Projeto Final: RaspNN

PCS3732

João Pedro Cabral Miranda	12551237
Rafael de Almeida Innecco	12550535
Pedro Hrosz Turini	12551133

18 de agosto de 2024

Sumário

1 Introdução

Esse projeto tem como objetivo a elaboração um rede neural com aceleração de hardware para operações vetoriais fornecida pela arquitetura ARM NEON, de forma que ela possa ser utilizada e manipulada de forma remota, tanto para o treinamento quanto para a realização de inferências por diferentes usuários.

O hardware utilizado será um Raspberry Pi 3B+, que é dotado de um processador Broadcom BCM2837B0 (arquitetura ARM Cortex A53), com o auxílio do processador vetorial ARM NEON (capaz de processar até 128 bits paralelamente). Para que os recursos computacionais do ARM NEON sejam aproveitados, foi utilizada a biblioteca <arm_neon.h> (também conhecida como ARM NEON *Intrinsics*), que contém *wrappers* para cada uma das instruções da extensão do ISA que utilizam esse recurso.

A Rede Neural foi desenvolvida na linguagem C com apenas duas camadas (entrada e saída, sem camadas escondidas) utilizando a função *softmax* como função de ativação. A aplicação escolhida para a rede neural foi o reconhecimento de dígitos manuscritos de 0 a 9 do conjunto de dados do MNIST.

Para realizar a comunicação com o Raspberry Pi foi criada uma aplicação Cliente-Servidor - também desenvolvida na linguagem C - na qual o Raspberry Pi terá o papel de Servidor, enquanto o Cliente será um segundo dispositivo operado pelo usuário. Essa aplicação Cliente-Servidor utiliza a API Socket para abstrair o protocolo TCP.

Nessa comunicação o cliente enviará os dados puros, ou seja, qualquer tratamento de leitura de arquivos deve ser feito pelo usuário. Em especial, será desenvolvido um leitor de arquivos *idx1-ubyte* e *idx3-ubyte* para obter os conjuntos de dados de treino e de teste desenvolvidos pela MNIST.

2 ARM NEON: Aceleração em Hardware

Essa seção tem como principal objetivo discutir sobre o ARM NEON, sua API *Intrinsics* e as funções que foram aceleradas com esse processador vetorial.

Como discutido na seção 1, esse projeto tem como objetivo acelerar uma rede neural utilizando o Raspberry Pi. A escolha dessa aplicação para explorar o uso do ARM NEON se deve ao fato de que o principal gargalo da rede neural é a realização massiva de operações matriciais (por exemplo, na operação de propagação direta), logo a aceleração será realizada em torno do princípio do DLP (Data Level Parallelism).

Inicialmente, foram consideradas duas opções para a implementação do paralelismo a nível de dados, o uso da GPU VideoVCore presente no Raspberry Pi 3B+, e o processador vetorial NEON. Contudo, mediante a inexistência de documentação oficial sobre o uso da GPU, e uma série de problemas relacionados à compilação dos binários para esse propósito, foi feita a decisão de seguir o projeto com o ARM NEON.

Essa unidade computacional é um processador vetorial da ARM contido dentro do fluxo de dados principal do Cortex-A53 capaz de operar sobre registradores vetoriais de 64 ou 128 bits, com suporte a load-store espaçados em até 4 bytes (*scatter-gather*). Para acessar as funcionalidades do ARM NEON é necessário usar instruções de máquina específicas. Com o intuito de facilitar o desenvolvimento e tornar o código da rede neural mais portável e elegante foi utilizado o ARM NEON *Intrinsics* que é uma API em C para o ARM NEON, na qual cada função da API é decodificada em uma ou mais instruções desse processador vetorial.

Contudo, houve dois problemas principais na compilação cruzada de códigos C que usam a API *Intrinsics*. O primeiro problema foi que o compilador cruzado *bare-metal* não substituía corretamente as funções do *Intrinsics* com as instruções assembly esperadas, tentou-se utilizar outras versões do compilador *bare-metal* arm-none-eabi-gcc, mas o problema ainda sim não foi resolvido. O segundo problema está no uso da *newlib* como biblioteca padrão, pois a *newlib* utiliza ponto-flutuante em software, mas para utilizar o *Intrinsics* é necessário utilizar ponto-flutuante em hardware, com isso culminando em um erro de ligação entre os binários da rede neural e a *newlib*. Para resolver esse problema tentou-se utilizar outras versões da *newlib* para ponto-flutuante em hardware além de compiladores *multilib*, todavia ambas as tentativas não foram bem sucedidas.

Desse modo, para solucionar esses dois problemas tomou-se a decisão de utilizar um Sistema Operacional no Raspberry Pi (Raspberry Pi OS Lite), na qual utilizando um compilador nativo foi possível compilar com êxito a API *Intrinsics*. Contudo, ainda sim não foi possível realizar a compilação cruzada com êxito, mesmo utilizando versões do compilador com suporte a ponto flutuante em hardware, pelos mesmos dois problemas citados acima. Com isso, tomou-se a decisão de realizar apenas a compilação nativa, com a aplicação executada pelo sistema operacional.

2.1 Processadores Vetoriais

Nesta seção será discutido os conceitos de processador vetorial, DLP e SIMD, usados como base no desenvolvimento do ARM NEON.

Primeiramente, o conceito de DLP (Data Level Parallelism) trata-se de explorar paralelismo atráves do dados, isto é, processar múltiplos dados paralelamente para maximizar o uso dos recursos computacionais e diminuir o tempo de execução dos programas. Alguns exemplos de aplicações em que esse conceito pode ser utilizado são: Operações Matriciais e Aplicações Multimídia.

Uma das implementações possíveis para o DLP é por meio do paradigma SIMD, na qual uma única instrução do processador realiza a mesma operação sob diversos dados paralelamente. Uma das principais vantagens dessa abordagem é sua praticidade em prover paralelismo para um programa sequencial sem que seja necessário realizar mudanças abruptas em sua lógica.

Ademais, um processador vetorial é um processador que implementa o paradigma SIMD para explorar DLP e com isso reduzir o tempo de execução dos programas. Para tal, um processador vetorial deve ser capaz de unir dados espalhados pela memória em grandes bancos de registradores, realizar operações paralelamente em múltiplos registros desse banco e devolvê-los corretamente na memória.

Um processador vetorial tem quatro componentes principais: o banco de registradores vetorial (onde cada entrada é capaz de armazenar um vetor cujos elementos serão processados paralelamente), unidades funcionais vetoriais (capazes de processar os elementos dos vetores paralelamente), unidade de escrita e leitura da memória (responsável por gerir a lógica de leitura/escrita esparsa) e o banco de registradores escalar (onde cada entrada contem apenas um elemento escalar que pode ser aplicado a todos os elementos de um vetor em uma determinada operação).

No caso do ARM NEON, o banco de registradores vetorial pode ser interpretado como 32 registradores de 64 bits ou 16 registradores de 128 bits. Vale ressaltar que a metade inferior desse banco é compartilhado com a Unidade de Ponto Flutuante, na qual ela interpreta esse trecho como 32 registradores de 32 bits ou 16 registradores de 64 bits. Suas unidades funcionais são capazes de realizar diversas operações entre elas, destacam-se soma, subtração, multiplicação, máximo, mínimo e permutação. Já as unidades de escrita e leitura são capazes de realizar o acesso a dados em endereços de memória não contíguos, onde cada acesso pode pular até 4 posições na sequência. A respeito do banco de registradores escalar, ele pode utilizar tanto os registros do banco de registradores de inteiros quanto os do banco de registradores de ponto flutuante.

Com relação as unidades funcionais vetoriais, vale ressaltar que cada uma delas têm uma pipeline própria, dessa forma elas podem iniciar uma operação a cada ciclo de clock.

Além disso, um processador vetorial tem mecanismos para alterar dinamicamente o tamanho dos elementos de um vetor, seguindo certas restrições. No caso do ARM NEON, ele é capaz de interpretar que os elementos de um vetor tem 64, 32, 16 ou 8 bits.

Ademais, defina-se como pista o trecho em pipeline de uma unidade funcional capaz de operar em um único elemento.

Outra característica relevante dos processadores vetoriais é a capacidade de lidar com desvios no programa. Alguns processadores implementam o conceito de predicado, onde utiliza-se uma máscara para definir se uma determinada operação acontecerá ou não para um determinado elemento de um vetor. Contudo, ao utilizar esse mecanismo há uma diminuição do uso efetivo das unidades funcionais, pois caso a operação não aconteça em um elemento, então a respectiva pista responsável por esse elemento ficará inoperante nesse ciclo de clock. Todavia, o ARM NEON utiliza outra estratégia mais simples para lidar com desvios, trata-se do operador ternário; na qual utiliza-se uma máscara para escolher entre dois valores qual será escrito no registrador de destino. Essa abordagem é mais simples, pois tira a necessidade de um mecanismo para desativar as pistas das unidades funcionais, contudo torna-se muito mais complexo para o programador lidar com desvios em sua aplicação, de certa forma reduzindo a praticidade oferecida pelo SIMD para o uso do paralelismo a nível de dados.

Por fim, vale ressaltar que os processadores vetoriais permitem que haja encaminhamento de dados entre as unidades funcionais, com isso permitindo que haja diferentes instruções executando em diferentes unidades funcionais com pipeline simultâneamente. Desse modo, a latência de uma instrução pode esconder o tempo de execução de outras, com isso reduzindo o tempo total do programa.

2.2 ARM NEON *Intrinsincs*

Nessa seção serão listadas as funções da biblioteca ARM NEON *Intrinsics* utilizadas nesse projeto, sendo que para cada função será exibido um exemplo das instruções assembly geradas após a compilação, vale notar que dependendo dos argumentos utilizados nessas funções e as otimizações feitas pelo compilador, as instruções decodificadas podem variar. Vale ressaltar que tudo que será cobrido nessa seção se refere as instruções do NEON existentes nos processadores v7, com isso desconsiderando as extensões do NEON para as arquiteturas seguintes (A32 e A64).

Primeiramente, cada função dessa biblioteca obedece a seguinte nomenclatura:

v<op><q>_<lane><n_><x><y>

onde <op> representa a operação que será realizada, <q> representa que os registradores vetoriais utilizados têm 128 bits (em sua ausência eles têm 64 bits), <lane> indica que a operação será feita em uma única pista (em sua ausência a operação é feita em todas as pistas), <n_> indica que um dos operandos base é um registrador escalar (em sua ausência todos os operandos base são registradores vetoriais), <x> indica o tipo dos dados contidos nos registradores, seus valores possíveis são s (*signed integer*), u (*unsigned integer*), f (*float-point*), bf(*brain float-point*) e p (*polynomial*), <y> indica o tamanho dos dados contidos no registrador podendo variar entre 8, 16, 32 e 64. Note que, o número de dados contidos em um registrador será dado pela divisão entre o número de bits (determinado por <q>) contidos no registradores e o tamanho de cada elemento (determinado por <y>).

A ARM NEON *Intrinsics* também define novos tipos de dados para permitir a manipulação desses registradores, como o *float32x4_t* e o *float32_t* que se referem respectivamente a um registrador vetorial de 128 bits com 4 floats de 32 bits e um registrador escalar contendo um float de 32 bits. Outros tipos de dados novos são *uint16x8_t* e *int64x2_t*.

Em especial, esse projeto sempre utilizará o <q> (para usufruir de registradores maiores), <y> igual a 32 (logo sempre haverão 4 elementos em cada vetor) e na maioria das ocasiões <x> valerá f (em alguns casos valerá s), pois grande parte das operações utilizadas são em ponto flutuante.

As funções utilizadas do *Intrinsics* podem ser vistas a seguir:

- **vaddq_f32**: Realiza a adição elemento a elemento de dois vetores de quatro elementos de ponto flutuante.
  Instrução NEON: `FADD Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vaddq_s32**: Realiza a adição elemento a elemento de dois vetores de quatro elementos inteiros.
  Instrução NEON: `ADD Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vsubq_f32**: Realiza a subtração elemento a elemento de dois vetores de quatro elementos de ponto flutuante.
  Instrução NEON: `FSUB Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vmulq_f32**: Realiza a multiplicação elemento a elemento de dois vetores de quatro elementos de ponto flutuante.
  Instrução NEON: `FMUL Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vmulq_n_f32**: Multiplica cada elemento de um vetor de quatro elementos de ponto flutuante por um valor escalar.
  Instrução NEON: `FMUL Vd.4s, Vn.4s, Vm.s[0]`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem), Vm.s[0] (escalar interpretado como um vetor)

- **vmlaq_n_f32**: Realiza uma multiplicação elemento a elemento de um vetor com um escalar e acumula o resultado em outro vetor. Essa função é decodificada nas duas instruções a seguir.
  Instrução NEON: `FMUL Vn.4s, Vn.4s, Vm.s[0]` e `FADD Vd.4s, Vd.4s, Vn.4s`
  Registradores: Vd.4s (vetor de destino e acumulador), Vn.4s (vetor de origem), Vm.s[0] (escalar interpretado como um vetor)

- **vfmaq_f32**: Realiza uma multiplicação de dois vetores de quatro elementos de ponto flutuante e acumula o resultado em um terceiro vetor. Sendo que, ela não realiza nenhuma aproximação após a multiplicação, apenas após a soma, assim tornando-a mais rápida do que uma multiplicação com acumulador comum, mas também menos precisa.
  Instrução NEON: `FMLA Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino e acumulador), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vmaxq_f32**: Calcula o máximo elemento a elemento entre dois vetores de quatro elementos de ponto flutuante.
  Instrução NEON: `FMAX Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vminq_f32**: Calcula o mínimo elemento a elemento entre dois vetores de quatro elementos de ponto flutuante.
  Instrução NEON: `FMIN Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vmovq_n_f32**: Inicializa um vetor de ponto flutuante com quatro elementos iguais ao valor fornecido. Caso o valor possa ser interpretado como imediato, ele será decodificado na instrução MOVI, caso contrário será decodificado na instrução DUP.
  Instrução NEON: `MOVI Vd.4s, #imm` ou `DUP Vd.4s, Vm.s[0]`
  Registrador: Vd.4s (vetor de destino), Vm.s[0] (vetor de origem)

- **vmovq_n_s32**: Inicializa um vetor de inteiros com quatro elementos iguais ao valor fornecido. Caso o valor possa ser interpretado como imediato, ele será decodificado na instrução MOVI, caso contrário será decodificado na instrução DUP.
  Instrução NEON: `MOVI Vd.4s, #imm` ou `DUP Vd.4s, Vm.s[0]`
  Registrador: Vd.4s (vetor de destino), Vm.s[0] (vetor de origem)

- **vst1q_f32**: Armazena os quatro elementos de um vetor de ponto flutuante em um array na memória.
  Instrução NEON: `STR {Vd.4s}, [Rn]`
  Registradores: Vd.4s (vetor de origem), Rn (endereço base de acesso a memória)

- **vst1q_s32**: Armazena os quatro elementos de um vetor de inteiros em um array na memória.
  Instrução NEON: `STR {Vd.4s}, [Rn]`
  Registradores: Vd.4s (vetor de origem), Rn (endereço base de acesso a memória)

- **vld1q_f32**: Carrega quatro elementos de ponto flutuante de um array da memória para um vetor.
  Instrução NEON: `LDR {Vd.4s}, [Rn]`
  Registradores: Vd.4s (vetor de destino), Rn (endereço base de acesso a memória)

- **vld1q_s32**: Carrega quatro elementos de inteiros de um array da memória para um vetor.
  Instrução NEON: `LDR {Vd.4s}, [Rn]`
  Registradores: Vd.4s (vetor de destino), Rn (endereço base de acesso a memória)

- **vzipq_f32**: Intercala os elementos de dois vetores de quatro elementos de ponto flutuante em dois novos vetores. No primeiro vetor de destino será armazenado os elementos ímpares do primeiro vetor de origem e depois os ímpares do segundo vetor. Já no segundo vetor de destino será armazenado os elementos pares do primeiro vetor de origem e depois os pares do segundo vetor.
  Instrução NEON: `ZIP1 Vd1.4s, Vn.4s, Vm.4s` e `ZIP2 Vd2.4s, Vn.4s, Vm.4s`
  Registradores: Vd1.4s (vetor de destino 1), Vd2.4s (vetor de destino 2), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vcombine_f32**: Combina dois vetores de dois elementos de ponto flutuante em um vetor de quatro elementos. Essa função é decodificada nas duas instruções a seguir.
  Instrução NEON: `DUP Vd.1d, Vn.d[0]` e `INS Vd.d[1], Vm.d[0]`
  Registradores: Vd (vetor de destino), Vn.d[0] (vetor de origem dos bits da parte inferior), Vm.d[0] (vetor de origem dos bits da parte superior)

- **vget_low_f32**: Extrai os dois elementos inferiores de um vetor de quatro elementos de ponto flutuante.
  Instrução NEON: `DUP Vd.1d, Vn.d[0]`
  Registradores: Vd.1d (vetor de destino, 64 bits), Vn.d[0] (vetor de origem, 128 bits)

- **vget_high_f32**: Extrai os dois elementos superiores de um vetor de quatro elementos de ponto flutuante.
  Instrução NEON: `DUP Vd.1d, Vn.d[1]`
  Registradores: Vd.1d (vetor de destino, 64 bits), Vn.d[1] (vetor de origem, 128 bits)

- **vgetq_lane_f32**: Extrai um elemento (representado por uma pista) de um vetor de quatro elementos de ponto flutuante.
  Instrução NEON: `DUP Sd, Vn.s[lane]`
  Registradores: Sd (registrador de destino, escalar), Vn.s[lane] (vetor de origem, escalar obtido pela pista especificada)

- **vbslq_s32**: Realiza uma operação ternária elemento a elemento, com um vetor sendo o seletor e o destino e os outros dois sendo os vetores de origem. Veja que há 3 argumentos e 1 resultado, então a priori seria necessário quatro registradores para essa função, contudo o compilador sempre adiciona instruções (ex: para copiar o seletor no vetor de destino com um par ldr e str), assim possibilitando a utilização do vetor de destino como seletor.
  Instrução NEON: `BSL Vd.16s, Vn.16s, Vm.16s`
  Registradores: Vd.16s (vetor de destino e seletor), Vn.16s (vetor de origem 1), Vm.16s (vetor de origem 2)

- **vceqq_s32**: Compara elemento a elemento a igualdade entre dois vetores de inteiros de 32 bits, retornando uma máscara de bits (-1 caso o resultado da comparação seja verdadeiro ou 0 caso contrário).
  Instrução NEON: `CMEQ Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

- **vcgtq_f32**: Compara elemento a elemento se os elementos de um vetor de ponto flutuante são maiores que os de outro vetor, retornando uma máscara de bits (-1 caso o resultado da comparação seja verdadeiro ou 0 caso contrário).
  Instrução NEON: `FCMGT Vd.4s, Vn.4s, Vm.4s`
  Registradores: Vd.4s (vetor de destino), Vn.4s (vetor de origem 1), Vm.4s (vetor de origem 2)

2.3 Funções Aceleradas com o ARM NEON

Nessa seção será debatido as funções dos arquivos *math_func.c* e *matrix.c* que foram acelerados com o uso da ARM NEON. Abaixo descreve-se as duas funções do *math_func.c*:

- float32x4_t relu(const float32x4_t A): retorna o resultado da operação ReLU (máx(x, 0)) em um float32x4_t;
- float32x4_t relu_derivate(const float32x4_t A): retorna o resultado da derivada da operação ReLU (retorna 1 caso x seja positivo, caso contrário retorna 0) em um float32x4_t.

Ambas as funções utilizam DLP ao operar sobre quatro dados paralelamente, para isso usam as funções do ARM NEON vmovq_n_f32 (para armazenar 0's e 1's em um float32x4_t), vcgtq_f32 (para decidir quais elementos do vetor A são maiores do que 0) e vbslq_f32 (operador ternário cujo seletor será o resultado do vcgtq_f32 com operandos sendo um float32x4_t de elementos nulos e o outro sendo a entrada A - caso seja o relu - ou um float32x4_t de elementos 1 - caso seja a derivada do relu).

As funções do arquivo *matrix.c* são descritas a seguir:

- void init_matrix(float32_t* A, const float32_t x, const int m, const int n): inicializa a matriz A (m por n) com o valor x;
- void init_matrix_random(float32_t* A, const int m, const int n): inicializa a matriz A (m por n) com valores aleatórios entre 0 e 1;
- void print_matrix(float32_t* A, int m, int n): imprime as entradas de uma matriz de ponto-flutuante m por n.
- void print_int_matrix(float32_t* A, int m, int n): imprime as entradas de uma matriz de inteiros m por n.
- void print_matrix_partial(float32_t* A, int m, int n, int k): imprime as k primeiras colunas de uma matriz de ponto-flutuante m por n.
- void sum_matrix(const float32_t* A, const float32_t* B, float32_t* C, const int m, const int n): realiza a soma das matrizes A e B (ambas m por n) e armazena o resultado em C (m por n);
- void sum_matrix_vector(const float32_t* A, const float32_t* B, float32_t* C, const int m, const int n): realiza a soma da matriz A (m por n) com um vetor B de n elementos, na qual o i-ésimo elemento do vetor será somado a todos os elementos da i-ésima coluna da matriz, o resultado da operação será armazenado na matriz C (m por n);
- void diff_matrix(const float32_t* A, const float32_t* B, float32_t* C, const int m, const int n): realiza a subtração das matrizes A e B (ambas m por n) e armazena o resultado em C (m por n);
- void sum_multiply_matrix_scalar_fast(float32_t* A, const float32_t* B, const float32_t x, const int m, const int n): multiplica os elementos da matriz B (m por n) por x e soma o resultado na matriz A, a função tem o sufixo *fast*, pois uma das entradas - no caso a matriz A - é modificada, em especial o resultado da operação é devolvido nela;
- void relu_matrix(const float32_t* A, float32_t* B, const int m, const int n): realiza a operação ReLU nos elementos da matriz A e armazena o resultado na matriz B (ambas m por n);
- void relu_derivate_matrix(const float32_t* A, float32_t* B, const int m, const int n): realiza a derivada da operação ReLU os elementos da matriz A e armazena o resultado na matriz B (ambas m por n);
- float32_t max_vector_fast(float32_t* A, const int n): retorna o maior elemento de um vetor de n posições, a função tem o sufixo *fast*, pois o vetor A é modificado;

- float32_t min_vector_fast(float32_t* A, const int n): retorna o menor elemento de um vetor de n posições, a função tem o sufixo *fast*, pois o vetor A é modificado;
- void minmax_matrix(const float32_t* A, float32_t* B, const int m, const int n): realiza a operação minmax em cada linha da matriz A (m por n), isto é, cada elemento x é substituído por $\frac{(x-min)}{(max-min)}$, onde min e max são respectivamente o menor e o maior elemento da linha que x pertence e armazena o resultado dessa operação na matriz B (m por n);
- void softmax_matrix(const float32_t* A, float32_t* B, const int m, const int n): realiza a operação softmax em cada linha da matriz A (m por n), isto é, cada elemento x_{ij} é substituído por $\frac{(e^{x_{ij}})}{(\sum_{j=1}^{j=n} e^{x_{ij}})}$, e armazena o resultado dessa operação na matriz B (m por n);
- void copy_vector(const float32_t* A, float32_t* B, const int n): copia um vetor A de n elementos em um vetor B;
- void transpose_matrix(const float32_t* A, const int m, const int n): retorna a transposta da matriz A de dimensões m por n;
- void one_hot_matrix(const int* A, float32_t* B, const int m, const int n): retorna a matriz *one hot* do vetor A de dimensão m e com elementos entre 0 e (n-1) na matriz B (m por n), isto é, na i-ésima linha da matriz B a coluna A[i] terá valor 1, enquanto as demais terão valor 0;
- void multiply_matrix_scalar(const float32_t* A, float32_t* B, const float32_t x, const int m, const int n): retorna o resultado da multiplicação de uma matriz A (m por n) por um escalar x na matriz B (m por n);
- void multiply_matrix_matrix(const float32_t* A, const float32_t* B, float32_t* C, const int m, const int l, const int n): retorna o resultado da multiplicação matricial (linha por linha, invés de linha por coluna) de uma matriz A (m por l) por outra B (n por l) na matriz C (m por n);
- void multiply_matrix_hadamard(const float32_t* A, const float32_t* B, float32_t* C, const int m, const int n): retorna o resultado da multiplicação hadamard entre duas matrizes A e B na matriz C (todas m por n), na qual o elemento ij da matriz resultante será o produto dos elementos A[i][j] e B[i][j];
- void compare_vector(const int* A, const int* B, int* C, const int n): retorna o resultado das comparações entre dois vetores A e B no vetor C de tamanho n, na qual cada elemento do vetor C terá valor -1 se os respectivos elementos dos vetores A e B forem iguais, e 0 caso contrário;
- void matrix_redux_float(const float32_t* A, float32_t* B, const int m, const int n): escreve no i-ésimo elemento de B a soma dos elementos da linha A[i] da matriz;
- void matrix_redux_int(const int* A, int* B, const int n, const int m): escreve no i-ésimo elemento de B a soma dos elementos da linha A[i] da matriz de inteiros;

As funções init_matrix, sum_matrix, sum_matrix_vector, diff_matrix, sum_multiply_matrix_scalar_fast, relu_matrix,
relu_derivate_matrix, copy_vector, multiply_matrix_scalar, multiply_matrix_hadamard e compare_vector exploram o DLP da mesma forma, carregando seus operandos vetoriais em um registrador32x4 utilizando a função vld1q_f32, realizando suas operações em registradores vetoriais e armazenando os resultados na memória utilizando a função vst1q_f32.

As funções max_vector_fast e min_vector_fast também utilizam a mesma estratégia de usar as funções NEON para carregar, computar e armazenar seus dados. Contudo, tratam-se de funções recursivas, pois a cada iteração determinam um grupo seleto de elementos que pode estar armazenado o maior/menor elemento do vetor original, sendo que para armazenar esse grupo seleto utiliza-se o vetor original, por isso são *fast*. A cada iteração separa-se o vetor em blocos de 8 elementos consecutivos que são subdivididos em fatias de 4 elementos, na qual as fatias de um mesmo bloco serão comparadas entre si - elemento a elemento - para determinar o maior (menor) elemento entre os dois, em seguida o resultado das 4 operações será armazenado em um float32x4_t e posteriormente guardado no vetor original.

A função minmax_matrix também utiliza a mesma estratégia de usar as funções NEON para carregar, computar e armazenar seus dados. Primeiramente, deve-se computar o mínimo e o máximo de cada linha da matriz (utilizando as funções max_vector_fast e min_vector_fast) e após isso será aplicado a operação

do minmax a cada elemento da determinada linha da matriz (sendo essa operação paralelizável com as funções vld1q_f32, vmlaq_n_f32 e vst1q_f32).

Para explorar o DLP, a função transpose_matrix separa a matriz em pequenas matrizes 4 por 4 que passam pela operação de transposição ao mesmo tempo, para tal utiliza-se diversas funções de manipulação de elementos de float32x4_t (vcombine_f32, vget_low_f32, vget_high_f32, vzipq_f32).

Para usufruir da DLP, as funções multiply_matrix_matrix, matrix_redux_float e matrix_redux_int realizam as suas operações em registradores vetoriais, todavia para obter um elemento da matriz (vetor) resultante será necessário somar os resultados parciais contidos em cada um dos 4 elementos do float32x4_t (por meio da função vgetq_lane_f32) e então armazenar esse resultado final na memória (sem o auxílio do ARM NEON).

Nas funções init_matrix_random, print_matrix, print_int_matrix e print_matrix_partial não é possível explorar o DLP, desse modo elas não utilizam as funções do ARM NEON. Ademais, a função one_hot_matrix utiliza o DLP apenas ao inicializar suas entradas com 0, mas não é capaz de explorá-lo durante a decodificação *one-hot*. Já no caso da função softmax_matrix é possível realizar o DLP, contudo devido a lógica do cálculo conter diversos desvios (if's) tornou-se impraticável de adaptar esse cálculo para o ARM NEON, visto que o ARM NEON não suporta predicados.

Por fim, vale ressaltar que por mais que quase todas as funções descritas acima utilizem o ARM NEON, nem todas terão um ganho significativo, devido a *overheads* da paralelização e a limitações desse processador vetorial (principalmente para acessar elementos por colunas). Em especial, ressalta-se que as funções minmax_matrix e transpose_matrix devem ter os menores ganhos de desempenho, contudo ainda sim - por motivos didáticos - optou-se por utilizar a vetorização nessas funções.

3 Rede Neural Convolucional

O reconhecimento de imagens neste projeto foi realizado a partir de uma Rede Neural Convolucional, ou "CNN". Elas podem ser entendidas como grafos conexos entre neurônios de entrada, intermediários e de saída.

3.1 MNIST

O MNIST (Modified National Institute of Standards and Technology) é um banco de dados de números de 0 a 9 manuscritos e com as *labels* correspondentes, com 60000 imagens para treinamento e 10000 para teste. Esse conjunto é amplamente utilizado para treinamento de redes neurais de identificação de dígitos.

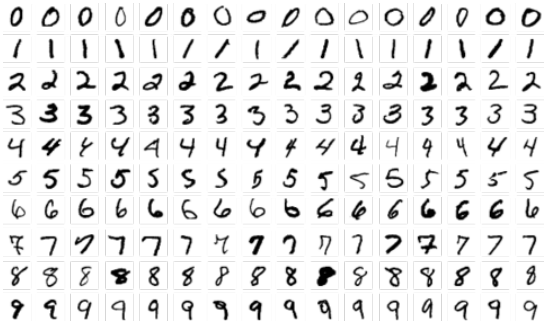

Figura 1: Exemplo de algarismos do MNIST

As imagens do banco possuem resolução de 28x28 pixels (784 no total), em que cada pixel será representado por um único *uint8* (imagens monocromáticas). Desse modo, será utilizado 784 neurônios no *input layer* (1 por pixel) da CNN.

3.2 Estrutura da Rede Neural

A rede neural convolucional adotada para o projeto possui 784 neurônios de entrada (um para cada pixel da imagem) e 10 neurônios de saída (um para cada algarismo de 0-9). Por simplicidade, optou-se por não incluir camadas intermediárias (*hidden layers*) na rede neural. Assim, a CNN pode ser entendida como um grafo totalmente conexo entre os neurônios das camadas de entrada e saída:

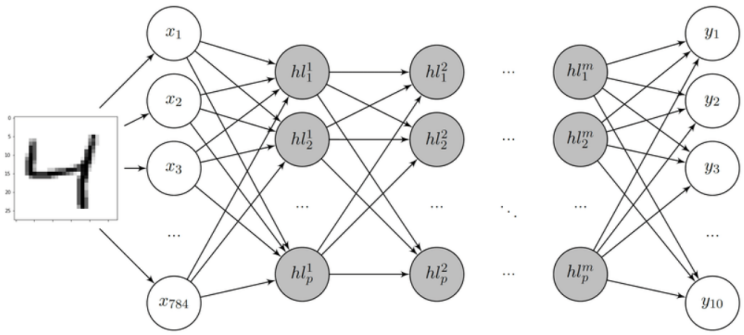

Figura 2: Rede neural convolucional genérica para MNIST

3.3 Operações na rede neural

O *aprendizado* na rede neural consiste em ajustar os parâmetros de weights e biases em cada vértice, de modo a, para certo input de imagem, determinar a saída correta com alta acurácia. Os valores presentes nos nós são multiplicados pelos pesos (*weights*) de cada vértice, e então somados a um viés (*bias*). Esse valor é então passado como argumento de uma função de ativação (no caso do projeto, foi escolhida a função *softmax*). Esse processo é determinado *forward propagation*. Para cada neurônio de saída,

tem-se a probabilidade de cada dígito ser a resposta correta, que é escolhida como o neurônio de maior probabilidade.

O algoritmo de aprendizado na CNN é realizado pelas operações de *forward propagation*, em que os valores de inputs são propagados para a saída, e *backward propagation*, em que os valores são propagados de volta pela rede neural, e os parâmetros atualizados, de modo a diminuir o erro nas inferências.

3.3.1 Forward Propagation

A operação realizada no vértice entre o nó de input i e o de output j é:

$$z_j = x_i \cdot w_{ij} + b_{ij}$$

Considera-se que m é o número de entradas sendo usadas em lote para treinar a rede neural. Em notação matricial, tem-se as matrizes de entrada $X_{784\times m}$, de peso $W_{10\times 784}$, de bias $B_{10\times 1}$ e de saída $Z_{m\times 10}$:

$$Z = X \times W + B$$

Como a saída deve ser uma distribuição de probabilidades, aplica-se a função *softmax* no vetor de saída, sendo, para cada elemento da matriz $A_{m\times 10}$:

$$A_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Com isso, obtém-se os valores de saída. Inicialmente, espera-se que a função de distribuição de probabilidades esteja desregulada, isto é, com altas taxas de erro/baixa acurácia. O processo de *aprendizado* refere-se a diminuir estes erros a partir da propagação.

O *erro* ou *custo* (dZ) da CNN é definido como a distância dos resultados inferidos aos valores reais da saída (Y_{real}):

$$dZ = A - Y_{real}$$

3.3.2 Backward Propagation e Descida de Gradiente

Backward propagation é o processo de calcular os gradientes da função de custo em relação aos pesos e biases, permitindo que a rede ajuste seus parâmetros.

Matematicamente, um gradiente é um vetor que aponta para a direção de maior aumento em uma função. Neste caso, deseja-se minimizar o erro e, portanto, decrementa-se os parâmetros da rede por um valor proporcional ao gradiente. O gradiente dos pesos pode ser calculado como:

$$dW = \frac{1}{m} dZ X^T$$

De maneira análoga, o gradiente dos biases podem ser definidos como:

$$dB = \frac{1}{m} \sum dZ$$

Os parâmetros da rede neural (weights e biases) são atualizados pela *descida de gradiente*, subtraindo deles o gradiente multiplicado por uma constante α, denominada "taxa de aprendizado"(*learning rate*:

$$W := W - \alpha \cdot dW$$

$$B := B - \alpha \cdot dB$$

Esse processo de *forward* e *backward propagation* é repetido até que a acurácia seja suficientemente alta ou até que um certo número de iterações desse processo seja atingido.

3.4 Implementação

Para a implementação da rede neural, foram aplicadas as funções descritas na função 2.3, de forma a executar o fluxo de informações descritos em 3.3. No entanto, antes é necessário definir o formato dos dados que serão tratados ao longo da rede neural, para isso, são definidos 4 *structs*, com o conteúdo de cada um descrito abaixo.

- neural_network_t_:
  - b1: vetor com *biases* associados à propagação entre as camadas da rede neural
  - W1: matrix de pesos multiplicativos asssociados à propagação entre as camadas da rede neural;
- neural_network_layers_t_:
  - Z1: Valores propagados da primeira para a segunda camada, antes da aplicação da função de ativação;
  - A1: Resultado da aplicação da função de ativação em Z1, nesse caso corresponde aos valores de saída da *output layer*;
- neural_network_gradient_t_:
  - db1: Vetor gradiente que deve ser multiplicado pela *learning rate* e somado aos vetor de *bias* durante a atualização de parâmetros;
  - dW1: Matriz que cumpre função semelhante à db1 para a matriz de pesos, ao invés do vetor de *bias*;
- batch_t_:
  - X: Matriz de entradas, cada valor representa um pixel em uma das imagens de entrada, normalizado para um ponto flutuante entre 0 e 1 (divisão simples do valor entre 0 e 255 por 255);
  - Y: Vetor de inteiros que representa a classificação correta das imagens utilizadas durante o treinamento.

A partir da definição dessas estruturas, são construídas duas funções essenciais para o funcionamento da rede neural, sendo elas *forward_propagation* e *backward_propagation*, descritas em seções abaixo. Além disso, também são essenciais as funções *train*, que executa o treinamento da rede neural, e *inference*, que obtém as predições para um determinado conjunto de dados.

3.4.1 *forward_propagation*

Essa função cumpre o papel de executar o fluxo de inputs para outputs da rede neural, propagando os dados de camada a camada. Como a rede neural implementada não possui camadas escondidas, essa função se torna extremamente simples, e pode ser resumida em três operações com vetores, sendo elas:

- A multiplicação da matriz de inputs pela matriz de pesos W1 da rede neural;
- A soma do resultado da operação anterior com o vetor de *bias* b1;
- A aplicação da operação de softmax no resultado anterior.

O código em C que implementa essa funcionalidade, utilizando as funções de operações matriciais com aceleração de hardware descritas na seção 2.3 está disponível abaixo.

```
1  void forward_propagation(const float32_t* X, neural_network_t* cnn,
2                           neural_network_layers_t* layers, int m) {
3    float32_t* M;
4    // input layer
5    M = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * m);
6    multiply_matrix_matrix(X, cnn->W1, M, m, INPUT_LAYER_SIZE, OUTPUT_LAYER_SIZE);
7    sum_matrix_vector(M, cnn->b1, layers->Z1, m, OUTPUT_LAYER_SIZE);
8    softmax_matrix(layers->Z1, layers->A1, m, OUTPUT_LAYER_SIZE);
9    free(M);
10   return;
11 }
```

3.4.2 *backward_propagation*

Como o nome indica, essa função realiza o fluxo contrário na rede neural, partindo das diferenças entre os outputs obtidos e o que era esperado pelas *labels* e populando os vetores db1 e dW1 para que possa ser realizada a atualização dos pesos e *biases* para a próxima iteração de treinamento. Para isso, ela deve receber como input os outputs da função *forward_propagation*, pois se baseia na comparação da matriz de saída A1 com a matriz gerada pela operação *one_hot* no vetor com os *labels* esperados.

A partir desses inputs, novamente é executada uma sequência de operações matriciais simples que resultam no vetor db1 e na matriz dW1:

- Aplicação da diferença entre a saída A1 e a saída esperada, com o resultado associado à variável temporária dZ;
- Multiplicação da transposta de dZ com a matriz de inputs X e registro do resultado na matriz dW1;
- Execução da operação *redux* (descrita na seção 2.3) em dZ e registro no vetor db1.

Ao longo dessa sequência de operações, além das funções de multiplicação, subtração e *redux*, também é necessário realizar duas chamadas à função de transposição de matrizes, pois, para melhor explorar o paralelismo de dados, a função de multiplicação é na verdade a multiplicação do primeiro argumento pela transposta do segundo, ou seja, devemos passar como inputs da multiplicação a transposta de dZ e a transposta de X. A implementação em C dessa função está disponível abaixo.

```
1  void backward_propagation(const float32_t* X, const float32_t* Y,
2                            neural_network_t* cnn,
3                            neural_network_gradient_t* dcnn,
4                            neural_network_layers_t* layers, int m) {
5    float32_t *dZ, *dZT, *XT;
6    // Calculo dos dZ's
7    dZ = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * m);
8    diff_matrix(layers->A1, Y, dZ, m, OUTPUT_LAYER_SIZE);
9    //    Calculo das saidas
10   //    dW1
11   dZT = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * m);
12   transpose_matrix(dZ, dZT, m, OUTPUT_LAYER_SIZE);
13   XT = malloc(sizeof(float32_t) * INPUT_LAYER_SIZE * m);
14   transpose_matrix(X, XT, m, INPUT_LAYER_SIZE);
15   multiply_matrix_matrix(dZT, XT, dcnn->dW1, OUTPUT_LAYER_SIZE, m,
16                          INPUT_LAYER_SIZE);
17   //  dB1
18   matrix_redux_float(dZT, dcnn->db1, OUTPUT_LAYER_SIZE, m);
19   free(dZT);
20   free(XT);
21   free(dZ);
22   return;
23 }
```

3.4.3 *train*

Por ser a função responsável por executar o treinamento da rede neural, é necessário que a função *train* realize a inicialização de alguns parâmetros antes de iniciar as iterações pela rede neural. Primeiro, é necessário inicializar os parâmetros b1 e W1 com valores aleatórios entre 0 e 1, evitando efeitos indesejados no comportamento da rede devido ao uso de valores que seguem padrões previsíveis. Segundo, as imagens do conjunto de treinamento são dividas em grupos menores, com 100 imagens cada; mais um grupo menor caso o número de imagens não seja múltiplo de 100. Cada um desses grupos, chamados de *batch* é utilizado nas iterações na forma *round robin*, evitando que o treinamento seja feito sempre sobre as mesmas imagens como o intuito de combater o efeito de *overfitting*, em que a rede neural está "viciada"nas imagens do conjunto de treinamento e não consegue uma boa performance para outros conjuntos de dados.

Com isso, para cada iteração de treinamento, a função deve executar os seguintes passos:

- Gerar a Matriz *One Hot* correspondente às *labels* da *batch* utilizada para a iteração;
- Gerar os outputs da rede neural através da função *forward_propagation*;
- Gerar os parâmetro db1 e dW1 para atualização dos pesos e *biases* utilizando a função de propagação descrita em 3.4.2;
- Atualização dos parâmetros da rede neural;
- Obtenção das predições e cálculo da acurácia.

Vale lembrar que, conforme descrito na seção 3.3, a atualização de parâmetros segue o modelo de *Gradient Descent*, logo os novos parâmetros são calculados pela soma dos parâmetros anteriores subtraindo o resultado da multiplicação da matriz dW1 (ou o vetor db1 no caso dos *biases*, pelo escalar $\frac{\alpha}{m}$, em que α é a *learning rate* e m o número de elementos no batch. Outro detalhe é que a obtenção das predições e da acurácia seguem funções extremamente simples, de forma que a primeira apenas encontra o índice associado ao elemento de maior probabilidade para cada imagem da *batch*, enquanto a segunda gera um vetor de comparações entre as predições e os outputs esperados (em que 0 indica um erro na predição e 1 um acerto) e executa uma operação de *redux* no mesmo, gerando assim o número de acertos da rede neural para essa leva de dados.

Abaixo, econtra-se a implementação na linguagem C para o treinamento da rede neural.

```
float32_t train(float32_t* X, int* Y, neural_network_t* cnn, const int set_size,
                const int iterations) {
  float32_t* Y_one_hot;
  int accuracy, m, number_of_batches, *predictions;
  neural_network_gradient_t dcnn;
  neural_network_layers_t layers;
  batch_t batch;
  predictions = malloc(sizeof(int) * BATCH_SIZE);
  layers.Z1 = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * BATCH_SIZE);
  layers.A1 = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * BATCH_SIZE);
  Y_one_hot = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * BATCH_SIZE);
  number_of_batches = set_size / BATCH_SIZE;
  random_params(cnn);
  for (int i = 0; i < iterations; i++) {
    m = init_batch(X, Y, &batch, i % number_of_batches, set_size);
    one_hot_matrix(batch.Y, Y_one_hot, m, OUTPUT_LAYER_SIZE);
    forward_propagation(batch.X, cnn, &layers, m);
    backward_propagation(batch.X, Y_one_hot, cnn, &dcnn, &layers, m);
    parameter_update(cnn, &dcnn, m);
    get_predictions(layers.A1, predictions, m);
    accuracy = get_accuracy(predictions, batch.Y, m);
    if (i % 20 == 0) {
      printf("Iteration: %d\n", i);
      printf("Accuracy: %f\n", ((float32_t)accuracy) / ((float32_t)m));
    }
  }
  free(layers.Z1);
  free(layers.A1);
  free(Y_one_hot);
  free(predictions);
  return ((float32_t)accuracy) / ((float32_t)m);
}
```

3.4.4 inference

O último componente operacional da rede neural é a função que realiza inferências para um determinado conjunto de imagens. As imagens tratadas por essa operação não possuem *labels* associadas, afinal o objetivo da função e determinar que dígitos estão sendo retratados, logo não é feito o cálculo da acurácia diretamente.

Sua implementação é extremamente simples, consistindo em realizar a propagação pela rede neural (*forward_propagation*), e então utilizar as distribuições de probabilidade geradas em A1 para escolher as predições para cada uma das imagens, utilizando a mesma função descrita em 3.4.3. O código que implementa essa funcionalidade está disponível a seguir.

```
int* inference(const float32_t* X, neural_network_t* cnn, const int set_size) {
  neural_network_layers_t layers;
  int* predictions;
  layers.Z1 = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * set_size);
  layers.A1 = malloc(sizeof(float32_t) * OUTPUT_LAYER_SIZE * set_size);
  forward_propagation(X, cnn, &layers, set_size);
  predictions = malloc(sizeof(int) * set_size);
  get_predictions(layers.A1, predictions, set_size);
  free(layers.Z1);
  free(layers.A1);
  return predictions;
}
```

4 Aplicação Cliente-Servidor

Com o objetivo de permitir o uso remoto da Rede Neural foi desenvolvido uma Aplicação Cliente-Servidor, onde o Raspberry Pi será o Servidor e proverá serviços de inferência e de treinamento, enquanto isso o computador do usuário exercerá o papel de cliente, na qual terá uma CLI (Command Line Interface) para facilitar o uso da aplicação.

Inicialmente, a comunicação entre o cliente e o servidor foi desenvolvida em cima do protocolo UART (com velocidade máxima de transmissão de 115200bps). Contudo, o conjunto de imagens de treinamento do MNIST tem 47 MB, dessa forma seria necessário 3264s (aproximadamente 55 minutos) para a transmissão desses dados. Assim, para reduzir o tempo de transmissão optou-se por utilizar o protocolo TCP - como a vazão de um link ethernet normalmente está entre 10Mbps - 100Mbps, então seriam gastos no máximo 38 segundos com a transmissão.

4.1 Socket

Para abstrair o uso do TCP, tomou-se a decisão de utilizar a API Socket (disponibilizada na biblioteca *<sys/socket.h>*), cuja responsabilidade é de abstrair a utilização do TCP. Sendo que, para facilitar ainda mais a utilização dessa API na aplicação, desenvolveu-se uma biblioteca denominada de *socket_wrapper* que encapsula a utilização da API Socket. Esta biblioteca tem as seguintes funções:

- int socket_connect(char* ip, int port): retorna o número do socket que conecta esse cliente a um servidor com IPv4 e porta definidos nos argumentos dessa função;
- int socket_server_init(int port, struct sockaddr_in* address): inicializa um servidor na porta desejada, retorna o socket desse servidor, além de um descritor do seu endereço socket (*sockaddr_in*);
- int socket_listen(int port, int handler, struct sockaddr_in* address): coloca o servidor com número socket (handler) e descritor de endereçamento socket (address) em modo de escuta em uma determinada porta (port), caso uma conexão com um cliente seja bem sucedida retorna o número do socket dessa conexão, caso contrário retorna -1;
- void socket_read(int sock, char* buf, ssize_t size): lê size bytes em uma conexão socket (indexada por sock) e insere esses valores em um buffer (buf);
- void socket_write(int sock, char* buf, ssize_t size): escreve size bytes de um buffer (buf) em uma conexão socket (indexada por sock);
- void socket_close(int sock): encerra uma conexão socket (indexada por sock);

4.2 Máquina de Estados do Servidor

Com o intuito de simplificar o protocolo de comunicação entre o cliente e o servidor foi definido uma máquina de estados no servidor (assim tirando a necessidade do uso de cabeçalhos nas mensagens) que pode ser vista na figura 3. Vale ressaltar que essa máquina foi desenvolvida apenas para tratar da comunicação com o cliente, desse modo a chamada das funções de inferência/treinamento será feito por um módulo a parte descrito na seção 3.4. O arquivo *server_fsm.c* (que implementa essa máquina) tem duas funções:

- int next_state(int state, int mode): retorna o próximo estado a partir do estado atual (state), o argumento mode determina qual o modo solicitado pelo usuário (válido apenas no estado IDLE);
- int action(int state, int sock, int* result, int size, float32_t** data): realiza uma ação (ligada a comunicação com o cliente) de acordo com o estado do servidor (o funcionamento dessa função será melohor descrito abaixo).

O comportamento de cada estado dessa máquina, isto é o comportamento da função action, está descrito a seguir:

- INIT: Inicializa o servidor, retorna o número do Socket do servidor;
- LISTEN: Servidor aguarda por um cliente, caso a conexão seja bem sucedida retorna o número do Socket dessa conexão, caso contrário retorna -1;

- IDLE: Obtêm e retorna o comando enviado pelo cliente que determinará o modo de operação do servidor, cujos valores possíveis são TRAIN (treinamento), INFER (inferência), CLOSE (encerrar conexão), NONE (comando inválido/continuar nesse estado);
- WAIT_TRAIN_DATA: Obtêm o tamanho do conjunto de dados (retorno da função) e as imagens (repassadas pela variável data) enviados pelo cliente;
- WAIT_TRAIN_LABELS: Obtêm os rótulos enviados pelo cliente e repassa eles via variável result (a memória dinâmica dessa variável já foi previamente alocada de acordo com o tamanho do conjunto de dados);
- WAIT_TRAIN_ITERATIONS: Obtêm e retorna o número de iterações que devem ser realizadas no treinamento;
- SEND_TRAINING_RESULT: Envia ao cliente a acurácia do treinamento (apontada por result), retorna 0;
- WAIT_INFERENCE_DATA: Obtêm o tamanho do conjunto de dados (retorno da função) e as imagens (repassadas pela variável data) enviados pelo cliente;
- SEND_INFERENCE_RESULT: Envia ao cliente as predições realizadas a partir da inferência (dadas pelo vetor result), retorna 0;
- END_CONNECTION: Encerra a conexão com o cliente, retorna 0;

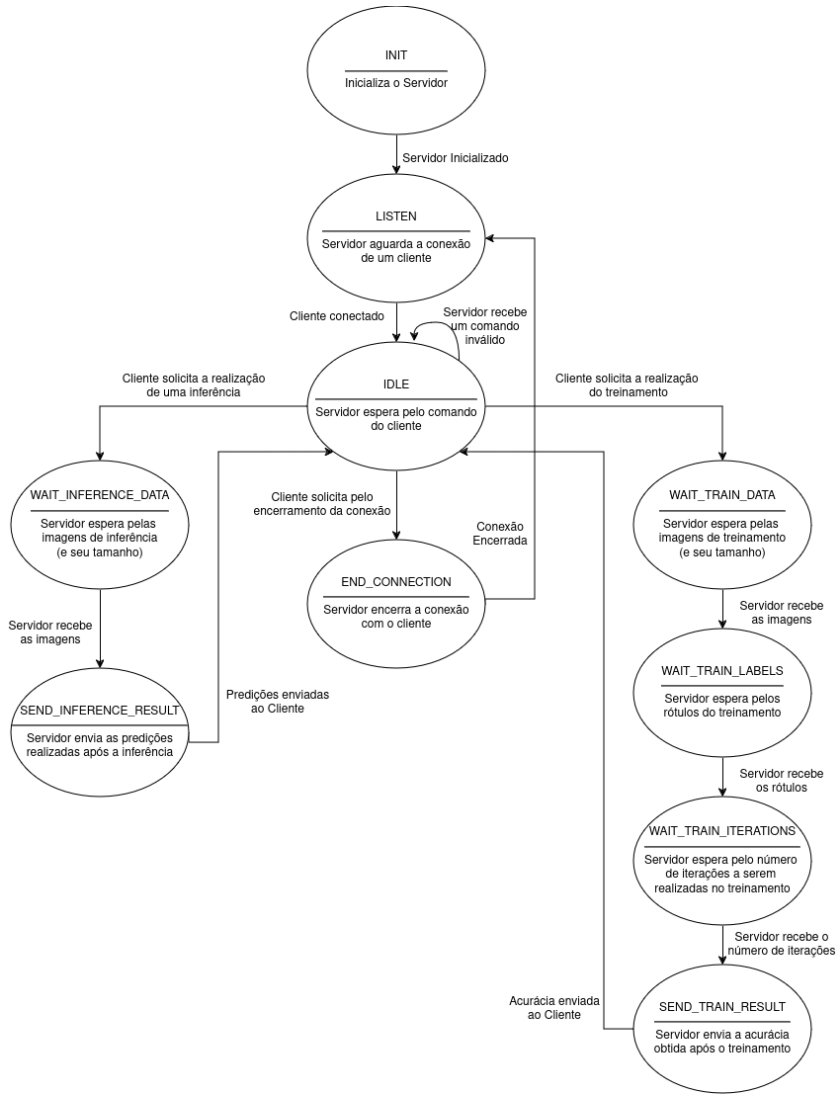

Figura 3: Máquina de Estados do Servidor

4.3 Aplicação Cliente: Interface em Linha de Comando

Nessa seção será discutida o cliente dessa aplicação (executado no computador do usuário), na qual será responsável por interagir com o servidor (via Socket) e com o usuário (via CLI). Em especial, esse cliente também será capaz de interpretar arquivos idx1-ubyte e idx3-ubyte para obter as imagens e rótulos dos conjuntos de dados do MNIST e enviá-los ao servidor. Vale ressaltar que a conexão com o servidor é feita automaticamente ao executar o cliente.

Essa Interface em Linha de Comando oferece 8 serviços:

- help: exibe a lista de comandos válidos e suas funcionalidades;
- connect <server_ip> <server_port> : conecta o cliente ao servidor com IPv4 e porta especificados;
- train_default: treina a rede neural com o data set padrão do MNIST;
- train_custom <image_path> <label_path>: Treina a rede neural com as imagens fornecidas em <image_path> e com os rótulos fornecidos em <label_path>;
- test_default: Testa a rede neural com o data set padrão do MNIST e verifica o resultado da inferência com os rótulos desse data set;
- test_custom <image_path> <label_path>: Testa a rede neural com as imagens fornecidas em <image_path> e verifica o resultado da inferência com os rótulos fornecidos em <label_path>;
- inference <image_path>: Realiza a inferência com as imagens fornecidas em <image_path> e exibe as predições na linha de comando;
- close: encerra o programa cliente e a conexão com o servidor;

O comando connect apenas realiza uma conexão via Socket com o servidor especificado pelo endereço IPv4 e pela porta dados como argumento desse comando.

Os comandos train_default e train_custom utilizam a função train() para realizar o treinamento, isto é, o cliente envia o caracter TRAIN para o servidor (iniciando o modo de treinamento), depois enviam o tamanho do conjunto de dados, as imagens, os rótulos e o número de iterações (seguindo os padrões definidos em 4.2). Após enviar todos esses dados, ele espera pela resposta do servidor contendo a acurácia do treinamento e então imprime essa informação na tela.

Já os comandos test_default e test_custom utilizam a função inference() para testar a inferência, isto é, o cliente envia o caracter INFER para o servidor (iniciando o modo de inferência), depois enviam o tamanho do conjunto de dados e as imagens (seguindo os padrões definidos em 4.2). Após enviar todos esses dados, ele espera pela resposta do servidor contendo as predições, então compara as predições do servidor com os rótulos esperados e imprime a acurácia obtida.

Assim como os comandos test_default e test_custom, o comando inference também utiliza a função inference(), contudo ele tem como propósito inferir o rótulo de uma nova imagem, ao invés de verificar a acurácia da rede neural. Desse modo, ele também realiza outro tratamento na resposta do servidor, na qual ele simplesmente imprime as predições.

Já o comando close utiliza a função close_connection() para realizar o encerramento da conexão, isto é, envia o caracter CLOSE para o servidor, com isso fechando a comunicação e encerrando a aplicação cliente.

4.3.1 Leitura de Arquivos MNIST

Ademais, o cliente também é capaz de realizar a leitura e interpretação dos arquivos idx1-ubyte e idx3-ubyte (padrão utilizado pela MNIST) que armazenam respectivamente as imagens dos dígitos e seus rótulos. Para encapsular a lógica da leitura foi desenvolvido o arquivo *mnist_file.c* com duas funções:

- int get_mnist_images(const char* image_path, char** images): verifica o cabeçalho do arquivo - de acordo com o padrão definido pela MNIST - (primeiros 32 bits valendo 0x00000803, seguidos de 32 bits informando o número de imagens no arquivo e de mais dois pares de 32 bits com ambos valendo 28 (largura e altura das imagens)), armazena as imagens do arquivo em images e retorna o número de imagens contida no arquivo. Vale ressaltar que esses arquivos estão em big-endian, logo caso o computador utilize little-endian será necessário converter os bytes lidos para little-endian antes de realizar as checagens.

- int get_mnist_labels(const char* label_path, char** labels): verifica o cabeçalho do arquivo - de acordo com o padrão definido pela MNIST - (primeiros 32 bits valendo 0x00000801, seguidos de 32 bits informando o número de rótulos no arquivo), armazena os rótulos do arquivo em labels e retorna o número de imagens contida no arquivo. Vale ressaltar que esses arquivos estão em big-endian, logo caso o computador utilize little-endian será necessário converter os bytes lidos para little-endian antes de realizar as checagens.

5 Função Main

A função main do Raspberry Pi é responsável por coordenar a interação entre a rede neural (neural_network) e a máquina de estados do servidor (server_fsm). Para isso, ele armazena diversas informações intermediárias, como socket do servidor e da conexão com o cliente, estado do servidor, os parâmetros da rede neural e as imagens enviadas pelo cliente.

Devido a presença da máquina de estados do server_fsm, tornou-se necessário que a função main também tivesse uma lógica de estados, na qual ela inicialmente chama a função action() do server_fsm, depois disso armazena as saídas dessa função e prepara os argumentos (isto é realiza (des)alocação de memória e repasse de valores) para a próxima chamada dessa função e da função next_state() e por fim chama a função next_state() do server_fsm para atualizar o estado. A lógica realizada pela função principal em cada estado pode ser visto a seguir:

- INIT: Armazena o número Socket do Servidor;
- LISTEN: Armazena o número Socket da conexão com o Cliente;
- IDLE: Armazena o modo de operação solicitado pelo Cliente;
- WAIT_TRAIN_DATA: Armazena as imagens de treino e o número de imagens;
- WAIT_TRAIN_LABELS: Armazena os rótulos das imagens de treino;
- WAIT_TRAIN_ITERATIONS: Armazena o número de iterações do treinamento, realiza o treinamento e repassa a acurácia do treino para o server_fsm;
- SEND_TRAINING_RESULT: Nada é feito;
- WAIT_INFERENCE_DATA: Armazena as imagens de inferência e o número de imagens;
- SEND_INFERENCE_RESULT: Nada é feito;
- END_CONNECTION: Nada é feito;

6 Resultados

Com o intuito de averiguar o ganho de performance obtido pelo uso do processador vetorial, realizamos a medição do tempo de execução da função de treinamento da nossa rede neural com a placa Raspberry Pi 3B+ sem aceleração de hardware e com a aceleração de hardware. A tabela comparativa de tempos pode ser vista na tabela 1.

Ambiente	Tempo Aproximado de Execução
Raspberry Pi 3B+ Sem Aceleração de Hardware	15 segundos
Raspberry Pi 3B+ Com Aceleração de Hardware	3.7 segundos

Tabela 1: Comparativo do Tempo de Execução em Diferentes Ambientes

O ganho observado de aproximadamente 4 vezes no tempo de execução se deve a uma combinação de todas as características discutidas na seção 2.1, sendo que se destacam a capacidade de executar múltiplas operações de forma paralela em uma única instrução em linguagem de montagem, e o uso de pipelines em cada uma das unidades funcionais, o que mascara a latência de cada uma delas. Além disso, a existência de instruções que realizam mais de uma operação aritmética por dado, a exemplo da instrução vfma, que realiza uma soma e uma multiplicação, reduzindo ainda mais o custo computacional das operações vetoriais quando comparado a um processador escalar.

7 Conclusão

Neste trabalho, discutimos as principais características do processador NEON, além de apresentar os principais componentes de uma rede neural construída utilizando suas características de aceleração de hardware. Através disso, pudemos nos familiarizar com o funcionamento de processadores vetoriais, além de melhor compreender o paradigma de programação que necessariamente os acompanha.

Adicionalmente, também conseguimos uma visão compreensiva do funcionamento de uma rede neural, algo de interesse dos integrantes do grupo mas um tópico com o qual não possuíamos experiência extensa. Durante o processo, tivemos também a oportunidade de compreender os desafios associados à implementação de uma rede neural utilizando uma linguagem de baixo nível como C, adquirindo conhecimentos de depuração e desenvolvimento que se estendem além desse domínio do conhecimento e podem ser aplicados na criação de diversas aplicações complexas.

Por fim, foi possível perceber a eficiência que o uso de um processador vetorial adiciona a uma aplicação, caso aplicada de forma correta. Isso justifica a necessidade de conhecimento da plataforma em que cada aplicação será implantada no momento de sua codificação, pois pode gerar grandes ganhos de desempenho, além de ilustrar o grande potencial presente nos processadores de arquitetura ARM.

8 Bibliografia

- **Building a neural network FROM SCRATCH (no Tensorflow/Pytorch, just numpy & math)**. Disponível em: ¡https://www.youtube.com/watch?v=w8yWXqWQYmU¿.
- **Documentation – Arm Developer**. Disponível em: ¡https://developer.arm.com/documentation/101028/0012/13--Advanced-SIMD--Neon--intrinsics¿. Acesso em: 16 ago. 2024.
- **Documentation – Arm Developer**. Disponível em: ¡https://developer.arm.com/documentation/den0018/a/¿. Acesso em: 16 ago. 2024.
- **Intrinsics – Arm Developer**. Disponível em: ¡https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=¿. Acesso em: 16 ago. 2024.
- THENIFTY. **GitHub - thenifty/neon-guide: Makes ARM NEON documentation accessible (with examples)**. Disponível em: ¡https://github.com/thenifty/neon-guide¿. Acesso em: 16 ago. 2024.
- PTH5804. **GitHub - pth5804/MatTrans_Mul_NEON_PQC: This source code is matrix transpose, matrix multiplication and vector addition by using ARM NEON intrinsic for lattice-based cryptography based on LWE problem**. Disponível em: ¡https://github.com/pth5804/MatTrans_Mul_NEON_PQC/tree/master¿. Acesso em: 16 ago. 2024.
- WWSALMON. **Simple MNIST NN from scratch (numpy, no TF/Keras)**. Disponível em: ¡https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras¿. Acesso em: 16 ago. 2024.