Construindo um Framework Completo de Deep Learning em C

Do Zero ao Producao: Matematica, Algoritmos e Arquitetura

Guia Master Avancado

12 de fevereiro de 2026

Conteúdo

Prefacio

Este livro nao e um tutorial superficial. E uma jornada profunda atraves de todos os aspectos do deep learning, desde os fundamentos matematicos mais abstratos ate os detalhes de implementacao de baixo nivel em C puro.

A cada capitulo, voce nao apenas aprendera conceitos, mas implementara codigo funcional e testado. Ao final, voce tera construido um framework completo equivalente a um PyTorch simplificado, totalmente funcional e capaz de treinar redes neurais complexas.

Pre-requisitos:

- Conhecimento solido de C (ponteiros, alocacao dinamica, structs)
- Matematica basica de graduacao (calculo, algebra linear)
- Vontade de entender cada detalhe

Quanto tempo leva?
Implementar todo o framework do zero pode levar de 2 a 3 meses de trabalho dedicado. Cada capitulo contem exercicios que consolidam o aprendizado.

Capítulo 1

Fundamentos Matematicos e Teoricos do Aprendizado

1.1 O Problema Fundamental do Aprendizado

O aprendizado de maquina supervisionado pode ser formalizado como um problema de otimizacao. Dado um conjunto de dados $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ onde $\mathbf{x}_i \in \mathbb{R}^d$ e $y_i \in \mathbb{R}^k$, queremos encontrar uma funcao $f : \mathbb{R}^d \to \mathbb{R}^k$ parametrizada por θ que minimize:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{(\mathbf{x},y)\sim p_{\text{data}}}[\mathcal{L}(f(\mathbf{x};\theta), y)]$$

Na pratica, como p_{data} e desconhecida, aproximamos pelo risco empirico:

$$\theta^* \approx \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}_i;\theta), y_i)$$

1.1.1 Espaco de Hipoteses e Capacidade

Uma rede neural define um espaco de hipoteses $\mathcal{H}$. O teorema da aproximacao universal (Cybenko, 1989) estabelece que uma rede feedforward com uma camada oculta pode aproximar qualquer funcao

continua em um compacto de $\mathbb{R}^d$.

Teorema 1.1 (Aproximacao Universal)**.** *Seja σ uma funcao de ativacao nao constante, limitada e monotona. Entao o conjunto de funcoes da forma:*

$$f(\mathbf{x}) = \sum_{j=1}^{m} v_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

e denso em $C([0,1]^d)$ no espaco das funcoes continuas.

1.2 Deducao Detalhada da Funcao de Custo

1.2.1 Mean Squared Error (MSE)

Para problemas de regressao, assumimos que a variavel alvo e gerada por:

$$y = f(\mathbf{x}; \theta) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

A verossimilhanca dos dados e:

$$p(y|\mathbf{x}, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - f(\mathbf{x}; \theta))^2}{2\sigma^2}\right)$$

Tomando o log negativo:

$$-\log p(y|\mathbf{x}, \theta) = \frac{1}{2}\log(2\pi\sigma^2) + \frac{(y - f(\mathbf{x}; \theta))^2}{2\sigma^2}$$

Ignorando constantes, minimizar MSE e equivalente a maximizar a verossimilhanca sob ruido Gaussiano.

1.2.2 Cross-Entropy para Classificacao

Para classificacao binaria, assumimos $y \in \{0, 1\}$ e modelamos:

$$p(y = 1|\mathbf{x}) = \sigma(f(\mathbf{x}; \theta)) = \frac{1}{1 + e^{-f(\mathbf{x};\theta)}}$$

A verossimilhanca:

$$p(y|\mathbf{x}) = \sigma(f)^y (1 - \sigma(f))^{1-y}$$

Log-verossimilhanca negativa:

$$\mathcal{L} = -y \log(\sigma(f)) - (1-y)\log(1-\sigma(f))$$

Para classificacao multiclasse com K classes, generalizamos via softmax:

$$p(y=c|\mathbf{x}) = \frac{e^{f_c(\mathbf{x})}}{\sum_{j=1}^{K} e^{f_j(\mathbf{x})}}$$

$$\mathcal{L} = -\sum_{c=1}^{K} y_c \log p(y=c|\mathbf{x})$$

1.3 Otimizacao e Gradiente Descendente

1.3.1 Deducao do Gradiente

Queremos minimizar $J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}_i;\theta), y_i)$. A expansao em serie de Taylor de primeira ordem:

$$J(\theta + \Delta\theta) \approx J(\theta) + \nabla_\theta J(\theta)^T \Delta\theta$$

Para garantir decrescimo, escolhemos $\Delta\theta = -\eta \nabla_\theta J(\theta)$:

$$J(\theta + \Delta\theta) \approx J(\theta) - \eta \|\nabla_\theta J(\theta)\|^2 \leq J(\theta)$$

1.3.2 Analise de Convergencia

Sob hipoteses de suavidade de Lipschitz do gradiente:

$$\|\nabla J(\theta_1) - \nabla J(\theta_2)\| \leq L \|\theta_1 - \theta_2\|$$

Podemos provar que com $\eta \leq 1/L$:

$$J(\theta_{t+1}) \leq J(\theta_t) - \frac{\eta}{2}\|\nabla J(\theta_t)\|^2$$

1.4 Regra da Cadeia Vetorial e Jacobianos

O backpropagation e uma aplicacao eficiente da regra da cadeia para funcoes compostas vetoriais. Seja $f : \mathbb{R}^m \to \mathbb{R}^n$ e $g : \mathbb{R}^n \to \mathbb{R}^p$. Entao:

$$\frac{\partial(g \circ f)}{\partial \mathbf{x}} = \frac{\partial g}{\partial \mathbf{f}} \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$$

Onde $\frac{\partial g}{\partial \mathbf{f}} \in \mathbb{R}^{p \times n}$ e $\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \in \mathbb{R}^{n \times m}$.

1.4.1 Diferenca entre Gradiente e Jacobiano

Para uma funcao escalar $L : \mathbb{R}^n \to \mathbb{R}$, ∇L e um vetor. Para uma funcao vetorial $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, a derivada e uma matriz Jacobiana.

Capítulo 2

Implementacao Base em C: Estruturas e Gerenciamento de Memoria

2.1 Gerenciamento de Memoria para Tensores

A base de qualquer framework e o gerenciamento eficiente de arrays multidimensionais. Vamos implementar do zero:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>

typedef struct Tensor {
    float* data;           // Dados contiguos em memoria
    float* grad;           // Gradientes acumulados
    int* shape;            // Dimensoes do tensor (ex: [3,224,224])
    int ndim;              // Numero de dimensoes
    int size;              // Produto das dimensoes
    int refcount;          // Contagem de referencias para garbage collection

    // Para autograd (sera expandido depois)
    struct Tensor* grad_fn;
```

```
    void (*backward)(struct Tensor*);

    char requires_grad;   // Se calcula gradientes
} Tensor;
```

Listing 2.1: Estrutura base do Tensor

2.2 Funcoes de Criacao e Destruicao

```
Tensor* tensor_create(int* shape, int ndim) {
    Tensor* t = (Tensor*)malloc(sizeof(Tensor));
    t->ndim = ndim;
    t->shape = (int*)malloc(ndim * sizeof(int));

    t->size = 1;
    for(int i = 0; i < ndim; i++) {
        t->shape[i] = shape[i];
        t->size *= shape[i];
    }

    t->data = (float*)calloc(t->size, sizeof(float));
    t->grad = (float*)calloc(t->size, sizeof(float));
    t->refcount = 1;
    t->requires_grad = 1;
    t->grad_fn = NULL;

    return t;
}

void tensor_free(Tensor* t) {
    if(!t) return;

    t->refcount--;
    if(t->refcount > 0) return;

    free(t->data);
    free(t->grad);
    free(t->shape);
    free(t);
}
```

Listing 2.2: Alocacao e liberacao de memoria

2.3 Operacoes Basicas Otimizadas

Operacoes vetorizadas usando loops otimizados:

```
void tensor_add_scalar(Tensor* t, float scalar) {
    for(int i = 0; i < t->size; i++) {
        t->data[i] += scalar;
    }
}

void tensor_multiply_elementwise(Tensor* a, Tensor* b,
   Tensor* out) {
    assert(a->size == b->size && a->size == out->size);

    for(int i = 0; i < a->size; i++) {
        out->data[i] = a->data[i] * b->data[i];
    }

    if(a->requires_grad || b->requires_grad) {
        // Guardar para backpropagation
        out->requires_grad = 1;
    }
}
```

Listing 2.3: Operacoes element-wise

2.4 Produto Matricial Otimizado

Implementacao de multiplicacao de matrizes com tres loops aninhados ($O(n^3)$) - posteriormente otimizaremos com blocagem e SIMD:

```
void tensor_matmul(Tensor* a, Tensor* b, Tensor* out) {
    // Assume a: [m, k], b: [k, n], out: [m, n]
    int m = a->shape[0];
    int k = a->shape[1];
    int n = b->shape[1];

    assert(a->shape[1] == b->shape[0]);

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            float sum = 0.0f;
            for(int l = 0; l < k; l++) {
                sum += a->data[i * k + l] * b->data[l * n
    + j];
```

```
            }
            out->data[i * n + j] = sum;
        }
    }
}
```

Listing 2.4: Multiplicacao de matrizes

2.5 Estrategias de Otimizacao de Memoria

2.5.1 Pooling de Alocacao

Para evitar fragmentacao de memoria, implementamos um pool de alocacao:

```
typedef struct MemoryPool {
    void* memory;
    size_t size;
    size_t used;
} MemoryPool;

MemoryPool* pool_create(size_t size) {
    MemoryPool* pool = malloc(sizeof(MemoryPool));
    pool->memory = malloc(size);
    pool->size = size;
    pool->used = 0;
    return pool;
}

void* pool_alloc(MemoryPool* pool, size_t size) {
    if(pool->used + size > pool->size) return NULL;
    void* ptr = pool->memory + pool->used;
    pool->used += size;
    return ptr;
}
```

Listing 2.5: Alocador de memoria customizado

Capítulo 3

Rede Neural: Matematica e Implementacao Detalhada

3.1 Arquitetura de uma Rede Neural

3.1.1 Neuronio Artificial: Modelo Matematico

Um neuronio artificial implementa:

$$z = \mathbf{w}^T \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b$$

$$a = \phi(z)$$

Onde ϕ e a funcao de ativacao. As derivadas sao cruciais para back-propagation:

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, $\tanh'(z) = 1 - \tanh^2(z)$
- ReLU: $\text{ReLU}(z) = \max(0, z)$, $\text{ReLU}'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$

3.2 Backpropagation: Deducao Passo a Passo

3.2.1 Caso Simples: 2 Camadas

Considere uma rede com uma camada oculta:

$$\begin{aligned} z_1 &= W_1 x + b_1 \\ a_1 &= \sigma(z_1) \\ z_2 &= W_2 a_1 + b_2 \\ \hat{y} &= z_2 \quad \text{(regressao)} \end{aligned}$$

Funcao de custo MSE:

$$L = \frac{1}{2}(\hat{y} - y)^2$$

Derivadas da Camada de Saida

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = (\hat{y} - y) \cdot 1 \cdot a_1^T$$

$$\frac{\partial L}{\partial b_2} = (\hat{y} - y) \cdot 1 \cdot 1$$

Derivadas da Camada Oculta

$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} = (\hat{y} - y) \cdot 1 \cdot W_2^T$$

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} = \frac{\partial L}{\partial a_1} \odot \sigma'(z_1)$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot x^T$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1}$$

3.2.2 Implementacao Completa do Backprop

```c
typedef struct Layer {
    Tensor* weights;
    Tensor* bias;
    Tensor* input;
    Tensor* output;
    Tensor* z;  // Pre-ativacao
    char activation; // 's' sigmoid, 'r' relu, 't' tanh, 'n' none
} Layer;

typedef struct MLP {
    Layer** layers;
    int n_layers;
    Tensor* (*forward)(struct MLP*, Tensor*);
    void (*backward)(struct MLP*, Tensor* loss);
} MLP;

void mlp_backward(MLP* net, Tensor* loss_grad) {
    Tensor* grad = loss_grad; // dL/dy

    for(int l = net->n_layers - 1; l >= 0; l--) {
        Layer* layer = net->layers[l];

        // Gradiente dos parametros da camada atual
        if(layer->weights->requires_grad) {
            // dL/dW = input^T * grad
            tensor_matmul_transpose(layer->input, grad,
                                    layer->weights->grad);
        }

        if(layer->bias->requires_grad) {
            // dL/db = sum(grad, axis=0)
            tensor_sum_axis(grad, 0, layer->bias->grad);
        }

        // Propagar gradiente para camada anterior
        if(l > 0) {
            // dL/d(input) = grad * W^T
            Tensor* new_grad = tensor_create(...);
            tensor_matmul(grad, layer->weights, new_grad, 1); // Transpose W

            // Multiplicar pela derivada da ativacao
            if(layer->activation == 's') {
                tensor_sigmoid_prime(layer->z, layer->z_prime);
```

```
                    tensor_multiply_elementwise(new_grad,
    layer->z_prime, new_grad);
                }
                // ... outras ativacoes

                grad = new_grad;
            }
        }
}
```

Listing 3.1: Backpropagation para MLP de 2 camadas

3.3 Inicializacao de Pesos

A inicializacao correta dos pesos e critica para evitar gradientes desvanecentes/explosivos.

3.3.1 Xavier/Glorot Initialization

Para ativacoes sigmoid/tanh:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

3.3.2 He Initialization

Para ReLU:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)$$

```
void init_weights_xavier(Tensor* w, int fan_in, int
    fan_out) {
    float limit = sqrtf(6.0f / (fan_in + fan_out));
    for(int i = 0; i < w->size; i++) {
        w->data[i] = ((float)rand() / RAND_MAX) * 2 *
    limit - limit;
    }
}

void init_weights_he(Tensor* w, int fan_in) {
    float std = sqrtf(2.0f / fan_in);
```

```
10      for(int i = 0; i < w->size; i++) {
11          // Box-Muller transform
12          float u1 = (float)rand() / RAND_MAX;
13          float u2 = (float)rand() / RAND_MAX;
14          w->data[i] = sqrtf(-2.0f * logf(u1)) * cosf(2 *
    M_PI * u2) * std;
15      }
16  }
```

Listing 3.2: Inicializadores de peso

Capítulo 4

Sistema de Autograd Completo

4.1 Arquitetura do Grafo Computacional

O autograd constroi um grafo aciclico dirigido (DAG) das operacoes.

```
typedef struct AutogradNode {
    Tensor* output;
    struct AutogradNode** parents;
    int n_parents;

    // Funcao que calcula gradientes dos pais
    void (*backward)(struct AutogradNode*);

    // Metadados da operacao
    int op_type;  // 0: add, 1: mul, 2: matmul, 3: relu, ...
    void* op_ctx; // Contexto especifico da operacao
} AutogradNode;

typedef struct Tensor {
    // ... campos anteriores ...

    AutogradNode* grad_node;
    char grad_computed;  // Para evitar computacao duplicada
} Tensor;
```

Listing 4.1: Estrutura do no do grafo

4.2 Implementacao de Operacoes com Grafo

```
Tensor* tensor_mul(Tensor* a, Tensor* b) {
    Tensor* out = tensor_create_like(a);  // Assume broadcasting implementado

    // Forward
    for(int i = 0; i < out->size; i++) {
        out->data[i] = a->data[i] * b->data[i];
    }

    // Setup autograd
    if(a->requires_grad || b->requires_grad) {
        out->requires_grad = 1;
        out->grad_node = (AutogradNode*)malloc(sizeof(AutogradNode));
        out->grad_node->output = out;
        out->grad_node->parents = (AutogradNode**)malloc(2 * sizeof(AutogradNode*));
        out->grad_node->parents[0] = a->grad_node;
        out->grad_node->parents[1] = b->grad_node;
        out->grad_node->n_parents = 2;
        out->grad_node->op_type = 1;  // multiplication
        out->grad_node->backward = mul_backward;
    }

    return out;
}

void mul_backward(AutogradNode* node) {
    Tensor* out = node->output;
    Tensor* a = node->parents[0]->output;
    Tensor* b = node->parents[1]->output;

    // dL/da = dL/dout * b
    if(a->requires_grad) {
        for(int i = 0; i < a->size; i++) {
            a->grad[i] += out->grad[i] * b->data[i];
        }
    }

    // dL/db = dL/dout * a
    if(b->requires_grad) {
        for(int i = 0; i < b->size; i++) {
            b->grad[i] += out->grad[i] * a->data[i];
        }
    }
```

```
}
```

Listing 4.2: Operacao de multiplicacao com autograd

4.3 Topological Sort e Propagacao Reversa

```
void tensor_backward(Tensor* loss) {
    if(!loss->requires_grad) return;

    // Inicializa gradiente da loss como 1 (dL/dL = 1)
    for(int i = 0; i < loss->size; i++) {
        loss->grad[i] = 1.0f;
    }

    // Coleta todos os nos do grafo via DFS
    AutogradNode** nodes = NULL;
    int n_nodes = 0;
    collect_nodes(loss->grad_node, &nodes, &n_nodes);

    // Ordenacao topologica (DFS reversa)
    topological_sort(nodes, n_nodes);

    // Backward em ordem reversa
    for(int i = n_nodes - 1; i >= 0; i--) {
        if(nodes[i]->backward) {
            nodes[i]->backward(nodes[i]);
        }
    }
}

void collect_nodes(AutogradNode* node, AutogradNode***
    nodes, int* n) {
    if(!node) return;

    // Evita duplicatas
    for(int i = 0; i < *n; i++) {
        if((*nodes)[i] == node) return;
    }

    // Adiciona a lista
    *nodes = realloc(*nodes, (*n + 1) * sizeof(
    AutogradNode*));
    (*nodes)[*n] = node;
    (*n)++;

```

```
38      // Recursao nos pais
39      for(int i = 0; i < node->n_parents; i++) {
40          collect_nodes(node->parents[i], nodes, n);
41      }
42  }
```

Listing 4.3: Backward recursivo com ordenacao topologica

Capítulo 5

Camadas Avancadas e Funcoes de Ativacao

5.1 Camada Linear (Fully Connected)

```c
typedef struct Linear {
    Tensor* weight;
    Tensor* bias;
    Tensor* output;
    Tensor* input_cache;

    int in_features;
    int out_features;
} Linear;

Linear* linear_create(int in_features, int out_features) {
    Linear* layer = (Linear*)malloc(sizeof(Linear));
    layer->in_features = in_features;
    layer->out_features = out_features;

    int w_shape[] = {out_features, in_features};
    layer->weight = tensor_create(w_shape, 2);
    init_weights_he(layer->weight, in_features);

    int b_shape[] = {out_features};
    layer->bias = tensor_create(b_shape, 1);
    for(int i = 0; i < out_features; i++) layer->bias->data[i] = 0;

```

```
24      return layer;
25  }
26
27  Tensor* linear_forward(Linear* layer, Tensor* input) {
28      // input shape: [batch_size, in_features]
29      // output shape: [batch_size, out_features]
30
31      int batch_size = input->shape[0];
32      int out_shape[] = {batch_size, layer->out_features};
33
34      if(!layer->output) {
35          layer->output = tensor_create(out_shape, 2);
36      }
37
38      // y = x @ W^T + b
39      tensor_matmul(input, layer->weight, layer->output, 1,
        0); // Transpose W?
40
41      // Adiciona bias (broadcasting)
42      for(int i = 0; i < batch_size; i++) {
43          for(int j = 0; j < layer->out_features; j++) {
44              layer->output->data[i * layer->out_features +
        j] +=
45                  layer->bias->data[j];
46          }
47      }
48
49      // Cache para backward
50      layer->input_cache = input;
51
52      return layer->output;
53  }
54
55  void linear_backward(Linear* layer, Tensor* grad_output)
        {
56      // grad_output: [batch_size, out_features]
57      int batch_size = grad_output->shape[0];
58
59      // grad_weight = input^T @ grad_output
60      if(layer->weight->requires_grad) {
61          // grad_weight shape: [out_features, in_features]
62          tensor_matmul_transpose(layer->input_cache,
        grad_output,
63                                  layer->weight->grad, 1, 0)
        ;
64      }
65
66      // grad_bias = sum(grad_output, dim=0)
```

```
    if(layer->bias->requires_grad) {
        for(int j = 0; j < layer->out_features; j++) {
            float sum = 0;
            for(int i = 0; i < batch_size; i++) {
                sum += grad_output->data[i * layer->
    out_features + j];
            }
            layer->bias->grad[j] += sum;
        }
    }

    // grad_input = grad_output @ weight
    if(layer->input_cache->requires_grad) {
        tensor_matmul(grad_output, layer->weight,
                      layer->input_cache->grad, 0, 1); //
    Transpose weight
    }
}
```

Listing 5.1: Implementacao completa da camada Linear

5.2 Funcoes de Ativacao com Autograd

5.2.1 ReLU e Leaky ReLU

```
Tensor* relu_forward(Tensor* x) {
    Tensor* out = tensor_create_like(x);

    for(int i = 0; i < x->size; i++) {
        out->data[i] = x->data[i] > 0 ? x->data[i] : 0;
    }

    if(x->requires_grad) {
        out->requires_grad = 1;
        out->grad_node = autograd_node_create(OP_RELU, 1,
    &x);
        out->grad_node->backward = relu_backward;
        out->grad_node->op_ctx = (void*)x;  // Salva
    input para mascara
    }

    return out;
}

```

```c
void relu_backward(AutogradNode* node) {
    Tensor* out = node->output;
    Tensor* x = (Tensor*)node->op_ctx;
    Tensor* grad_out = out->grad;  // dL/d(out)

    if(x->requires_grad) {
        for(int i = 0; i < x->size; i++) {
            if(x->data[i] > 0) {
                x->grad[i] += grad_out->data[i];
            }
        }
    }
}
```

Listing 5.2: ReLU com autograd

5.2.2 Softmax Estavel Numericamente

Softmax pode sofrer overflow/underflow. A versao estavel subtrai o maximo:

$$p_i = \frac{e^{z_i - \max(z)}}{\sum_j e^{z_j - \max(z)}}$$

```c
Tensor* softmax_forward(Tensor* x, int dim) {
    // Assume x shape: [batch_size, n_classes]
    int batch_size = x->shape[0];
    int n_classes = x->shape[1];

    Tensor* out = tensor_create_like(x);

    for(int b = 0; b < batch_size; b++) {
        // Encontra maximo para estabilidade numerica
        float max_val = x->data[b * n_classes];
        for(int c = 1; c < n_classes; c++) {
            if(x->data[b * n_classes + c] > max_val) {
                max_val = x->data[b * n_classes + c];
            }
        }

        // Calcula exponenciais e soma
        float sum = 0;
        for(int c = 0; c < n_classes; c++) {
            int idx = b * n_classes + c;
```

```
21              out->data[idx] = expf(x->data[idx] - max_val)
    ;
22              sum += out->data[idx];
23          }
24
25          // Normaliza
26          for(int c = 0; c < n_classes; c++) {
27              out->data[b * n_classes + c] /= sum;
28          }
29      }
30
31      return out;
32  }
33
34  Tensor* softmax_backward(Tensor* out, Tensor* grad_out) {
35      // Jacobiana da softmax: p_i * (delta_ij - p_j)
36      // dL/dx_i = sum_j (dL/dp_j * p_i * (delta_ij - p_j))
37      // = p_i * (dL/dp_i - sum_j(dL/dp_j * p_j))
38
39      int batch_size = out->shape[0];
40      int n_classes = out->shape[1];
41
42      Tensor* grad_in = tensor_create_like(out);
43
44      for(int b = 0; b < batch_size; b++) {
45          // Calcula dot product: sum(dL/dp_j * p_j)
46          float dot = 0;
47          for(int j = 0; j < n_classes; j++) {
48              int idx = b * n_classes + j;
49              dot += grad_out->data[idx] * out->data[idx];
50          }
51
52          for(int i = 0; i < n_classes; i++) {
53              int idx = b * n_classes + i;
54              grad_in->data[idx] = out->data[idx] * (
    grad_out->data[idx] - dot);
55          }
56      }
57
58      return grad_in;
59  }
```

Listing 5.3: Softmax estavel e seu gradiente

Capítulo 6

Redes Convolucionais: Implementacao Profunda

6.1 Fundamentos da Convolucao Discreta

Convolucao 2D discreta para imagens:

$$(I * K)(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n) \cdot K(m,n)$$

6.1.1 Implementacao Ingênua vs Otimizada

```c
typedef struct Conv2D {
    Tensor* weight;  // [out_channels, in_channels, kernel_h, kernel_w]
    Tensor* bias;    // [out_channels]
    int stride;
    int padding;

    // Cache para backward
    Tensor* input;
    Tensor* output;
} Conv2D;

Tensor* conv2d_forward(Conv2D* layer, Tensor* input) {
    // input: [batch, in_channels, h, w]
    int batch = input->shape[0];
    int in_c = input->shape[1];
```

```
16      int in_h = input->shape[2];
17      int in_w = input->shape[3];
18
19      int out_c = layer->weight->shape[0];
20      int k_h = layer->weight->shape[2];
21      int k_w = layer->weight->shape[3];
22
23      // Calcula dimensoes de saida
24      int out_h = (in_h + 2*layer->padding - k_h) / layer->
    stride + 1;
25      int out_w = (in_w + 2*layer->padding - k_w) / layer->
    stride + 1;
26
27      int out_shape[] = {batch, out_c, out_h, out_w};
28      Tensor* out = tensor_create(out_shape, 4);
29
30      // Aplica padding se necessario
31      Tensor* input_padded = input;
32      if(layer->padding > 0) {
33          input_padded = tensor_pad(input, layer->padding);
34      }
35
36      // 6 loops: batch, out_c, out_h, out_w, k_h, k_w
37      for(int n = 0; n < batch; n++) {
38          for(int oc = 0; oc < out_c; oc++) {
39              for(int oh = 0; oh < out_h; oh++) {
40                  for(int ow = 0; ow < out_w; ow++) {
41                      float sum = layer->bias->data[oc];
42
43                      for(int ic = 0; ic < in_c; ic++) {
44                          for(int kh = 0; kh < k_h; kh++) {
45                              for(int kw = 0; kw < k_w; kw
    ++) {
46                                  int ih = oh * layer->
    stride + kh;
47                                  int iw = ow * layer->
    stride + kw;
48
49                                  int in_idx = ((n * in_c +
     ic) * in_h + ih) * in_w + iw;
50                                  int w_idx = ((oc * in_c +
     ic) * k_h + kh) * k_w + kw;
51
52                                  sum += input_padded->data
    [in_idx] *
53                                         layer->weight->
    data[w_idx];
54                              }
```

```
                        }
                    }

                    int out_idx = ((n * out_c + oc) *
    out_h + oh) * out_w + ow;
                    out->data[out_idx] = sum;
                }
            }
        }
    }

    if(layer->padding > 0) {
        tensor_free(input_padded);
    }

    return out;
}
```

Listing 6.1: Convolucao 2D com 6 loops

6.2 Backpropagation em Convolucao

O backward da convolucao e notoriamente complexo. Podemos implementar como convolucao com rotacao de kernels.

```
void conv2d_backward(Conv2D* layer, Tensor* grad_output)
    {
    // grad_output: [batch, out_c, out_h, out_w]
    int batch = grad_output->shape[0];
    int out_c = grad_output->shape[1];
    int out_h = grad_output->shape[2];
    int out_w = grad_output->shape[3];

    // grad_weight: rotacionar input e convoluir com
    grad_output
    if(layer->weight->requires_grad) {
        // Inicializa grad_weight com zeros
        for(int i = 0; i < layer->weight->size; i++) {
            layer->weight->grad[i] = 0;
        }

        for(int n = 0; n < batch; n++) {
            for(int oc = 0; oc < out_c; oc++) {
                for(int ic = 0; ic < layer->weight->shape
    [1]; ic++) {
```

```
18                      for(int kh = 0; kh < layer->weight->
    shape[2]; kh++) {
19                          for(int kw = 0; kw < layer->
    weight->shape[3]; kw++) {
20                              float sum = 0;
21
22                              for(int oh = 0; oh < out_h;
    oh++) {
23                                  for(int ow = 0; ow <
    out_w; ow++) {
24                                      int ih = oh * layer->
    stride + kh;
25                                      int iw = ow * layer->
    stride + kw;
26
27                                      int in_idx = ((n *
    layer->weight->shape[1] + ic) *
28                                                    layer->
    input->shape[2] + ih) *
29                                                    layer->
    input->shape[3] + iw;
30                                      int out_idx = ((n *
    out_c + oc) * out_h + oh) * out_w + ow;
31
32                                      sum += layer->input->
    data[in_idx] *
33                                             grad_output->
    data[out_idx];
34                                  }
35                              }
36
37                              int w_idx = ((oc * layer->
    weight->shape[1] + ic) *
38                                           layer->weight->
    shape[2] + kh) *
39                                           layer->weight->
    shape[3] + kw;
40                              layer->weight->grad[w_idx] +=
     sum;
41                          }
42                      }
43                  }
44              }
45          }
46      }
47
48      // grad_bias: soma sobre batch, altura, largura
49      if(layer->bias->requires_grad) {
```

```
        for(int oc = 0; oc < out_c; oc++) {
            float sum = 0;
            for(int n = 0; n < batch; n++) {
                for(int oh = 0; oh < out_h; oh++) {
                    for(int ow = 0; ow < out_w; ow++) {
                        int idx = ((n * out_c + oc) *
    out_h + oh) * out_w + ow;
                        sum += grad_output->data[idx];
                    }
                }
            }
            layer->bias->grad[oc] += sum;
        }
    }

    // grad_input: convolucao transposta (full
    convolution)
    if(layer->input->requires_grad) {
        // Implementar conv2d_transpose
        Tensor* grad_input = conv2d_transpose(grad_output
    , layer->weight,
                                              layer->
    stride, layer->padding);
        // Acumular em layer->input->grad
        tensor_add_to(grad_input, layer->input->grad);
        tensor_free(grad_input);
    }
}
```

Listing 6.2: Backward da convolucao

6.3 Otimizacoes: Im2Col e GEMM

A abordagem de 6 loops e muito lenta. Podemos converter convolucao em multiplicacao de matrizes:

```
Tensor* im2col(Tensor* im, int k_h, int k_w, int stride,
    int pad) {
    // im: [batch, channels, h, w]
    int batch = im->shape[0];
    int channels = im->shape[1];
    int h = im->shape[2];
    int w = im->shape[3];

    int out_h = (h + 2*pad - k_h) / stride + 1;
```

```
    int out_w = (w + 2*pad - k_w) / stride + 1;

    // Matriz de saida: [batch * out_h * out_w, channels * k_h * k_w]
    int col_shape[] = {batch * out_h * out_w, channels * k_h * k_w};
    Tensor* col = tensor_create(col_shape, 2);

    for(int b = 0; b < batch; b++) {
        for(int oh = 0; oh < out_h; oh++) {
            for(int ow = 0; ow < out_w; ow++) {
                int row_idx = (b * out_h + oh) * out_w + ow;

                for(int c = 0; c < channels; c++) {
                    for(int kh = 0; kh < k_h; kh++) {
                        for(int kw = 0; kw < k_w; kw++) {
                            int ih = oh * stride + kh - pad;
                            int iw = ow * stride + kw - pad;

                            float val = 0;
                            if(ih >= 0 && ih < h && iw >= 0 && iw < w) {
                                int im_idx = ((b * channels + c) * h + ih) * w + iw;
                                val = im->data[im_idx];
                            }

                            int col_idx = row_idx * col->shape[1] +
                                          ((c * k_h + kh) * k_w + kw);
                            col->data[col_idx] = val;
                        }
                    }
                }
            }
        }
    }

    return col;
}

Tensor* conv2d_im2col(Conv2D* layer, Tensor* input) {
    // Converte para multiplicacao de matrizes
    Tensor* col = im2col(input, layer->weight->shape[2],
```

```
                          layer->weight->shape[3],
                          layer->stride, layer->padding);

    // weight: [out_c, in_c * k_h * k_w]
    int w_shape[] = {layer->weight->shape[0],
                     layer->weight->shape[1] *
                     layer->weight->shape[2] *
                     layer->weight->shape[3]};
    Tensor* w_flat = tensor_reshape(layer->weight,
   w_shape, 2);

    // output = col @ w_flat^T
    int out_shape[] = {input->shape[0],
                       layer->weight->shape[0],
                       col->shape[0] / input->shape[0],
   1}; // Ajustar
    Tensor* out = tensor_matmul(col, w_flat, 0, 1);

    // Adicionar bias e reshape

    tensor_free(col);
    return out;
}
```

Listing 6.3: Algoritmo im2col para convolucao otimizada

Capítulo 7

Otimizadores: Teoria e Implementacao

7.1 Stochastic Gradient Descent (SGD)

7.1.1 Com e sem Momentum

Momentum acumula gradientes passados:

$$v_t = \mu v_{t-1} + \eta \nabla L(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$

```c
typedef struct SGD {
    float lr;
    float momentum;
    float weight_decay;

    // Velocidades para cada tensor
    Tensor** velocities;
    int n_params;
} SGD;

SGD* sgd_create(float lr, float momentum, float
    weight_decay) {
    SGD* opt = (SGD*)malloc(sizeof(SGD));
    opt->lr = lr;
    opt->momentum = momentum;
    opt->weight_decay = weight_decay;
```

```
16      opt->velocities = NULL;
17      opt->n_params = 0;
18      return opt;
19  }
20
21  void sgd_add_param(SGD* opt, Tensor* param) {
22      opt->n_params++;
23      opt->velocities = realloc(opt->velocities,
24                                opt->n_params * sizeof(
    Tensor*));
25
26      int idx = opt->n_params - 1;
27      opt->velocities[idx] = tensor_create_like(param);
28      // Inicializa velocidades com zero
29      memset(opt->velocities[idx]->data, 0,
30             opt->velocities[idx]->size * sizeof(float));
31  }
32
33  void sgd_step(SGD* opt) {
34      for(int i = 0; i < opt->n_params; i++) {
35          Tensor* param = opt->velocities[i]; // Na verdade
     param
36          Tensor* vel = opt->velocities[i];
37
38          for(int j = 0; j < param->size; j++) {
39              // weight decay: adiciona penalidade L2
40              float grad = param->grad[j];
41              if(opt->weight_decay > 0) {
42                  grad += opt->weight_decay * param->data[j
    ];
43              }
44
45              // momentum
46              vel->data[j] = opt->momentum * vel->data[j] +
     opt->lr * grad;
47
48              // update
49              param->data[j] -= vel->data[j];
50          }
51      }
52  }
```

Listing 7.1: SGD com momentum

7.2 Adam: Adaptive Moment Estimation

Adam combina momentum com adaptacao de taxa de aprendizado por parametro:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

```c
typedef struct Adam {
    float lr;
    float beta1;
    float beta2;
    float eps;

    Tensor** m;  // Primeiro momento
    Tensor** v;  // Segundo momento
    int* t;      // Timestep para cada parametro
    int n_params;
} Adam;

Adam* adam_create(float lr, float beta1, float beta2,
   float eps) {
    Adam* opt = (Adam*)malloc(sizeof(Adam));
    opt->lr = lr;
    opt->beta1 = beta1;
    opt->beta2 = beta2;
    opt->eps = eps;
    opt->m = NULL;
    opt->v = NULL;
    opt->t = NULL;
    opt->n_params = 0;
    return opt;
}

void adam_add_param(Adam* opt, Tensor* param) {
    opt->n_params++;
    opt->m = realloc(opt->m, opt->n_params * sizeof(
   Tensor*));
```

```
29      opt->v = realloc(opt->v, opt->n_params * sizeof(
    Tensor*));
30      opt->t = realloc(opt->t, opt->n_params * sizeof(int))
    ;
31
32      int idx = opt->n_params - 1;
33      opt->m[idx] = tensor_create_like(param);
34      opt->v[idx] = tensor_create_like(param);
35      memset(opt->m[idx]->data, 0, opt->m[idx]->size *
    sizeof(float));
36      memset(opt->v[idx]->data, 0, opt->v[idx]->size *
    sizeof(float));
37      opt->t[idx] = 0;
38  }
39
40  void adam_step(Adam* opt) {
41      for(int i = 0; i < opt->n_params; i++) {
42          Tensor* param = /* ... */;
43          Tensor* m = opt->m[i];
44          Tensor* v = opt->v[i];
45          opt->t[i]++;
46
47          float bias_correction1 = 1.0f - powf(opt->beta1,
    opt->t[i]);
48          float bias_correction2 = 1.0f - powf(opt->beta2,
    opt->t[i]);
49
50          for(int j = 0; j < param->size; j++) {
51              float g = param->grad[j];
52
53              // Atualiza momentos
54              m->data[j] = opt->beta1 * m->data[j] + (1 -
    opt->beta1) * g;
55              v->data[j] = opt->beta2 * v->data[j] + (1 -
    opt->beta2) * g * g;
56
57              // Correcao de vies
58              float m_hat = m->data[j] / bias_correction1;
59              float v_hat = v->data[j] / bias_correction2;
60
61              // Atualiza parametro
62              param->data[j] -= opt->lr * m_hat / (sqrtf(
    v_hat) + opt->eps);
63          }
64      }
65  }
```

Listing 7.2: Implementacao completa do Adam

Capítulo 8

Treinamento e Validacao

8.1 DataLoader e Batching

```c
typedef struct Dataset {
    Tensor* data;
    Tensor* targets;
    int size;
} Dataset;

typedef struct DataLoader {
    Dataset* dataset;
    int batch_size;
    int* indices;
    int current_idx;
    int shuffle;
} DataLoader;

DataLoader* dataloader_create(Dataset* dataset, int batch_size, int shuffle) {
    DataLoader* dl = (DataLoader*)malloc(sizeof(DataLoader));
    dl->dataset = dataset;
    dl->batch_size = batch_size;
    dl->shuffle = shuffle;
    dl->current_idx = 0;

    dl->indices = (int*)malloc(dataset->size * sizeof(int));
    for(int i = 0; i < dataset->size; i++) {
        dl->indices[i] = i;
    }
```

```
26
27      if(shuffle) {
28          for(int i = dataset->size - 1; i > 0; i--) {
29              int j = rand() % (i + 1);
30              int temp = dl->indices[i];
31              dl->indices[i] = dl->indices[j];
32              dl->indices[j] = temp;
33          }
34      }
35
36      return dl;
37  }
38
39  int dataloader_next_batch(DataLoader* dl, Tensor**
        batch_data, Tensor** batch_targets) {
40      if(dl->current_idx >= dl->dataset->size) {
41          dl->current_idx = 0;
42          if(dl->shuffle) {
43              // Reshuffle
44          }
45          return 0;
46      }
47
48      int batch_end = dl->current_idx + dl->batch_size;
49      if(batch_end > dl->dataset->size) {
50          batch_end = dl->dataset->size;
51      }
52      int current_batch_size = batch_end - dl->current_idx;
53
54      // Alocar tensores do batch
55      int data_shape[] = {current_batch_size,
56                          dl->dataset->data->shape[1]}; //
        Simplificado
57      *batch_data = tensor_create(data_shape, 2);
58
59      int target_shape[] = {current_batch_size,
60                            dl->dataset->targets->shape
        [1]};
61      *batch_targets = tensor_create(target_shape, 2);
62
63      // Copiar dados
64      for(int i = 0; i < current_batch_size; i++) {
65          int idx = dl->indices[dl->current_idx + i];
66
67          // Copiar features
68          for(int j = 0; j < dl->dataset->data->shape[1]; j
        ++) {
```

```
69              (*batch_data)->data[i * dl->dataset->data->
    shape[1] + j] =
70                  dl->dataset->data->data[idx * dl->dataset
    ->data->shape[1] + j];
71          }
72
73          // Copiar targets
74          for(int j = 0; j < dl->dataset->targets->shape
    [1]; j++) {
75              (*batch_targets)->data[i * dl->dataset->
    targets->shape[1] + j] =
76                  dl->dataset->targets->data[idx * dl->
    dataset->targets->shape[1] + j];
77          }
78      }
79
80      dl->current_idx = batch_end;
81      return current_batch_size;
82  }
```

Listing 8.1: DataLoader com shuffle

8.2 Loop de Treinamento Completo

```
1   typedef struct Trainer {
2       MLP* model;
3       Optimizer* optimizer;
4       float (*loss_fn)(Tensor*, Tensor*);
5       void (*loss_backward)(Tensor*, Tensor*);
6
7       float train_loss;
8       float val_loss;
9       float train_acc;
10      float val_acc;
11  } Trainer;
12
13  void trainer_train_epoch(Trainer* trainer, DataLoader*
    train_loader) {
14      float total_loss = 0;
15      int total_correct = 0;
16      int total_samples = 0;
17
18      while(1) {
19          Tensor* batch_x;
20          Tensor* batch_y;
```

```
21          int batch_size = dataloader_next_batch(
    train_loader, &batch_x, &batch_y);
22          if(batch_size == 0) break;
23
24          // Forward pass
25          Tensor* output = mlp_forward(trainer->model,
    batch_x);
26
27          // Compute loss
28          float loss = trainer->loss_fn(output, batch_y);
29          total_loss += loss * batch_size;
30
31          // Compute accuracy
32          int correct = compute_accuracy(output, batch_y);
33          total_correct += correct;
34          total_samples += batch_size;
35
36          // Backward pass
37          trainer->loss_backward(output, batch_y);
38          tensor_backward(output);
39
40          // Update weights
41          optimizer_step(trainer->optimizer);
42
43          // Zero gradients
44          optimizer_zero_grad(trainer->optimizer);
45
46          // Cleanup
47          tensor_free(batch_x);
48          tensor_free(batch_y);
49          tensor_free(output);
50      }
51
52      trainer->train_loss = total_loss / total_samples;
53      trainer->train_acc = (float)total_correct /
    total_samples;
54
55      printf("Train Loss: %.4f, Train Acc: %.4f\n",
56             trainer->train_loss, trainer->train_acc);
57  }
58
59  void trainer_validate(Trainer* trainer, DataLoader*
    val_loader) {
60      // Similar ao train, mas sem gradientes e updates
61      // ...
62  }
```

Listing 8.2: Training loop profissional

Capítulo 9

Serializacao e Persistencia de Modelos

9.1 Salvando e Carregando Modelos

Formato binario portavel com metadados:

```
typedef struct ModelHeader {
    char magic[4];  // "DLFW"
    int version;
    int n_layers;
    int model_type;
    int data_size;
} ModelHeader;

void model_save(MLP* model, const char* filename) {
    FILE* fp = fopen(filename, "wb");

    // Header
    ModelHeader header = {
        .magic = {'D','L','F','W'},
        .version = 1,
        .n_layers = model->n_layers,
        .model_type = 0,
        .data_size = 0
    };

    // Calcular tamanho total
    for(int i = 0; i < model->n_layers; i++) {
        if(model->layers[i]->weights) {
            header.data_size += model->layers[i]->weights
```

```
    ->size * sizeof(float);
25              header.data_size += model->layers[i]->bias->
    size * sizeof(float);
26          }
27      }
28
29      fwrite(&header, sizeof(ModelHeader), 1, fp);
30
31      // Salvar arquitetura
32      for(int i = 0; i < model->n_layers; i++) {
33          Layer* layer = model->layers[i];
34
35          // Tipo da camada
36          int layer_type = 0; // 0: linear, 1: conv2d, etc.
37          fwrite(&layer_type, sizeof(int), 1, fp);
38
39          // Parametros da camada
40          int in_features = layer->weights->shape[1];
41          int out_features = layer->weights->shape[0];
42          fwrite(&in_features, sizeof(int), 1, fp);
43          fwrite(&out_features, sizeof(int), 1, fp);
44
45          // Ativacao
46          fwrite(&layer->activation, sizeof(char), 1, fp);
47
48          // Pesos e bias
49          fwrite(layer->weights->data, sizeof(float),
50                 layer->weights->size, fp);
51          fwrite(layer->bias->data, sizeof(float),
52                 layer->bias->size, fp);
53      }
54
55      fclose(fp);
56  }
57
58  MLP* model_load(const char* filename) {
59      FILE* fp = fopen(filename, "rb");
60
61      // Ler header
62      ModelHeader header;
63      fread(&header, sizeof(ModelHeader), 1, fp);
64
65      if(header.magic[0] != 'D' || header.magic[1] != 'L'
    ||
66         header.magic[2] != 'F' || header.magic[3] != 'W')
    {
67          printf("Formato de arquivo invalido\n");
68          return NULL;
```

```
    }

    // Reconstruir modelo
    MLP* model = mlp_create();

    for(int i = 0; i < header.n_layers; i++) {
        int layer_type;
        fread(&layer_type, sizeof(int), 1, fp);

        int in_features, out_features;
        fread(&in_features, sizeof(int), 1, fp);
        fread(&out_features, sizeof(int), 1, fp);

        char activation;
        fread(&activation, sizeof(char), 1, fp);

        Layer* layer = layer_create(in_features,
   out_features);
        layer->activation = activation;

        // Carregar pesos
        fread(layer->weights->data, sizeof(float),
              layer->weights->size, fp);
        fread(layer->bias->data, sizeof(float),
              layer->bias->size, fp);

        mlp_add_layer(model, layer);
    }

    fclose(fp);
    return model;
}
```

Listing 9.1: Save/Load completo

Capítulo 10

Estudos de Caso e Projetos Praticos

10.1 Exemplo 1: MNIST do Zero

Implementacao completa de reconhecimento de digitos:

```
int main() {
    // 1. Carregar dados
    Dataset* train_dataset = mnist_load("train-images.idx3-ubyte",
                                        "train-labels.idx1-ubyte");
    Dataset* test_dataset = mnist_load("t10k-images.idx3-ubyte",
                                       "t10k-labels.idx1-ubyte");

    // 2. Criar modelo
    MLP* model = mlp_create();
    mlp_add_layer(model, linear_create(784, 256));
    mlp_add_layer(model, relu_create());
    mlp_add_layer(model, linear_create(256, 128));
    mlp_add_layer(model, relu_create());
    mlp_add_layer(model, linear_create(128, 10));
    mlp_add_layer(model, softmax_create());

    // 3. Configurar treinamento
    Adam* optimizer = adam_create(0.001, 0.9, 0.999, 1e-8);
    mlp_add_parameters(model, optimizer);
```

```
20
21      Trainer* trainer = trainer_create(model, optimizer,
22                                        cross_entropy_loss,
23
    cross_entropy_backward);
24
25      // 4. Loop de treinamento
26      DataLoader* train_loader = dataloader_create(
    train_dataset, 64, 1);
27      DataLoader* test_loader = dataloader_create(
    test_dataset, 64, 0);
28
29      for(int epoch = 0; epoch < 10; epoch++) {
30          trainer_train_epoch(trainer, train_loader);
31          trainer_validate(trainer, test_loader);
32
33          printf("Epoch %d: Train Acc: %.4f, Test Acc: %.4f
    \n",
34                 epoch, trainer->train_acc, trainer->
    val_acc);
35      }
36
37      // 5. Salvar modelo treinado
38      model_save(model, "mnist_model.dlfw");
39
40      return 0;
41  }
```

Listing 10.1: MNIST pipeline completo

10.2 Exemplo 2: CNN para CIFAR-10

```
1   Model* create_cifar10_cnn() {
2       Model* model = model_create();
3
4       // Conv1: 32x32x3 -> 32x32x32
5       model_add(model, conv2d_create(3, 32, 3, 1, 1));
6       model_add(model, batch_norm_create(32));
7       model_add(model, relu_create());
8       model_add(model, maxpool2d_create(2, 2));
9
10      // Conv2: 16x16x32 -> 16x16x64
11      model_add(model, conv2d_create(32, 64, 3, 1, 1));
12      model_add(model, batch_norm_create(64));
13      model_add(model, relu_create());
```

```
    model_add(model, maxpool2d_create(2, 2));

    // Conv3: 8x8x64 -> 8x8x128
    model_add(model, conv2d_create(64, 128, 3, 1, 1));
    model_add(model, batch_norm_create(128));
    model_add(model, relu_create());

    // Global average pooling
    model_add(model, global_avgpool2d_create());

    // Classifier
    model_add(model, linear_create(128, 256));
    model_add(model, relu_create());
    model_add(model, dropout_create(0.5));
    model_add(model, linear_create(256, 10));
    model_add(model, softmax_create());

    return model;
}
```

Listing 10.2: CNN completa para CIFAR-10

Capítulo 11

Otimizacoes Avancadas e Performance

11.1 BLAS-Level Optimizations

11.1.1 Tiling para Cache L1/L2

```
#define BLOCK_SIZE 32

void matmul_tiled(Tensor* a, Tensor* b, Tensor* out,
                  int M, int N, int K) {

    for(int i = 0; i < M; i += BLOCK_SIZE) {
        for(int j = 0; j < N; j += BLOCK_SIZE) {
            for(int k = 0; k < K; k += BLOCK_SIZE) {

                // Processa block (i,i+BS) x (k,k+BS) * (k,k+BS) x (j,j+BS)
                for(int ii = i; ii < i + BLOCK_SIZE && ii < M; ii++) {
                    for(int jj = j; jj < j + BLOCK_SIZE && jj < N; jj++) {
                        float sum = out->data[ii * N + jj];

                        for(int kk = k; kk < k + BLOCK_SIZE && kk < K; kk++) {
                            sum += a->data[ii * K + kk] * b->data[kk * N + jj];
                        }
```

```
                            out->data[ii * N + jj] = sum;
                        }
                    }
                }
            }
        }
}
```

Listing 11.1: Multiplicacao de matrizes com tiling

11.1.2 SIMD Intrinsics

```
#include <immintrin.h>

void tensor_add_simd(float* a, float* b, float* out, int
    n) {
    int i = 0;

    #ifdef __AVX__
    // Processa 8 floats por vez com AVX
    for(; i <= n - 8; i += 8) {
        __m256 va = _mm256_loadu_ps(&a[i]);
        __m256 vb = _mm256_loadu_ps(&b[i]);
        __m256 vout = _mm256_add_ps(va, vb);
        _mm256_storeu_ps(&out[i], vout);
    }
    #elif __SSE__
    // Processa 4 floats por vez com SSE
    for(; i <= n - 4; i += 4) {
        __m128 va = _mm_loadu_ps(&a[i]);
        __m128 vb = _mm_loadu_ps(&b[i]);
        __m128 vout = _mm_add_ps(va, vb);
        _mm_storeu_ps(&out[i], vout);
    }
    #endif

    // Processa o restante
    for(; i < n; i++) {
        out[i] = a[i] + b[i];
    }
}
```

Listing 11.2: Usando SSE/AVX para operacoes vetoriais

11.2 Parallelism with OpenMP

```
#include <omp.h>

void conv2d_parallel(Conv2D* layer, Tensor* input, Tensor
    * output) {
     int batch = input->shape[0];
     int out_c = output->shape[1];
     int out_h = output->shape[2];
     int out_w = output->shape[3];

     #pragma omp parallel for collapse(4) schedule(dynamic
    )
     for(int n = 0; n < batch; n++) {
         for(int oc = 0; oc < out_c; oc++) {
             for(int oh = 0; oh < out_h; oh++) {
                 for(int ow = 0; ow < out_w; ow++) {
                     // Calculo da convolucao
                     // ...
                 }
             }
         }
     }
}
```

Listing 11.3: Paralelizacao de operacoes

Capítulo 12

Proximos Passos: Alem do Framework

12.1 Suporte a GPU com CUDA

```
#ifdef USE_CUDA
#include <cuda_runtime.h>

__global__ void relu_kernel(float* x, float* out, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) {
        out[idx] = x[idx] > 0 ? x[idx] : 0;
    }
}

void relu_cuda(Tensor* x, Tensor* out) {
    float* d_x, d_out;

    // Alocar memoria na GPU
    cudaMalloc(&d_x, x->size * sizeof(float));
    cudaMalloc(&d_out, out->size * sizeof(float));

    // Copiar dados para GPU
    cudaMemcpy(d_x, x->data, x->size * sizeof(float),
               cudaMemcpyHostToDevice);

    // Lancar kernel
    int block_size = 256;
```

```
    int num_blocks = (x->size + block_size - 1) /
    block_size;
    relu_kernel<<<num_blocks, block_size>>>(d_x, d_out, x
    ->size);

    // Copiar resultado de volta
    cudaMemcpy(out->data, d_out, out->size * sizeof(float
    ),
               cudaMemcpyDeviceToHost);

    // Liberar memoria
    cudaFree(d_x);
    cudaFree(d_out);
}
#endif
```

Listing 12.1: Interface CUDA para operacoes

12.2 Implementacao de Transformers

Arquitetura completa de Transformer para processamento de sequencias:

- Multi-Head Self-Attention
- Positional Encoding
- Feed-Forward Networks
- Layer Normalization
- Residual Connections

12.3 Mixed Precision Training

Implementacao de treinamento com FP16 para maior performance:

```
void convert_to_fp16(float* fp32, __fp16* fp16, int n) {
    for(int i = 0; i < n; i++) {
        fp16[i] = (__fp16)fp32[i];
    }
}
```

```
6
7  void convert_to_fp32(__fp16* fp16, float* fp32, int n) {
8      for(int i = 0; i < n; i++) {
9          fp32[i] = (float)fp16[i];
10     }
11 }
12
13 void matmul_mixed(Tensor* a_fp32, Tensor* b_fp32, Tensor*
       out_fp32) {
14     // Converter para FP16
15     __fp16* a_fp16 = malloc(a_fp32->size * sizeof(__fp16)
       );
16     __fp16* b_fp16 = malloc(b_fp32->size * sizeof(__fp16)
       );
17
18     convert_to_fp16(a_fp32->data, a_fp16, a_fp32->size);
19     convert_to_fp16(b_fp32->data, b_fp16, b_fp32->size);
20
21     // Multiplicacao em FP16 (usando hardware nativo)
22     __fp16* out_fp16 = malloc(out_fp32->size * sizeof(
       __fp16));
23     matmul_fp16(a_fp16, b_fp16, out_fp16, /* dims */);
24
25     // Converter de volta
26     convert_to_fp32(out_fp16, out_fp32->data, out_fp32->
       size);
27
28     free(a_fp16);
29     free(b_fp16);
30     free(out_fp16);
31 }
```

Listing 12.2: Cast de precisao mista

Conclusao

Voce agora tem em maos um framework completo de deep learning implementado em C puro, com:

- Sistema de tensores com gerenciamento de memoria
- Autograd com grafo computacional dinamico
- Camadas: Linear, Conv2D, BatchNorm, Dropout
- Ativacoes: ReLU, Sigmoid, Tanh, Softmax
- Otimizadores: SGD, Adam
- Funcoes de custo: MSE, Cross-Entropy
- DataLoaders e batching
- Serializacao de modelos
- Otimizacoes de performance: SIMD, OpenMP, CUDA

Este codigo e completamente funcional e pode ser usado para treinar redes neurais reais em problemas como MNIST, CIFAR-10, e ate mesmo tarefas mais complexas.

O proximo passo e expandir para arquiteturas modernas como Transformers, GANs, e modelos de difusao, sempre mantendo a filosofia de implementar cada detalhe do zero para compreensao profunda.

A jornada continua...