1 2 UNIVERSIDADE D 9 0
COIMBRA

deer.uc
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Departamento de Engenharia Eletrotécnica e de Computadores

Projeto de Sistemas Digitais

Professor: Jorge Dias

Relatório do Projeto Final: Implementação de uma Rede Neuronal em FPGA para Reconhecimento de círculos.

Autor:
Gonçalo Bastos
Leonardo Cordeiro

Numero de Estudante:
2020238997
2020228071

18 Junho, 2024

Abstract

Este relatório destina-se a apresentar o Projeto Final da disciplina de Projetos de Sistemas Digitais. Neste Relatório será exposto o desenvolvimento de um Multy Layer Perceptron de modo a detetar círculos através do input de imagens por parte do utilizador através do uso de um módulo Mouse e VGA, implementado na placa FPGA, Altera DE2-115.

1 Introdução

Com o avanço da tecnologia as redes neuronais têm ganho cada vex mais destaque em diversas aplicações de Aprendizagem Computacional. Recentemente, FPGA's têm sido consideradas para o desenvolvimento e aplicação de diversos algoritmos de Machine Learning, como o caso dos Multy Layer Perceptron, devido à flexibilidade e eficiência destas placas, tirando proveito da execução sequencial e da implementação de Arquiteturas de Hardware específicas.

Este trabalho foca-se na implementação de um Multy Layer Perceptron numa FPGA para deteção de círculos. Diferente de abordagens que se focam em tarefas complexas, o nosso objetivo é simplificar o problema e demonstrar a viabilidade e a eficiência de uma implementação de um MLP em FPGA, reduzindo o nosso problema para uma classificação binária: identificar a presença de círculos numa imagem. A implementação será efetuada usando o software Quartus 9.1, numa placa Altera de2-115.

2 State of the art

2.1 O uso da FPGA

O uso de FPGA's para implementação de algoritmos de aprendizagem computacional oferece vantagens significativas sobre as abordagens convencionais baseadas em CPU's e GPU's. Enquanto os CPU's processam instruções sequencialmente e GPU's, apesar do alto nível de paralelismo, são projetadas para operações gráficas, as FPGA's permitem a execução em paralelo de várias operações com baixa latência e eficiência. Isso resulta em uma aceleração significativa do processamento, tornando as FPGA's ideais para aplicações de aprendizagem computacional em tempo real, e ainda proporcionam uma alta eficiência energética em comparação com as GPU's. Além disso, a flexibilidade das FPGA's permite ajustes de Hardware sem necessidade de projetar um novo chip, o que pode ser útil para testar diferentes arquiteturas e tirar o máximo partido das placas. O uso de FPGA's pode vir a ser útil em muitas das aplicações geradas pelo aumento crescente da Inteligência Artificial, nas quais o escasso acesso a energia e o processamento de tempo real são fatores cruciais para o possível funcionamento do sistema, como o caso dos drones.

2.2 Rede Neuronal

Uma Rede Neuronal(NN) é um modelo de aprendizagem computacional composto por unidades de neurónios artificiais. Esses neurónios estão organizados em camadas: a camada de entrada, que recebe os dados, as camadas ocultas, que processam os dados e a camada de saída que fornece o resultado final. Cada neurónio está conectado aos outros através dos pesos e do bias, que ajustam a importância de cada entrada. Durante o treino, o objetivo é ajustar os pesos de forma a minimizar o erro entre a saída prevista e a entrada.

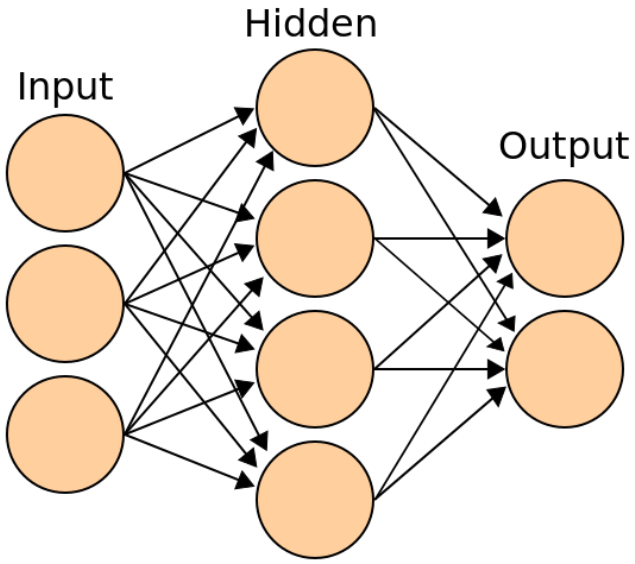

Figure 1: Neural Network

Um Multy Layer Perceptron (MLP) é uma rede neural feedforward totalmente conectada, onde cada neurónio de uma camada está conectado a todos os neurónios da camada seguinte. O MLP é composto por uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída. O funcionamento do MLP baseia-se em operações matemáticas que incluem a multiplicação de entradas (x) pelos pesos (w), a soma dos produtos resultantes ($\sum$), a adição de um bias (b) e a aplicação de uma função de ativação (f). Matematicamente, a saída de um neurónio (y) pode ser expressa como:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Onde x_i são as entradas, w_i são os pesos associados, b é o bias, e f é a função de ativação, como a sigmoide $f(x) = \frac{1}{1+e^{-x}}$ ou a ReLU $f(x) = \max(0, x)$. O treinamento do MLP envolve ajustar os pesos e bias para minimizar o erro entre a saída prevista e a saída real, utilizando algoritmos de otimização como o gradiente descendente. A backpropagation é o método utilizado para atualizar os pesos, onde o erro é propagado de volta através da rede, e os gradientes dos pesos são calculados para ajustar os valores na direção que reduz o erro. Isso permite que o MLP aprenda relações complexas entre as entradas e saídas.

3 Plano de Desenvolvimento e Testes

3.1 Requerimentos

- Deve ser implementada um MLP em Hardware, de modo a permitir a deteção de círculos;
- Os módulos devem ser definidos individualmente de modo a permitir a flexibilidade na definição do MLP;
- O modelo deve ser pré-treinado, de modo a permitir a integração da rede na FPGA;
- Deve ser possível ao utilizar realizar o input de uma imagem através da utilização de um módulo de Mouse e VGA;
- O sistema deve detetar se a imagem é ou não um círculo;

3.2 Plano de Desenvolvimento

1. Semana 1: Planeamento e Design
    - Definição dos requisitos detalhados do sistema.
    - Especificação da arquitetura do MLP e dos componentes de hardware no FPGA.
    - Planeamento do cronograma detalhado para as próximas semanas.
2. Semana 2: Desenvolvimento Inicial de Hardware
    - Implementação do modelo básico de Neurónio.
    - Teste e verificação do modelo de neurónio utilizando simulações.
    - Início da integração do modelo de neurónio em camadas.
3. Semana 3: Desenvolvimento e Integração de Camadas
    - Continuação da implementação e teste das camadas ocultas e de saída.
    - Verificação das conexões entre as camadas e ajustes conforme necessário.
    - Desenvolvimento do módulo VGA para interface gráfica.
4. Semana 4: Treino do Modelo em Python
    - Criação do modelo MLP em Python utilizando TensorFlow.
    - Treinamento do modelo com dados de imagem de círculos e não-círculos.
    - Extração dos pesos e bias treinados para uso na FPGA.
5. Semana 5: Integração e Teste de Hardware
    - Integração dos pesos e bias treinados no modelo de hardware.

- Teste do sistema completo no FPGA, incluindo o módulo VGA e a entrada do mouse.
- Ajustes de hardware e software conforme necessário.

6. Semana 6: Testes Finais e Validação
   - Execução de testes abrangentes para validar a precisão e funcionalidade do MLP.
   - Documentação dos resultados dos testes e ajustes finais.
   - Preparação de relatórios e documentação final do projeto.

3.3 Plano de Testes

1. Semana 1-2: Teste de Unidade
   - Teste de componentes individuais, como neurónios e camadas, em simulação.
   - Verificação da funcionalidade básica e correção de erros.
2. Semana 3-4: Teste de Integração
   - Teste da integração entre camadas e componentes.
   - Verificação da comunicação entre o modelo MLP e o módulo VGA.
   - Ajustes na interface do mouse para garantir precisão na entrada de dados.
3. Semana 5: Teste de Sistema
   - Teste do sistema completo no FPGA com dados reais.
   - Validação do desempenho do MLP para detecção de círculos.
   - Análise de desempenho e otimização, se necessário.
4. Semana 6: Teste de Aceitação
   - Execução de testes finais para garantir que todos os requisitos foram atendidos.
   - Documentação dos resultados dos testes e correção de quaisquer problemas remanescentes.
   - Aprovação final e preparação do sistema para entrega.

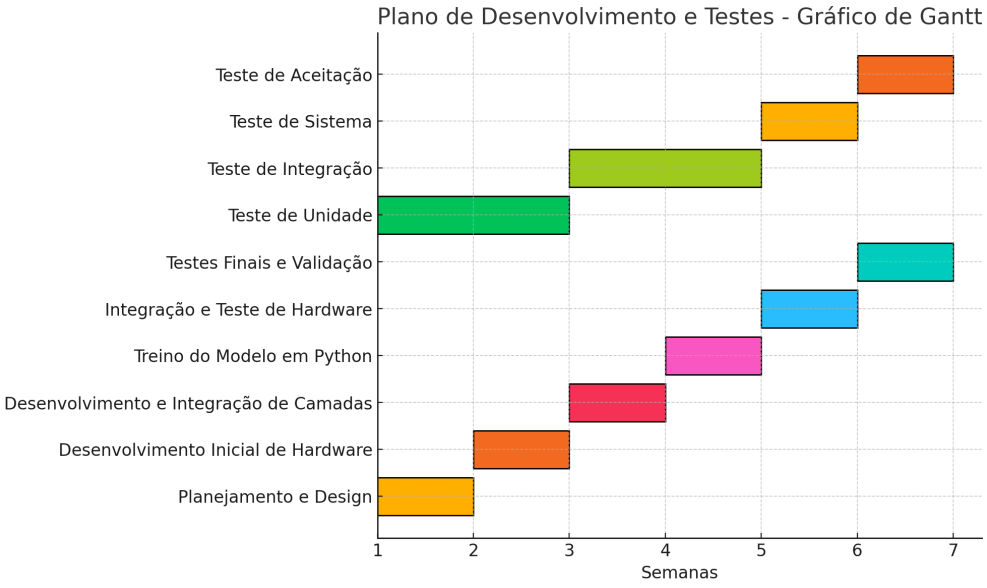

Figure 2: GANT Chart

4 Módulos

4.1 Neurónio

O módulo neurónio é responsável por simular um único neurónio numa camada de rede neural. Ele executa as seguintes tarefas:

- **Inicialização de Pesos**: Carrega os pesos pré-treinados na memória.
- **Inicialização de Bias**: Carrega os valores de bias na memória.
- **Multiplicação de Entradas**: Multiplica a entrada pelo peso correspondente.
- **Acumulação**: Soma os produtos e adiciona o bias.
- **Função de Ativação**: Aplica uma função de ativação (ReLU ou Sigmoide) ao resultado.
- **Saída**: Produz o valor final de saída do neurónio.

O módulo neurónio é parametrizado, permitindo flexibilidade em termos do número de pesos, largura de dados, tamanho da sigmoide. Os parâmetros também incluem caminhos de ficheiros para inicialização de bias e pesos.

Variáveis locais são definidas para gerir os endereços, dados de entrada, pesos, produtos e somas durante o cálculo.

O módulo inclui um mecanismo para carregar pesos e bias de ficheiros externos se a diretiva 'pretrained' estiver definida. Caso contrário, usa os valores fornecidos de 'weightValue' e 'biasValue'.

O módulo de neurónio realiza a multiplicação da entrada com o peso e acumula os resultados, adicionando o bias no final.

O módulo inclui uma diretiva para imprimir o valor de saída quando 'outvalid' é afirmado.

O funcionamento do módulo 'neurónio' baseia-se em operações matemáticas fundamentais que incluem a multiplicação de entradas (x) pelos pesos (w), a soma dos produtos resultantes ($\sum$), a adição de um bias (b) e a aplicação de uma função de ativação (f). Matematicamente, a saída de um neurónio (y) pode ser expressa como:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Onde:

- x_i são as entradas,
- w_i são os pesos associados,
- b é o bias, e
- f é a função de ativação, que pode ser a função sigmoide $f(x) = \frac{1}{1+e^{-x}}$ ou a ReLU $f(x) = \max(0, x)$.

Durante o treino da rede, o objetivo é ajustar esses pesos e bias para minimizar a diferença entre a saída prevista e a saída real, utilizando algoritmos de otimização como o gradiente descendente.

O diagrama de alto nível do módulo de neurónio pode ser representado com os seguintes inputs e outputs:

- **clk**: Sinal de clock.
- **rst**: Sinal de reset.
- **myinput**: Dados de entrada para o neurónio.
- **myinputValid**: Validade dos dados de entrada.
- **weightValid**: Validade dos dados de peso.
- **biasValid**: Validade dos dados de bias.
- **weightValue**: Valor do peso para inicialização.
- **biasValue**: Valor do bias para inicialização.
- **config_layer_num**: Número da camada para configuração.

- **config_neuron_num**: Número do neurónio para configuração.
- **out**: Dados de saída do neurónio após processamento.
- **outvalid**: Validade dos dados de saída.

O módulo neurónio proporciona uma maneira flexível e eficiente de implementar um único neurónio numa rede neural em um FPGA. Ao parametrizar o módulo, ele pode ser facilmente adaptado para várias configurações de rede e funções de ativação.

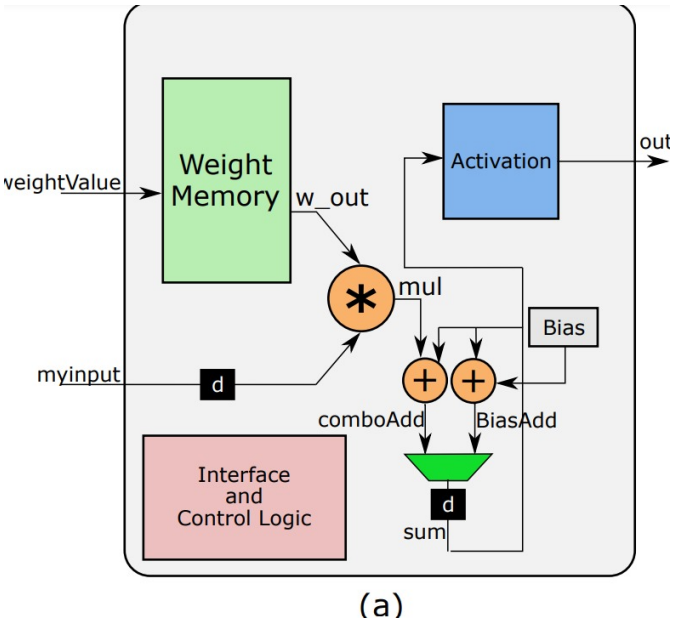

Figure 3: Implementação High-Level do Neurónio

4.2 Funções de ativação

4.2.1 ReLU

A função de ativação ReLU (Rectified Linear Unit) é implementada no módulo 'ReLU' em Verilog. Este módulo recebe um valor de entrada 'x', aplica a função ReLU, e produz o resultado na saída 'out'.

A função ReLU é definida matematicamente como:

$$f(x) = \max(0, x)$$

Esta função retorna x se x for maior ou igual a zero; caso contrário, retorna zero. A implementação em Verilog verifica se o valor de entrada é maior ou igual a zero. Se for, verifica se há saturação positiva. Caso contrário, retorna o valor de entrada ajustado. Se o valor de entrada for negativo, retorna zero.

Se a parte inteira do valor de entrada transbordar para o bit de sinal, a saída é saturada no valor positivo máximo representável. Este processo garante que a saída não exceda o intervalo permitido pela largura de dados especificada.

4.2.2 Sigmoide

A função de ativação sigmoide é outra função amplamente utilizada em redes neurais. É definida pela expressão:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Esta função retorna um valor entre 0 e 1, o que a torna adequada para modelos que precisam prever probabilidades. Na nossa implementação, os valores da função sigmoide foram calculados previamente em Python e armazenados num ficheiro MIF (Memory Initialization File) para serem usados no FPGA.

Os valores da função sigmoide são calculados usando a expressão matemática acima. Em Python, isto pode ser feito utilizando bibliotecas matemáticas para gerar uma tabela de valores correspondentes. Estes valores são então normalizados e armazenados em um ficheiro MIF.

No FPGA, os valores pré-calculados da função sigmoide são carregados na memória ROM. Durante a operação da rede neural, a entrada é utilizada para indexar a tabela da função sigmoide e obter o valor correspondente, que é então usado como a saída ativada do neurónio.

As funções de ativação ReLU e sigmoide desempenham papéis cruciais no funcionamento de redes neuronais, introduzindo as não linearidades necessárias para a aprendizagem eficaz. A implementação destas funções em Verilog, combinada com a utilização de valores pré-calculados da função sigmoide..

4.3 Pré-treino da rede em Software

Para capturar os pesos e bias utilizados na implementação da rede neural no FPGA, foi desenvolvido um código em Python que utiliza a biblioteca 'network2' para treinar a rede neuronal. O processo de treino envolve várias etapas, desde o carregamento das imagens de treino até a divisão dos dados e o treino propriamente dito. Os dados são preparados no formato esperado pela rede neural. As imagens de treino, validação e teste são convertidas em arrays uni-dimensionais, e os rótulos são transformados para arrays de uma única entrada. Esta preparação é crucial para garantir que os dados são compatíveis com a estrutura da rede neural utilizada.

A rede neural é inicializada com uma arquitetura específica. No nosso caso, a arquitetura consiste em 784 neurónios na camada de entrada, duas camadas ocultas com 30 neurónios cada, uma camada intermediária com 10 neurónios e uma camada de saída com 1 neurónio.

O treino da rede neural é realizado utilizando o método de Gradiente Descendente Estocástico (SGD). Durante o treino, a rede passa por várias épocas, ajustando os pesos e bias com base nos dados de treino. A taxa de aprendizagem e o parâmetro de regularização lambda (λ) são configurados para otimizar o processo de treino. A validação é feita ao final de cada época para monitorizar a precisão da rede neural.

Após o treino, os pesos e bias de cada neurónio são calculados e guardados num ficheiro .mif. Este ficheiro é utilizado posteriormente para inicializar a rede neural no FPGA, garantindo que os mesmos pesos e bias treinados em Python são aplicados na implementação em hardware.

Este processo completo de treino e captura de pesos e bias é essencial para a criação de uma rede neural eficiente e precisa, pronta para ser implementada na FPGA.

5 Trabalho Desenvolvido

5.1 Neurónio

5.1.1 Máquina de Estados Finitos

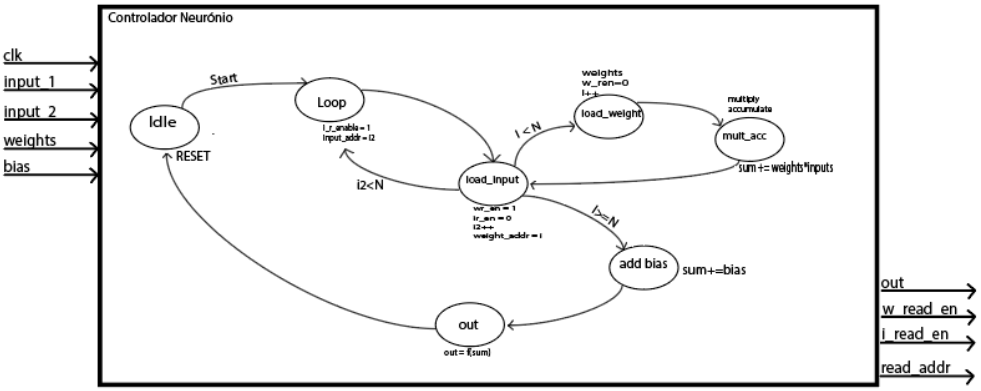

Figure 4: Finite State Machine

Esta máquina de estados visa a implementar corretamente o processamento de um neurónio, processando entradas binárias com a aplicação de pesos e bias para calcular uma soma ponderada.O Bloco *neuron* define os parâmetros principais e as portas de entrada e saída.As entradas (input_1 e input_2), pesos e bias são recebidas pelo neurónio. Saídas incluem a resposta do neurónio e sinais de controle para leitura da memória ROM. É utilizada uma máquina de estados para controlar o comportamento do neurónio, desde a inicialização até a aplicação da função de ativação que determina a saída com base na soma ponderada das entradas.

O processo principal utiliza uma sequência de estados para controlar a leitura das entradas e pesos, acumulação da soma ponderada, e aplicação do bias. Em cada ciclo de relógio, o neurónio lê uma entrada e um peso, multiplica-os, e acumula o resultado em um somatório. Após processar todas as entradas, o bias é adicionado à soma, e uma função de ativação determina a saída do neurónio: se a soma for positiva ou zero, a saída é 1, caso contrário, é 0. Esse método estruturado garante que o neurónio processa as entradas corretamente, oferecendo uma base sólida para simulações e implementações mais complexas em redes neuronais.

Uma das nossas principais preocupações foi construir uma arquitetura de um neurónio que fosse possível escalar, para tal criámos entradas parametrizáveis o que nos permitirá definir o número de inputs e outputs de cada neurónio, de modo a ser possível a integração deste módulo numa MLP.

5.1.2 Representação dos números

Os pesos e os bias podem ser positivos ou negativos, mas geralmente estes números tem uma parte frácionária. Os números frácionários podem ser representados por **floating point representation** e **fixed point representation**. A representação em floating point ajudam a representar números grandes e oferecem maior precisam mas a sua implementação e mnaipulação pode ser díficil, além disso consume muitos recursos. Deste modo, optámos por a representação eem valores fixos. Uma vez que podemos normalizar os valores de entrada entre -1 e 1 não iremos ter problemas com esta representação. Pode existir alguma degradação da precisão as se não ocorrer overflow, a aritmética de valor fixo vai dar um bom resultado. Nesta representação temos de definir o número de bits da parte inteira e da parte fracionada. Como referido anteriormente os pesos e os bias podem ser números negativos, logo para realizar esta representação utilizámos o complemento para 2.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity fixed_point_adder_noOverflow is
6      Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
7             b : in  STD_LOGIC_VECTOR (7 downto 0);
8             sum : out  STD_LOGIC_VECTOR (15 downto 0));
9  end fixed_point_adder_noOverflow;
10
11 architecture Behavioral of fixed_point_adder_noOverflow is
12 begin
13     process(a, b)
14         variable signed_a, signed_b : SIGNED(7 downto 0);
15         variable temp_sum : SIGNED(15 downto 0); -- 16 bits to store the result
16     begin
17         -- Convert inputs to signed
18         signed_a := SIGNED(a);
19         signed_b := SIGNED(b);
20
21         -- Perform addition
22         temp_sum := resize(signed_a, 16) + resize(signed_b, 16); -- Extend both to 16 bits and add them
23
24         -- Assign the 16-bit result to the sum
25         sum <= STD_LOGIC_VECTOR(temp_sum);
26     end process;
27 end Behavioral;
28
```

Figure 5: Fixed Point Arithemtics

5.1.3 Funções de Ativação

A maioria das redes neuronais usam funções de ativação não lineares como a sigmoid ou a tangente hiperbólica. Criar circuitos digitais que geram estas funções é muito desafiador e estas requerem muitos recursos. Nós optámos por calcular estes valores previamente e armazená-los numa ROM, que vai ser chamada comoo Look Up Table(LUTs). No nosso teste do neurónio, que consiste em codificar o comportamento de uma porta AND, a função de ativação foi apenas subsituida por um threshold.

6 Implementação Futura

6.1 Design final do Hardware completo

A arquitetura da rede neural implementada é composta por cinco camadas, cada uma com um número específico de neurónios e funções de ativação. A rede recebe imagens de 28x28 pixeis como entrada, totalizando 784 neurónios na camada de entrada. As camadas subsequentes processam essas entradas utilizando pesos e bias pré-treinados, aplicando funções de ativação adequadas.

A definição das camadas da rede é a seguinte:

- **Camada 1**: Contém 30 neurónios, cada um com 784 pesos. A função de ativação utilizada é a Sigmoide.
- **Camada 2**: Contém 30 neurónios, cada um com 30 pesos provenientes da camada anterior. A função de ativação utilizada é a Sigmoide.
- **Camada 3**: Contém 10 neurónios, cada um com 30 pesos provenientes da camada anterior. A função de ativação utilizada é a Sigmoide.

- **Camada 4**: Contém 10 neurónios, cada um com 10 pesos provenientes da camada anterior. A função de ativação utilizada é a Sigmoide.
- **Camada 5**: Contém 10 neurónios, cada um com 10 pesos provenientes da camada anterior. A função de ativação utilizada é Hardmax, esta retorna a previsão da rede em binário.

Cada camada da rede é responsável por realizar operações de multiplicação de entrada pelos pesos, acumulação de produtos e adição de bias, seguida pela aplicação da função de ativação. Os valores de peso e bias são pré-calculados e armazenados em ficheiros de memória que são carregados na inicialização.

A rede utiliza máquinas de estados para realizar o pipeline dos dados através das camadas. Cada camada processa os dados e os transfere para a próxima camada até que a camada final produza a saída binária.

6.2 Interface com o usuário

Para a implementação da UI, decidimos ajustar o módulo utilizado no mini-projeto de desenho livre de modo a ser possível ao utilizador desenhar dentro de uma área específica. De seguida, o módulo deve de converter a imagem desenhada para uma memória que seja possível enviar a imagem desenhada para a rede neuronal de modo a realizar a previsão.

Para permitir que o utilizador desenhe no VGA usando o mouse, foram implementadas lógica e módulos adicionais. A abordagem adotada inclui:

- Inicialização do VGA com a cor branca.
- Limitação da área de desenho no VGA para uma região de 28x28 pixels, centrada no ecrã de resolução 640x480.
- Permissão para que o utilizador desenhe a preto dentro desta área utilizando o mouse.
- Implementação de uma RAM que funciona em paralelo com o VGA_SYNC para armazenar e recuperar as cores dos pixels. Esta RAM garante que a cor de cada pixel possa ser lida ou atualizada conforme necessário, permitindo operações de desenho e apagamento baseadas na entrada do utilizador (botões do mouse) e nas coordenadas do cursor do mouse.
- Implementação de uma segunda RAM para armazenar as cores dos pixels da área 28x28 do VGA, uma vez que o resto do ecrã permanecerá branco. Esta RAM serve como uma captura da zona de desenho.

7 Conclusão e Análise Crítica

O objetivo final deste projeto não foi alcançado com sucesso, encontrámos várias dificuldades com o desenvolvimento deste projeto, inicialmente por uma abordagem pouco conservadora, num estilo Top-Down. Na segunda abordagem, começámos por implementar os módulos por partes e por necessidade, tentando sempre utilizar os módulos parametrizáveis de modo a possibilitar a escalabilidade do projeto. Contudo, alcançámos com sucesso a implementação funcional de um neurónio que nos permitirá a escalabilidade para uma multy layer perceptron, além disso, implementámos funções de ativação e fnções de aritmética fixa.

Em suma, apesar de objetivo final e do sistema não ser usável, este projeto permitiu-nos desenvolver conhecimentos numa área em constante crescimento como é a Inteligência Artificial, conseguimos superar as dificuldades do desenvolvimento de uma arquitetura específica de uma rede neuronal em Hardware. Com a pesquise efetuada para este projeto também retemos noções da importância da utilização das FPGA's na área da Inteligência Artificial, bem como a utilização destas arquiteturas para superar barreiras atualmente impostas pelos os dispositivos convencionais como a Power-Wall e a capacidade de processamento em tempo real.

[1][2][4][3]

References

[1] Wentao Cui and Poplar Wang. *FPGA_NN*. https://github.com/FSq-Poplar/FPGA_NN. 2019.

[2] Nikhil Gaikwad et al. "Efficient FPGA Implementation of Multilayer Perceptron for Real-Time Human Activity Classification". In: *IEEE Access* PP (Feb. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2900084.

[3] Vipin Kizheppatt. *neuralNetwork*. https://github.com/vipinkmenon/neuralNetwork. 2020.

[4] A. Muthuramalingam, S. Himavathi, and E. Srinivasan. "Neural Network Implementation Using FPGA: Issues and Application". In: *Int. J. Inform. Technol.* 4 (Nov. 2007).