock lk(mu_);
      cv_.wait(lk, [&]{ return stop_ || !tasks_.empty(); });
      if (stop_ && tasks_.empty()) return;
      task = move(tasks_.front());
      tasks_.pop_front();
    }
    try {
      nesting_flag() = true;  // prevent nested parallel_for
      task.fn(task.begin, task.end);
    } catch (...) {
      capture_exception(current_exception());
    }
    nesting_flag() = false;
    complete_one();  // decrement inflight_
  }
}
```

Key Idea

The pool is a **Meyer's singleton**: `pool()` returns a reference to a function-local static `ThreadPool` object, which is constructed on first call and destroyed at program exit. This avoids the "static initialization order fiasco" (a classic C++ pitfall where global objects constructed in different translation units have undefined initialization order).

17.3 Thread-Local Storage (TLS)

Several components use `thread_local` variables for per-thread state without synchronization overhead:

- `tls_worker_id`: each pool thread gets a unique integer ID (0, 1, ...), used to index per-thread scratch buffers in the GEMM kernel.
- `nesting_flag()`: a boolean that is `true` when executing inside a pool worker. `parallel_for` checks this to avoid deadlock from nested parallelism.
- `thread_local std::vector<float> Ap_buf`: per-thread packing buffers in the GEMM kernel, reused across calls to avoid allocation overhead.

Key Idea

`thread_local` is a C++11 storage class that gives each thread its own independent copy of a variable. Unlike a global variable (shared by all threads, requiring locks), or a local variable (created/destroyed each function call), a `thread_local` variable is created once per thread and persists for that thread's lifetime. It combines the performance of a global with the safety of thread isolation.

17.4 parallel_for

`parallel_for(n, grain, body)` is the high-level API. It divides the half-open range $[0, n)$ into chunks and dispatches them to the pool.

17.4.1 Algorithm

1. **Compute thread cap** T: $\min(\texttt{get\_max\_threads}(), n)$.
2. **Check serial gates**: if $T = 1$, or $n = 0$, or `serial_override()` returns true (see below), run `body(0, n)` directly and return.
3. **Determine chunk count**:
   - If grain $= 0$ (auto): use T chunks.
   - Otherwise: chunks $= \min(T, \lceil n/\text{grain} \rceil)$.
4. **Divide work evenly**: $q = n/\text{chunks}$, $r = n \bmod \text{chunks}$. The first r chunks get $q+1$ elements, the rest get q. This ensures perfectly balanced loads (at most 1 element difference).
5. **Submit all chunks** via `submit_range()`.
6. **Wait**: call `wait_for_all()`, which blocks until `inflight_` reaches 0.
7. **Rethrow**: if any chunk threw an exception, rethrow it.

17.4.2 Grain Size

The **grain** parameter controls the minimum amount of work per task. Setting it too small creates excessive task overhead (mutex contention, function call overhead); too large wastes cores. Typical values in AutoCalc:

Use site	Grain
Elementwise ops (`add`, `mul`, ...)	1024 elements
GEMM tile dispatch	0 (auto: one chunk per thread)
Conv2d im2col blocks	1 block per task
Cross-entropy backward	B (serialize: avoids data race)

17.4.3 The Nesting Problem

Consider: `parallel_for` dispatches work to 8 threads. Inside each chunk, the code calls `matmul`, which itself calls `parallel_for`. If the inner call also submits to the same pool, we deadlock: the

outer chunks are waiting for pool workers, but those workers are blocked waiting for inner tasks that will never execute (all workers are busy with outer tasks).

AutoCalc prevents this with a **nesting guard**: each pool worker sets `nesting_flag() = true` before executing a task. `serial_override()` checks this flag and returns `true`, causing the inner `parallel_for` to run serially on the current worker thread.

17.5 Configuration

`include/ag/parallel/config.hpp` provides the unified control layer:

`get_max_threads() / set_max_threads(n)`
: A global `std::atomic<size_t>` thread cap. Defaults to `hardware_concurrency()`, overridable by the `AG_THREADS` environment variable. Setting to 1 forces all parallelism off.

`ScopedSerial`
: An RAII guard with a TLS depth counter. While active, all `parallel_for` calls run serially. Used in tests for reproducibility.

`deterministic_enabled() / AG_DETERMINISTIC`
: When true, floating-point operations must be order-deterministic. Since parallel reduction can sum in different orders (yielding different rounding), this mode forces serial execution by default.

`ScopedDeterministicParallel`
: An opt-in escape hatch: code that is parallel *and* deterministic by construction (e.g., GEMM where each tile writes to disjoint output locations) wraps itself in this guard. When active, parallel execution is allowed even under `AG_DETERMINISTIC=1`.

`serial_override()`
: The unified gate checked by `parallel_for`. Returns `true` if *any* of:

  - `ScopedSerial` is active
  - We are inside a pool worker (nesting)
  - Deterministic mode is on and `ScopedDeterministicParallel` is *not* active
  - `max_threads` ≤ 1

17.5.1 Determinism vs. Performance Trade-off

Floating-point addition is not associative: $(a + b) + c \neq a + (b + c)$ in general due to rounding. When multiple threads accumulate into the same gradient buffer with different chunking boundaries, the final result depends on thread scheduling—making training non-reproducible.

AutoCalc's solution: backward passes that accumulate into shared buffers use `ScopedDeterministicParallel` or serialize entirely (e.g., cross-entropy sets grain $= B$ to force one chunk). Operations where each thread writes to *disjoint* output locations (like GEMM, where each tile owns its own C block) can safely parallelize even in deterministic mode.

Part VI

Data Loading

Chapter 18

Datasets, Examples, and DataLoader

18.1 Dataset and Example

Listing 18.1: Dataset interface

```
struct Example {
  Variable x;  // input sample
  Variable y;  // label / target
};

struct Dataset {
  virtual ~Dataset() = default;
  virtual size_t size() const = 0;
  virtual Example get(size_t index) const = 0;
};
```

A `Dataset` is an abstract interface: you implement `size()` and `get(i)` to return individual samples.

18.2 DataLoader

`DataLoader` wraps a `Dataset` and provides batched iteration:

- `batch_size`: number of samples per batch.
- `shuffle`: whether to randomize sample order each epoch.
- `drop_last`: whether to discard the final incomplete batch.

Each call to `next()` collates the next `batch_size` samples into a single `Batch` (with a leading batch dimension) using the `collate()` function, which stacks samples along a new axis 0.

18.3 Transforms

`include/ag/data/transforms.hpp` provides image preprocessing:

- `Normalize(mean, std)`: $x \to (x - \mu)/\sigma$
- `Flatten`: reshapes spatial dims to a single vector

Part VII

Python Bindings

Chapter 19

pybind11 Architecture

AutoCalc's Python interface is built with **pybind11**, a header-only library that generates CPython extension modules from C++ code.

19.1 Binding Files

`bindings/variable.cpp` The main module definition (`PYBIND11_MODULE(_backend, m)`). Binds `Variable`, all operators (`add`, `matmul`, `relu`, etc.), grad mode functions, the `nograd` context manager, and the `live_node_count()` diagnostic.

`bindings/nn.cpp` Binds the neural network classes: `Linear`, `Conv2d`, `BatchNorm2d`, `Sequential`, loss functions, and `SGD`.

`bindings/data.cpp` Binds `DataLoader`, `DataLoaderOptions`, and dataset helpers (including a concrete `MNISTDataset` class).

19.2 The Python Package Shim

`ag/__init__.py` re-exports everything from the compiled `_backend` extension:

Listing 19.1: ag/__init__.py (simplified)

```
from ag._backend import *
from ag._backend import nn, data
# Also re-exports: live_node_count, nograd, Variable, ...
```

A CMake `POST_BUILD` command copies this file into the build directory so that `PYTHONPATH=build/python` makes `import ag` work correctly.

Warning

Without the `__init__.py` copy, Python treats `build/python/ag/` as a *namespace package* (PEP 420) and silently skips the re-exports. This was a real bug that was fixed by adding the `POST_BUILD` copy command to CMakeLists.txt.

Part VIII

Build System

Chapter 20

CMake Configuration

The project uses CMake $\geq$ 3.20 with C++17.

20.1 Key Targets

`autocalc_lib` Static library containing all `src/ag/` sources. Built with `POSITION_INDEPENDENT_CODE ON` so it can be linked into the shared Python module.

`ag_python` The pybind11 module (`_backend.so`). Links `autocalc_lib` and `pybind11::module`.

`tests` C++ test executable (sanitized, debug flags).

`fast_mnist` / `fast_resnet` / `fast_lstm`]: Optimized demo executables.

20.2 Compile Flags

The Python extension and core library are built with aggressive optimization:

- `-O3`: maximum optimization level.
- `-ffast-math`: allows the compiler to reorder floating-point operations, use approximate reciprocals, and assume no NaN/Inf. This can yield 10–30% speedups for compute-bound code.
- `-fno-finite-math-only`: re-enables proper NaN/Inf handling (a subset of `-ffast-math` that's dangerous to leave on for numerical code like logsumexp).
- `-mcpu=native`: tune instruction selection for the build machine's CPU (e.g., Apple M-series NEON instructions).
- `-DNDEBUG`: disables `assert()` checks.

The test executable uses `-O0 -g -fsanitize=address,undefined` for maximum debuggability.

Part IX

Memory Management and the OOM Fix

Chapter 21

The shared_ptr Ownership Model

Every `Node` is heap-allocated and managed by `std::shared_ptr`. When an operator creates an output node, it stores `shared_ptr`s to its input nodes in the `parents` vector. This keeps inputs alive as long as the output exists (which is necessary because backward needs to read their values).

The user holds a `Variable` (which contains a `shared_ptr<Node>`) for the loss. The loss node points to its parents, which point to their parents, and so on—forming a tree of shared ownership that keeps the entire computation graph alive.

Chapter 22

The Reference Cycle Bug

22.1 The Problem

Consider the backward closure for matmul. Originally:

Listing 22.1: Buggy backward closure (creates cycle)

```
C.n->backward = [An = A.n, Bn = B.n, Cn = C.n, ...] () {
  // Cn->grad is the upstream gradient
  // Uses Cn to read the gradient
  ...
};
```

The closure captures `Cn = C.n`, a `shared_ptr<Node>` to the output node. But `C.n->backward` *is* this closure. So:

- `C.n` owns `C.n->backward` (the closure is stored in the Node)
- `C.n->backward` owns `Cn` (captured `shared_ptr`)
- `Cn` points to `C.n` (same object!)

This is a **reference cycle**. Even when the user drops their `Variable`, the reference count of the Node never reaches zero because the closure still holds a reference. The Node is leaked.

This happened in three places: `matmul`, `transpose`, and `at(begin,end)`.

22.2 The Symptom

When training on MNIST with $n = 60{,}000$ samples, each forward pass creates thousands of intermediate nodes. Without cleanup, these nodes accumulate across training steps. After a few hundred steps, the process exceeds available memory and is killed by the OS (exit code 137 / SIGKILL).

Chapter 23

The Fix

23.1 Fix 1: weak_ptr in Closures

Replace `shared_ptr` self-capture with `weak_ptr`:

Listing 23.1: Fixed backward closure

```
std::weak_ptr<Node> Cw = C.n;  // weak reference
C.n->backward = [An = A.n, Bn = B.n, Cw, ...] () {
  auto Cn = Cw.lock();  // try to promote to shared_ptr
  if (!Cn) return;       // node already freed
  // ... use Cn->grad safely ...
};
```

A `weak_ptr` observes a `shared_ptr`-managed object without contributing to the reference count. `lock()` returns a valid `shared_ptr` if the object still exists, or `nullptr` if it has been freed.

23.2 Fix 2: Post-Backward Cleanup

Even with the `weak_ptr` fix, intermediate nodes would stay alive (via the parent pointers) until the next forward pass. To eagerly free them:

Listing 23.2: Post-backward cleanup

```
// After running all backward closures:
for (auto& node : order) {
  if (node->parents.empty()) continue; // leaf
  node->backward = nullptr;  // drop closure
  node->parents.clear();     // drop parent refs
}
```

This breaks all remaining references from non-leaf nodes, allowing the entire intermediate graph to be freed immediately after backward.

23.3 Verified Results

After both fixes, training on $n = 60{,}000$ MNIST samples:

- RSS stays flat at ∼300 MB (was growing unboundedly).
- Loss decreases from ∼2.3 to ∼0.33 in one epoch.
- No OOM kill.

Chapter 24

Leak Detection Infrastructure

A global atomic counter tracks live Node objects:

Listing 24.1: Node counter

```
inline std::atomic<int64_t> g_live_node_count{0};

struct Node {
  Node()  { g_live_node_count.fetch_add(1); }
  ~Node() { g_live_node_count.fetch_sub(1); }
  // ... copy/move deleted ...
};
```

This is exposed to Python as `ag.live_node_count()`, enabling test assertions like:

Listing 24.2: Leak detection test

```
before = ag.live_node_count()
# ... run forward + backward ...
gc.collect()
after = ag.live_node_count()
assert after <= before + small_tolerance
```

Part X

Appendices

Appendix A

Complete Type Reference

Type	Header	Role
`Node`	`core/variables.hpp`	DAG node (value + grad + backward)
`Variable`	`core/variables.hpp`	User-facing tensor handle
`Module`	`nn/module.hpp`	Abstract NN layer
`Sequential`	`nn/sequential.hpp`	Layer container
`Linear`	`nn/layers/linear.hpp`	Fully connected layer
`Conv2d`	`nn/layers/conv2d.hpp`	2D convolution
`BatchNorm2d`	`nn/layers/normalization.hpp`	Batch normalization
`MaxPool2d`	`nn/layers/pooling.hpp`	Max pooling
`AvgPool2d`	`nn/layers/pooling.hpp`	Average pooling
`Dropout`	`nn/layers/dropout.hpp`	Dropout regularization
`SGD`	`nn/optim/sgd.hpp`	SGD optimizer
`Dataset`	`data/dataset.hpp`	Abstract dataset
`DataLoader`	`data/dataloader.hpp`	Batched data iterator
`NoGradGuard`	`ops/graph.hpp`	RAII grad disabler
`ThreadPool`	`parallel/pool.hpp`	Persistent worker pool

Appendix B

Operator Reference

Function	File	Forward	Backward
`add(A,B)`	`ops_elmwise.cpp`	$A + B$	$\nabla A += g,\ \nabla B += g$
`sub(A,B)`	`ops_elmwise.cpp`	$A - B$	$\nabla A += g,\ \nabla B -= g$
`mul(A,B)`	`ops_elmwise.cpp`	$A * B$	$\nabla A += g * B,\ \nabla B += g * A$
`div(A,B)`	`ops_elmwise.cpp`	A/B	$\nabla A += g/B,\ \nabla B -= gA/B^2$
`neg(x)`	`ops_elmwise.cpp`	$-x$	$\nabla x -= g$
`expv(x)`	`ops_elmwise.cpp`	e^x	$\nabla x += g \cdot e^x$
`relu(x)`	`activations.cpp`	$\max(0, x)$	$g \cdot \mathbf{1}[x > 0]$
`matmul(A,B)`	`ops_linalg.cpp`	AB	$gB^T,\ A^T g$
`transpose(A)`	`ops_linalg.cpp`	swap last 2 dims	transpose grad
`at(A,b,e)`	`ops_slice.cpp`	slice	scatter grad
`sum(x,axes)`	`reduce.cpp`	sum along axes	broadcast grad
`mean(x,axes)`	`reduce.cpp`	mean along axes	broadcast grad/n

Appendix C

Key Design Patterns

Shared ownership via `shared_ptr` All Nodes are managed by reference-counted smart pointers. Operators store parent pointers; the user holds the output. The graph stays alive exactly as long as needed.

Closures as backward functions Each operator packages its backward logic as a `std::function<void()>` that captures the necessary context (parent nodes, intermediate values) by value. This decouples the forward and backward passes.

RAII everywhere `NoGradGuard`, `ScopedSerial`, `NestedParallelGuard` all use constructor/destructor pairs to manage state transitions safely.

Thread-local storage Per-thread scratch buffers (in GEMM packing), worker IDs, and nesting flags use `thread_local` to avoid synchronization overhead.

Value semantics for Variable Copying a `Variable` is cheap (just a `shared_ptr` copy). This simplifies the API: functions can return `Variable` by value.

Appendix D

Glossary

Autograd Automatic differentiation—computing gradients by recording and replaying operations.

Backward pass The reverse traversal of the computation graph to compute gradients.

Broadcasting Extending a smaller tensor to match a larger one by replicating along size-1 dimensions.

DAG Directed Acyclic Graph—the structure of the computation graph.

GEMM General Matrix Multiply: $C \leftarrow \alpha AB + \beta C$.

Grad Short for gradient ($\partial L/\partial x$).

im2col Image-to-column: rearranging convolution input patches into a matrix for GEMM-based convolution.

Leaf node A node with no parents (typically a parameter or input).

OOM Out Of Memory—when a process exceeds available RAM.

RAII Resource Acquisition Is Initialization—a C++ idiom where constructors acquire and destructors release resources.

Reference cycle Two or more objects that reference each other via smart pointers, preventing deallocation.

RSS Resident Set Size—the amount of physical RAM used by a process.

SGEMM Single-precision GEMM (float32).

Tiling Breaking a large computation into cache-sized blocks.

Topological sort Ordering DAG nodes so each node appears after its dependencies.

VJP Vector-Jacobian Product—the fundamental operation of reverse-mode AD.

weak_ptr A C++ smart pointer that observes a `shared_ptr`-managed object without preventing its destruction.