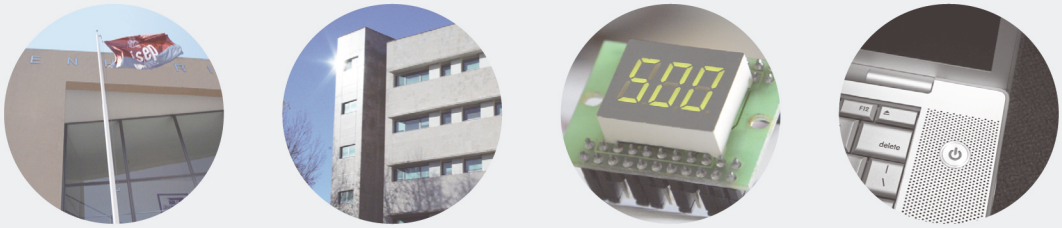
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Benchmark de Sistemas Embebidos para Machine Learning em Visão Computacional

MIGUEL ÂNGELO LOURENÇO LOPES
Agosto de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Electrotécnica
Rua Dr. António Bernardino de Almeida, 431, 4249-015 Porto

Benchmark de Sistemas Embebidos para Machine Learning em Visão Computacional

Mestrado em Engenharia Eletrotécnica e de Computadores - Sistemas Autónomos

Miguel Ângelo Lourenço Lopes

Orientador:
André Miguel Pinheiro Dias
Coorientador:
José Miguel Soares de Almeida

Ano Letivo: 2020-2021

Resumo

Machine learning é um método poderoso para construir modelos que usam dados para fazer previsões: de uma forma simplificada, é um campo que dá aos computadores a habilidade de ”aprenderem” sem terem de ser programados diretamente.

Os sistemas embebidos, normalmente os microcontroladores, são caracterizados pela existência de limitações no número de ciclos de processamento, memória, tamanho, peso, consumo de energia e custo, o que torna o *machine learning* difícil de implementar.

No entanto com a evolução da tecnologia, o *machine learning* está agora acessível num maior número de dispositivos com baixos recursos computacionais.

Esta dissertação centrou-se na comparação de alguns sistemas com baixos recursos computacionais para *machine learning* na classificação de imagens. Esta comparação foi obtida através da análise de diferentes arquiteturas de *hardware*, escolhendo um algoritmo que exige uma grande quantidade de cálculos e utilizando diferentes bibliotecas para *machine learning*.

Palavras-Chave: *machine learning*, sistemas embebidos, visão artificial e arquiteturas de *hardware*

Abstract

Machine learning is a powerful method for build models that use data to make provisions, in a simple way, is a field that give to computers the ability to ”learn” without having to be programmed directly.

Embedded systems, usually microcontrollers, are characterized by the limitations in the number of cycles of processament, memory, size, weight, power and cost, what makes machine learning hard to implement in embedded systems.

However, as technology evolves, machine learning is easy to implement on low-resource devices.

The main focus of this dissertation is the comparison of low resources systems for machine learning in image classification. This comparison was made with the analysis of different hardware architectures, choosing an algorithm that requires a large amount of calculations and using different libraries for machine learning.

Keywords: machine learning, embedded systems, artificial vision and hardware architectures

Índice

Lista de Figuras

Lista de tabelas

Lista de Equações

Glossary

ALU Arithmetic Logic Unit. 13

ARM Advanced RISC Machine. 21

CISC Complex Instruction Set Computer. 42

CLB Configurable Logic Block. 18

CNNs Convolutional neural networks. 8

CPU *Central Processing Unit*. 2

CSV Comma-Separated Values. 31

DRAM Dynamic Random-Access Memory. 13

FPGA Field-Programmable Gate Array. 2

GB Gigabyte. 21

GPIO General Purpose Input/Output. 21

GPU Graphics Processing Unit. 2

ISEP Instituto Superior de Engenharia do Porto. 2

KNN k-nearest neighbors. 9

LSA Laboratório de Sistemas Autónomos. 2

MB Megabyte. 32

MILA Montreal Institute for Learning Algorithms. 26

NPU *Neural Processing Unit*. 2

RAM Random Access Memory. 21

ReLU Rectified Linear Unit. 8

RGB Red Green Blue. 1

RISC Reduced Instruction Set Computer. 42

RTOS Real Time Operating System. 21

TPU *Tensor Processing Unit*. 17

Agradecimentos

A presente dissertação não poderia ter sido concluída sem ajuda de várias pessoas.

Em primeiro lugar não posso deixar de agradecer ao meu orientador Professor Doutor André Miguel Pinheiro Dias e coorientador Professor José Miguel Soares de Almeida, por toda a paciência e empenho ao longo desta dissertação.

Agradeço igualmente aos meu colegas do mestrado.

Agradeço em especial ao Ronaldo Marques e Eduarda Barbosa pelas revisões que deram ao longo da elaboração da dissertação.

Por último quero agradecer à minha família e amigos por todo o apoio dado.

Capítulo 1

Introdução

Atualmente, com a capacidade computacional existente, o tópico de *machine learning* assumiu uma grande relevância. O *Machine learning* é hoje a fundação de imensas aplicações importantes como a pesquisa *web*, reconhecimento de voz, recomendação de produtos, visão computacional e muitas outras [11].

Esta dissertação foca-se numa das aplicações do *machine learning*, a visão computacional, que tenta replicar aquilo de que a visão humana é capaz.

O campo da visão computacional tem beneficiado muito com o *machine learning* [12], dado que é mais eficiente utilizar modelos para identificação de imagens. Sem a utilização de modelos de *machine learning*, seria necessário codificar manualmente os modelos para a identificação de características numa imagem.

Comparando com a visão humana, a visão computacional assume uma perspectiva semelhante, ambas assentam na interface entre um sensor e um interpretador. Na visão computacional, a câmara é o equivalente ao olho humano e pode ser de vários tipos como, por exemplo, *Red Green Blue (RGB)*, monocromática, hiperespectral, entre muitas outras. No caso do interpretador, a combinação do *hardware* e do *software* é usada para imitar o cérebro humano, que vai ser responsável por interpretar o sinal enviado pela câmara [13].

Trabalhar com *machine learning* exige bastante poder computacional, sendo que os *developers* raramente pensam na gestão de memória, já que nos dias de hoje, até os telemóveis têm um bom poder de processamento [14].

Correr tarefas computacionais intensivas, como *machine learning*, num sistema embebido é um desafio. Algoritmos de *machine learning* necessitam de ser ajustados para serem executados em *hardware* com recursos computacionais limitados e com restrições de energia. No entanto, quando o *machine learning* estiver adaptado para sistemas de baixos recursos, irá desbloquear o uso valioso de dados que são, atualmente, descartados devido ao custo, largura de banda ou restrições de energia [15].

Através da junção da visão computacional com *machine learning* em sistemas embebidos, é criada uma solução que pode ter aplicações em diversas áreas como: linhas de montagem, robótica, tecnologia biomédica, drones, sistemas de assistência ao condutor, condução autónoma, entre outras.

A quantidade e a qualidade de aplicações que utilizam visão embebida está a crescer, sendo uma grande promessa para o futuro, mas ainda existem problemas com a exigência de demasiado poder de processamento dos algoritmos ou mesmo a dificuldade de implementação dos mesmos.

1.1 Enquadramento e Motivação

O trabalho apresentado nesta dissertação tem como principal objetivo a comparação de desempenhos entre diferentes sistemas embebidos na execução de algoritmos de *machine learning* aplicado a visão computacional.

O projeto surgiu no âmbito da unidade curricular Tese/Dissertação, unidade integrante do plano de estudos do Mestrado de Sistemas Autónomos e a respetiva dissertação foi desenvolvida no Laboratório de Sistemas Autónomos (LSA), pertencente ao Instituto Superior de Engenharia do Porto (ISEP).

O tema em questão surgiu da necessidade de analisar como diferentes tipos de *hardware* (*Central Processing Unit (CPU)*, *Graphics Processing Unit (GPU)*, *Neural Processing Unit (NPU)* e *Field-Programmable Gate Array (FPGA)*) se comportam na execução de algoritmos de *machine learning*, para visão computacional. É pretendido verificar qual das arquiteturas apresenta melhor desempenho (velocidade de processamento e fiabilidade de resultados). Irá também ser comparado como diferentes bibliotecas se comportam nos sistemas embebidos escolhidos, para chegar à melhor correspondência entre *board* e biblioteca.

1.2 Objetivos

Esta dissertação tem os seguintes objetivos principais:

- Efetuar o estudo prévio de tópicos relevantes para a dissertação, como o *machine learning*, visão artificial, sistemas embebidos e perceber como estes conceitos se relacionam;
- Caracterizar e avaliar técnicas de *machine learning* aplicadas a sistemas de visão:
- Investigar as diferentes arquiteturas de *hardware* para o processamentos de dados;
- Escolher diferentes sistemas embebidos para a comparação;
- Procurar diferentes bibliotecas para a implementação dos algoritmos;
- Implementar e testar o algoritmo escolhido nas diferentes placas;
- Comparar resultados em termos de *performance* e fiabilidade entre os diferentes sistemas.

1.3 Estrutura da Dissertação

A estrutura desta dissertação é divida em seis partes: introdução, estado de arte, arquitetura, implementação, testes e validação de resultados e conclusão.

Neste primeiro capítulo é feita uma breve introdução ao tema, é exposto o enquadramento e a motivação do trabalho e por fim são apresentados os objetivos.

No segundo capítulo é apresentado o estado de arte, onde são apresentados alguns algoritmos para a classificação de imagens, é abordado o tema da visão computacional e é feita uma análise dos sistemas embebidos.

O terceiro capítulo apresenta a arquitetura das redes neurais a serem testadas.

No quarto capítulo é demonstrado como as redes neurais foram desenvolvidas com recurso às diferentes bibliotecas, é também apresentado o *dataset* utilizado e como os sistemas operativos foram instalados nas *boards*.

O quinto capítulo expõe os resultados para as diferentes *boards* e são feitas também algumas comparações.

Por fim, no sexto capítulo é apresentada a conclusão desta dissertação.

Capítulo 2

Fundamentos

Neste capítulo irão ser apresentados os fundamentos de *machine learning*, que serão objeto de estudo ao longo desta dissertação .

Na secção 2.1 realiza-se uma breve apresentação do *machine learning*, são apresentados diferentes algoritmos e como o *machine learning* se combina com a visão artificial.

A secção 2.2 trata da visão computacional propriamente dita, é feita uma breve apresentação, onde são referidas algumas das suas aplicações e a forma como é realizado o processamento.

A secção 2.3 é acerca dos sistemas embebidos: como estão incorporados com a visão artificial, diferentes arquiteturas de *hardware*. São apresentados alguns modelos de placas que suportam *machine learning* e por fim, as bibliotecas usadas para a implementação.

2.1 Machine Learning

Nesta secção irá ser tratado o tópico de *machine learning*, os seus benefícios e diferentes tipos de *machine learning*. Estes conceitos terão como maior foco as aplicações em visão artificial.

O *machine learning* surgiu na década de 50, não muito depois do primeiro computador de uso geral, quando Alan Turing criou o primeiro *Turing test* [16], para determinar se um computador conseguia pensar ou mais precisamente, aprender. O Dr. Turing acreditava que as máquinas eventualmente iriam conseguir pensar e aprender e assim passar o seu teste. Com base nesta suposição, muitos cientistas desenvolveram a ideia de inteligência artificial. O diagrama da figura 2.1 mostra como os termos de inteligência artificial, *machine learnig* e *deep learning* se relacionam.

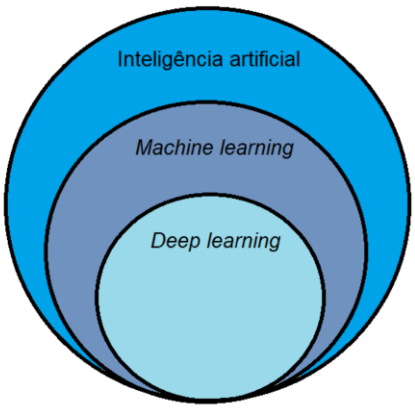


Figura 2.1: Inteligência artificial, *machine learning* e *deep learning*

Imenso trabalho foi feito desde que a ideia surgiu, com diferentes algoritmos sendo descobertos [17].

Nos dias de hoje, inteligência artificial, *machine learning* e *deep learnig* são algumas das expressões em voga, mais faladas e estudadas na área de ciência da computação [18].

2.1.1 Tipos de machine learning

Machine learning, de uma forma muito resumida, consiste em algoritmos que têm a capacidade de aprender com a experiência.

Recolha e preparação de dados -> algoritmo de aprendizagem -> previsão

Os algoritmos podem ser divididos em três tipos:

- ***Supervised learning*** - Este tipo de algoritmos é treinado para no final escolher a função que melhor descreve os dados de entrada. Estes algoritmos criam relações e dependências entre as previsões de saída e os dados de entrada. Têm como objetivo problemas de regressão e classificação; [19]
- ***Unsupervised learning*** - Com este tipo de *learning*, o computador é treinado com dados não rotulados e são principalmente usados para a aprendizagem de regras de associação; [19]
- ***Reinforcement learning*** - Este método usa a observação em conjunto com interacções do ambiente e sendo executado repetidamente numa tarefa definida com um *feedback* de recompensa, para maximizar o resultado ou minimizar o risco. São principalmente utilizados para resolver problemas lógicos e de planeamento. [19]

A classificação de imagens utiliza algoritmos do tipo *supervised learning*.

De uma forma mais detalhada, algoritmos de *supervised learning* são projetados para aprender com o exemplo. Quando um algoritmo deste tipo é treinado, as entradas estão emparelhadas com as saídas, ou seja, durante o treino, o algoritmo procura por padrões nos dados que se relacionam com as saídas desejadas. Após o treino, o algoritmo receberá

entradas nunca vistas pelo mesmo e com base nos dados da fase de treino, determinará como os dados serão classificados.

Na figura 2.2 é apresentada uma *pipeline* para *supervised learning*. Para a fase de treino é necessário fornecer um *dataset* com dados de entrada e as *labels* associadas aos mesmos, essas informações são aplicadas num modelo que vai ser treinado e é gerado uma modelo de classificação, que consegue interpretar entradas de dados com base na fase de treino.

É importante realçar que este tipo de algoritmos variam apenas na fase de treino, o *predict* é igual para todos.

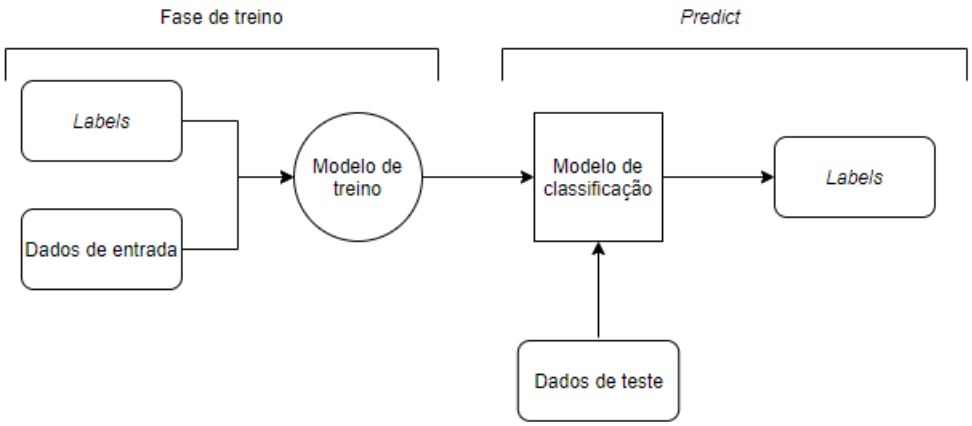


Figura 2.2: Esquema de *supervised learning*

2.1.2 Machine learning para classificação de imagens

A classificação de imagens nesta pesquisa usa a definição ampla tirada de [20].

A classificação de imagens tem sido uma área importante de pesquisa desde o início do *machine learning*, pois é uma das principais formas das máquinas interagirem com nosso mundo físico. Antes de se passar para o algoritmo utilizado para o processamento de imagens propriamente dito, serão apontados alguns desafios da visão artificial que são triviais para humanos, mas um desafio para máquinas:

- A orientação dos objetos, dependendo da posição da câmara;
- Variações de escala;
- Deformações, pois o meio ambiente não é estático;
- Nem sempre é possível obter todo o objeto numa só imagem;
- Condições de iluminação;
- O objeto pretendido pode estar misturado com o meio;
- O mesmo tipo de objetos pode ter diferentes formas.

2.1.2.1 Convolutional neural networks

Um algoritmo muito utilizado para a classificação de imagem é o *Convolutional neural networks (CNNs)*, foi projetado no Bell Labs em 1988 por Yan LeCun, sendo que foi inspirado em processos biológicos [21]. É considerado um dos maiores avanços no *supervised learning*, pois tende a ter uma demanda menor de pré-processamento, em comparação com outros algoritmos de classificação de imagens. A rede "aprende" filtros que num algoritmo tradicional precisariam de ser implementados manualmente.

A classificação de imagens com CNN utiliza uma imagem de entrada, que corresponde a uma matriz de pixeis, processa-a e classifica-a numa determinada categoria.

Convolução é a primeira camada a extrair os recursos da imagem. Nesta fase, são utilizadas pequenas partes da imagem que são multiplicadas por um filtro, o resultado dessa multiplicação é chamado mapa de características. Podem ser aplicados diferentes filtros para diferentes operações, como a detecção de bordas, desfoque ou nitidez [1].

A Camada de *pooling* tem como intenção reduzir o número de parâmetros quando a quantidade de dados é muito elevada, apesar da imagem ser reduzida são retidas as informações mais importantes[1].

Na camada totalmente conectada, a matriz gerada pelo *pooling* é transformada num vetor que depois alimenta uma rede neural como na figura 2.3.

Em CNN é importante conhecer o *Rectified Linear Unit (ReLU)*, que tem como objetivo introduzir a não linearidade no sistema, evitar que valores negativos sejam adicionados à CNN.

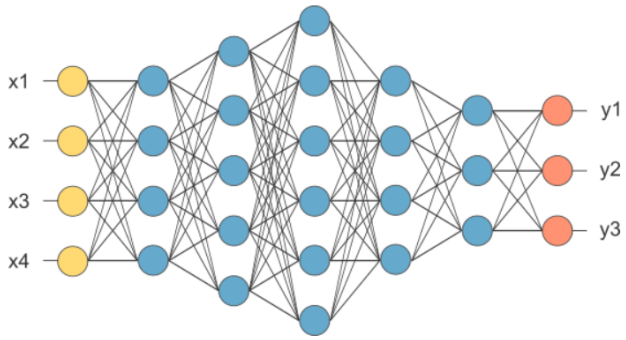


Figura 2.3: Rede Neural [1]

Na figura 2.4 está representado um modelo de uma arquitetura CNN com todas as camadas referidas acima.

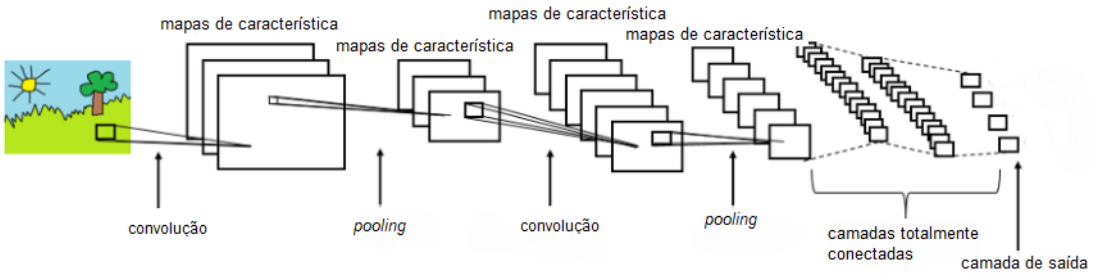


Figura 2.4: Arquitetura CNN completa [1]

2.1.2.2 Decision Tree

O algoritmo *decision tree* consiste na construção de um modelo de classificação ou regressão em forma de árvore. Dado um conjunto de entradas, é possível mapear vários resultados, que irão ser consequências de escolhas anteriores [22].

Na imagem 2.5 é possível entender mais facilmente o algoritmo. Existe uma amostra com 4 animais (gaivotas, pinguins, ursos e golfinhos). Inicialmente, os animais são separados num conjunto que tem penas e num conjunto que não tem penas e, de seguida, é avaliado outro fator, que vai permitir atingir o animal em questão.

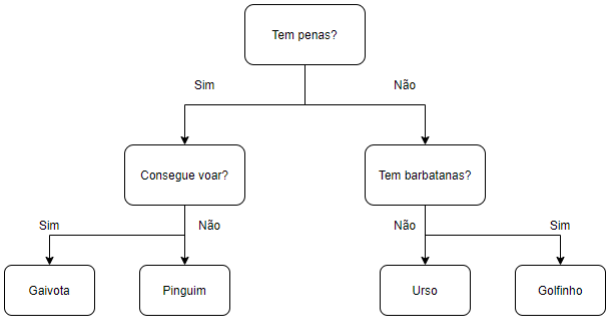


Figura 2.5: *Decision Tree*

As *decision trees* são o resultado de vários passos hierárquicos que ajudam a obter certas decisões. Para construir este tipo de algoritmos, existem dois passos: a introdução e a poda.

A introdução constrói a árvore e a poda vai removendo a complexidade.

2.1.2.3 K-Nearest Neighbors

O *k-nearest neighbors (KNN)* é um algoritmo baseado na *supervised learning* que pode ser usado para problemas de classificação e regressão.

O KNN assume que objetos semelhantes estão perto uns dos outros, o algoritmo depende dessa suposição para ser útil. O KNN reconhece como semelhança a ideia de

proximidade, ou seja, em uma imagem é calculada a distância entre dois pontos e decide se esses dois pontos estão próximos [23].

O KNN é composto por duas fases; a primeira é a determinação dos vizinhos mais próximos e a segunda é a determinação de classe, utilizando os vizinhos [2].

A sua desvantagem é que se torna bastante lento com uma maior número de variáveis e/ou exemplos.

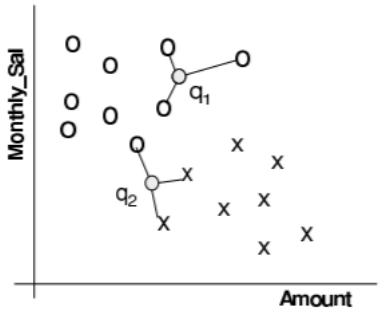


Figura 2.6: Simples exemplo da classificação KNN[2]

2.1.2.4 Naive Bayes

Naive Bayes é uma algoritmo probabilístico para classificar diferentes objetos. De forma a entender melhor o algoritmo, é necessário entender primeiro o teorema de Bayes. O teorema calcula a probabilidade de um evento A acontecer, dado que o evento B ocorreu.

Teorema de Bayes

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

Onde:

- P(A|B): Probabilidade de B acontecer, dado que A ocorreu;
- P(A): Probabilidade de A ocorrer;
- P(B): Probabilidade de B ocorrer.

Este modelo assume que a presença de um determinado recurso numa classe não está relacionada com a presença de qualquer outro recurso.

Para trabalhar com o Naives Bayes é necessário seguir um conjunto de passos que são; converter os dados fornecidos para uma tabela de frequência, obter a probabilidade de certos recursos da tabela anterior e por fim utilizar o o teorema de Bayes para calcular a probabilidade posterior.

A sua principal desvantagem é considerar que todas as variáveis são independentes, o que no mundo real é quase impossível [24].

2.1.2.5 Conclusões

O principal objetivo desta dissertação é obter uma *benchmark* de sistemas embebidos para *machine learning* em visão computacional. Para obter resultados relevantes no tema é necessário utilizar um algoritmo com um elevado volume de cálculos. Com base nesse argumento, o algoritmo que melhor se parece adequar é o CNN, pois durante a execução das diferentes etapas, convolução, *pooling* e camadas totalmente conectadas, são executadas milhares ou milhões de operações, sobretudo multiplicações matriciais.

Uma outra razão para o CNN ser uma escolha pertinente, é a capacidade do algoritmo "aprender" com base no *dataset* utilizado, o que torna o algoritmo mais simples de ser implementado, sem existir a necessidade de implementar alguns filtros de forma manual.

2.2 Visão computacional

Nesta secção irá ser feita uma introdução à visão artificial, bem como algumas das suas aplicações e o seu processamento.

2.2.1 Definição de visão computacional

A definição de visão computacional nesta pesquisa está a usar uma ampla definição tirada de [25].

No sentido de melhor entender o tópico da visão computacional, primeiro é necessário analisar os componentes do termo ("computador e visão").

Um computador é definido como uma máquina electrónica capaz de realizar vários processos, cálculos ou um conjunto de instruções definidas em *hardware* ou *software*.

A visão é algo que nos dá a compreensão de um ambiente, por meio da iluminação do ambiente no espectro da luz visível.

Através da junção dos dois termos, a visão computacional é a forma como as máquinas tentam entender o ambiente para chegar ao objetivo. Isto significa que a visão computacional é o processo pelo qual uma máquina ou um sistema cria um entendimento de um dado visual através de um ou mais algoritmos. Este entendimento é traduzido em decisões, classificações ou outras acções.

2.2.2 Aplicações e processamento

A visão computacional tem um amplo número de aplicações nos dias atuais, sendo por exemplo um ponto chave na condução autónoma da Tesla [26].

Entre outras aplicações estão:

- Reconhecimento facial -> Este tipo de algoritmos, procuram características faciais e comparam com uma base de dados de perfis faciais;
- Realidade aumentada -> A visão computacional é utilizada para detetar objetos no mundo real, para depois serem colocados na realidade virtual;
- Saúde -> Ajuda a automatizar algumas tarefas de detecção, como por exemplo a detecção de cancros na pele.

A detecção de um certo objeto ou a análise de uma cena, hoje em dia, continua a ser um processo desafiador para um computador, mas antes do chegada do *deep learning*, a visão computacional era muito limitada e exigia a criação de muito código manualmente.

O *Machine learning* fornece uma aproximação diferente, em vez de cada regra na interpretação da imagem precisar de ser codificada manualmente, recursos menores são programados para detetar padrões específicos na imagem e ainda oferecendo mais robustez para sistemas críticos com uma maior taxa de reconhecimento do objetivo.

2.3 Sistemas embebidos

Embedded systems are special-purpose computing systems embedded in application environments or in other computing systems and provide specialized support. [27]

Um sistema embebido tem como principais características, ser desenvolvidos para tarefas específicas, têm recursos computacionais limitados (não tem ecrãs, pouca memória disponível e pouco poder de processamento), já têm diversos periféricos incorporados no sistema, existe uma variedade de arquiteturas disponíveis em relação aos computadores pessoais e o preço deste tipo de sistemas é reduzido.

2.3.1 Arquiteturas de hardware

As escolhas de *hardware* para *machine learning* podem variar muito, as opções que vão ser analisadas são: GPU, CPU, NPU e FPGA.

2.3.1.1 CPU

Todo o computador tem um componente primário para executar lógica aritmética e controlo, o CPU. A sua função primária é executar, sequencialmente, instruções que são mantidas na memória do computador [28].

O funcionamento do CPU consiste em procurar uma instrução, executar, passar à instrução seguinte e assim sucessivamente até que seja desligado ou aconteça uma falha no sistema.

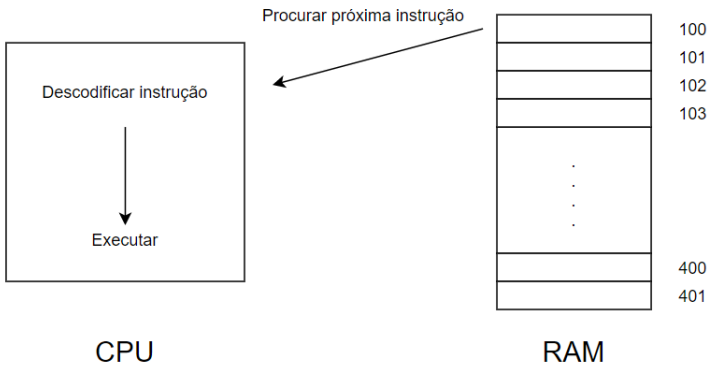


Figura 2.7: Funcionamento do CPU

Passos no funcionamento de um CPU:

- A instrução é procurada na memória;
- A instrução é descodificada;
- A instrução é executada.

Para o CPU saber que pedaço de memória deve executar, é utilizado um contador para indicar instrução que está a realizar. Esse endereço fornece a localização de memória que contém a próxima instrução a ser executada [29].

Através da análise de um pedaço de *assembly*, é possível observar que grande parte das interacções dos programas são movimentações de pedaços de informação e operações aritméticas.

```
MOV   EAX, 0FFFFh
ADD   AX, 1
ADD   AL, 1
MOV   EBX, 0FFh
AND   BL, 55h
AND   BL, 0AAh
INC   BL
INC   BL
DEC   BL
DEC   BL
```

Figura 2.8: Parte de um código em *assembly*

A arquitetura de um CPU divide-se em quatro partes; Controlo, *Arithmetic Logic Unit (ALU)*, cache e *Dynamic Random-Access Memory (DRAM)*.

O parte do controlo destina-se a coordenar as operações do computador. Indica à memória, ALU e dispositivos de entrada e saída como responder às operações que foram enviadas para o CPU [30]. A maioria dos recursos de um computador são geridos pela unidade de controlo.

A ALU é um circuito digital combinacional que realiza operações aritméticas. As entradas de uma ALU são os dados a serem operados e um código que indica o tipo de operação a ser realizada, a saída é o resultado da operação.

As ALU pode executar dois tipos de operações:

- Operações simples -> Operações aritméticas, operações lógicas (AND, NOT, OR, XOR) e operações de deslocamento de *bits*;
- Operações complexas -> Calcula qualquer operação.

A cache é uma memória de acesso rápido. Esta memória tem pouco espaço, porém é muito rápida. Armazena informações que são utilizadas com muita frequência pelo CPU. Quando o CPU necessita de procurar alguma informação, a mesma é procurada primeiro na cache e só depois na memória principal.

Por fim, a DRAM é um tipo de memória de acesso aleatório que, normalmente, é usada para os dados ou programas necessários para o funcionamento do CPU. O acesso aleatório permite que que o CPU aceda diretamente a qualquer parte da memória, sem ter de proceder sequencialmente a partir do ponto inicial [31].

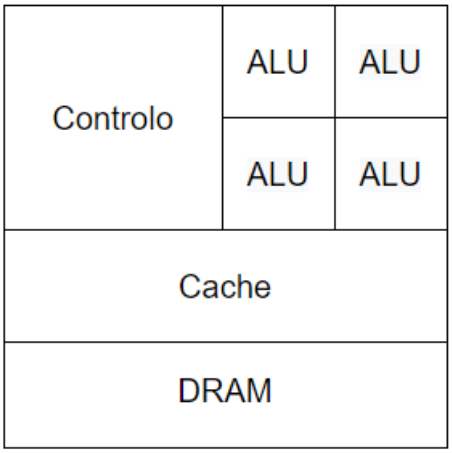


Figura 2.9: Arquitetura de um CPU

O ponto forte dos CPUs é executar algumas operações complexas com eficiência, enquanto que para o *machine learning* é o desafio oposto. A maior parte dos cálculos no processo de treino são tarefas simples, mas existem "toneladas" de cálculos , o que leva o CPU a ficar sobrecarregado, não pela complexidade dos cálculos, mas sim pela quantidade de cálculos existentes. Portanto, apesar dos processadores modernos terem vários *cores*, em que cada um é capaz de executar várias *threads*, unicamente o CPU não funciona muito bem para *machine learning*. Isto deve-se sobretudo ao número insuficiente de tarefas que podem ser executadas em simultâneo no CPU [32], no entanto é importante para cálculos complexos.

2.3.1.2 GPU

O GPU tal como o CPU é usado para processar informação, com a diferença que o GPU pode executar múltiplas instruções ao mesmo tempo, através da utilização da paralelização. O GPU é normalmente construído com vários *cores* mas muitos menos poderosos que os *cores* dos CPUs.

As diferenças entre a GPU e o CPU, é como comparar o CPU como uma sala com 4 matemáticos que são capazes de resolver qualquer problema aritmético e que a GPU consiste numa sala com 576 alunos do ensino secundário. Caso sejam dadas milhares de operações aritméticas simples a cada uma das salas, apesar dos matemáticos facilmente executarem as operações, não conseguem dar conta de todas as operações no mesmo tempo de execução da sala com os 576 alunos do secundário. Algo semelhante acontece num computador quando este necessita de processar grandes quantidades de informação.

A arquitetura de uma GPU é dividida pelo mesmo tipo de componentes que um CPU, só que organizados numa estrutura diferente.

Uma GPU está estruturada em multi-processadores, cada um dos multiprocessadores tem uma unidade de controlo, cache e múltiplas ALUs, orientadas para operações simples.

Uma outra diferença é a GPU ter memória RAM dedicada enquanto o CPU necessita de partilhar a RAM com o resto do sistema.

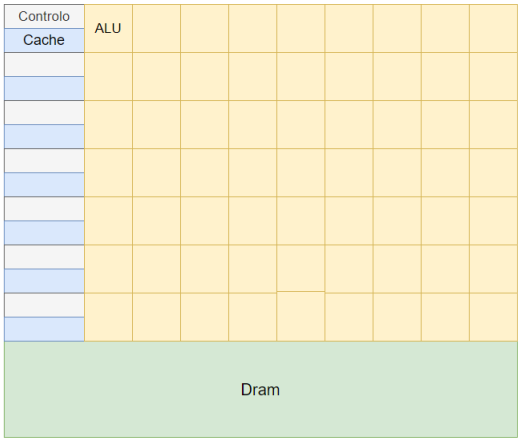


Figura 2.10: Arquitetura de uma GPU

As GPUs foram inicialmente construídas para realizar o processamento de computação gráfica, portanto, são projetadas para computação paralela massiva [32]. No entanto, mais recentemente, têm sido utilizadas para treinar redes neurais, pois estas requerem um grande número de cálculos [33], sobretudo multiplicações matriciais.

A razão da GPU ser um bom dispositivo para redes neurais, deve à sua estrutura em multi-processadores, uma vez que a GPU é composta por muito mais ALUs que os CPUs, é natural que a GPU consiga executar mais cálculos por segundo.

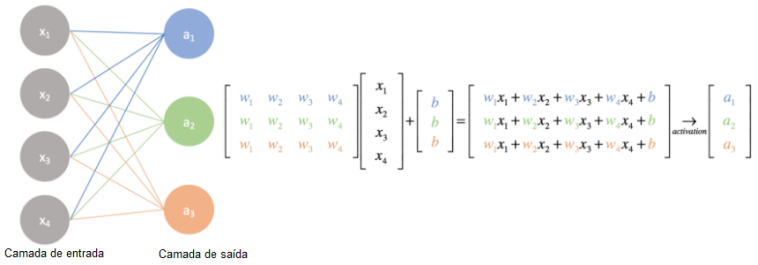


Figura 2.11: Simples rede neural [3]

2.3.1.3 NPU

NPU é um circuito integrado para uma aplicação específica com uma alta eficiência energética, são construídos para acelerar as tarefas de inteligência artificial, como; classificação de imagens, traduções, detecções de objetos e vários modelos preditivos [34]. Para a execução dessas tarefas, a NPU utiliza várias técnicas de processamento paralelo [35].

A estrutura de uma NPU não é muito diferente dos processadores *multicore*. A arquitetura divide-se em 5 partes; cache, unidade de controlo, DRAM, unidade de multiplicação de matriz e acumulador. A principal diferença entre os processadores e a NPU está na unidade que executa o processamento de dados, ou seja, a unidade de multiplicação de matriz e o acumulador .

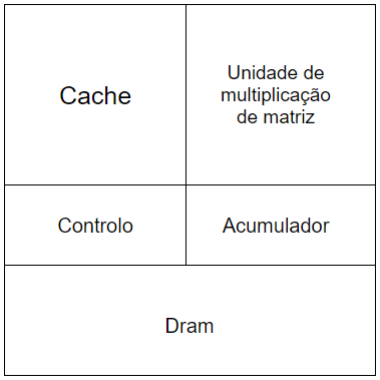


Figura 2.12: Arquitetura de uma NPU

A unidade de multiplicação de matriz e o acumulador utilizam o conceito de matriz sistólica.

A matriz sistólica é uma rede homogénea de unidade de processamento de dados conectados, chamados células ou nós. Cada nó calcula independentemente um resultado parcial como uma função dos dados recebidos dos vizinhos acima, o resultado é armazenado dentro do nó e é transmitido para o vizinho de baixo [36].

Na figura 2.13 está um exemplo de cálculo, utilizando a matriz sistólica. As matrizes A e B são carregadas para a memória, onde os dados vão ser multiplicados e somados, cada vez que uma multiplicação é executada, os seu dados são passados para o próximo nó. A saída de dados vai ser a soma de todas as multiplicações resultantes dos dados de entrada.

Durante este processo de cálculo massivo e passagem de dados, não é necessário o acesso à memória, sendo esta a razão da alta velocidade de execução de cálculos de uma NPU.

Tendo duas matrizes A*B, em que A = $\begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix}$ e B = $\begin{bmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \\ b41 & b42 & b43 & b44 \end{bmatrix}$

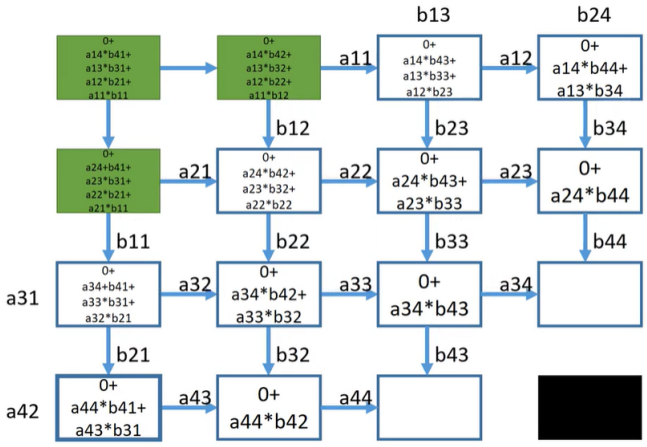


Figura 2.13: Exemplo de cálculo através da matriz sistólica [4]

NPUs não podem executar o mesmo tipo de tarefas que um CPU pode, são dispositivos que foram criados para uma aplicação específica, realizar milhares de multiplicação e adições necessárias para redes neurais [37].

As NPU podem ser classificadas em dois tipos (sendo que certas NPU podem executar as duas tarefas):

- Treino - São projetadas para treinar novos modelos;
- Inferência - Criadas para acelerar a inferência de modelos já existentes.

Dentro das NPUs, existe a *Tensor Processing Unit (TPU)* que foi desenvolvido pela Google especialmente para *machine learning* com redes neurais. A TPU foi desenhada para a *framework* de *software* TensorFlow, uma biblioteca usada para aplicações de *machine learning*. Uma única TPU pode processar mais de 100 milhões de fotos em um dia [38]

Aplicações:

- Ideal para modelos com grande volume de dados;
- Para modelos que tenham um longo tempo de treino;
- Modelos dominados por cálculos matriciais.

2.3.1.4 FPGA

A FPGA é um circuito integrado reprogramável que contêm uma matriz de blocos lógicos. São dispositivos com uma alta flexibilidade, velocidade e projetadas para execuções de tarefas paralelas.

Ao contrários dos CPUs que executam tarefas de forma sequencial, as FPGAs funcionam de através de tarefas paralelas, sem que diferentes operações necessitem de competir pelos mesmos recursos [39].

A arquitetura de uma FPGA é baseada na memória SRAM (uma memória volátil). Esse tipo de arquitetura torna possível que qualquer *hardware* digital possa ser implementado na FPGA [5].

Os principais elementos de uma FPGA com *Configurable Logic Block (CLB)* estão apresentados na figura 2.14.

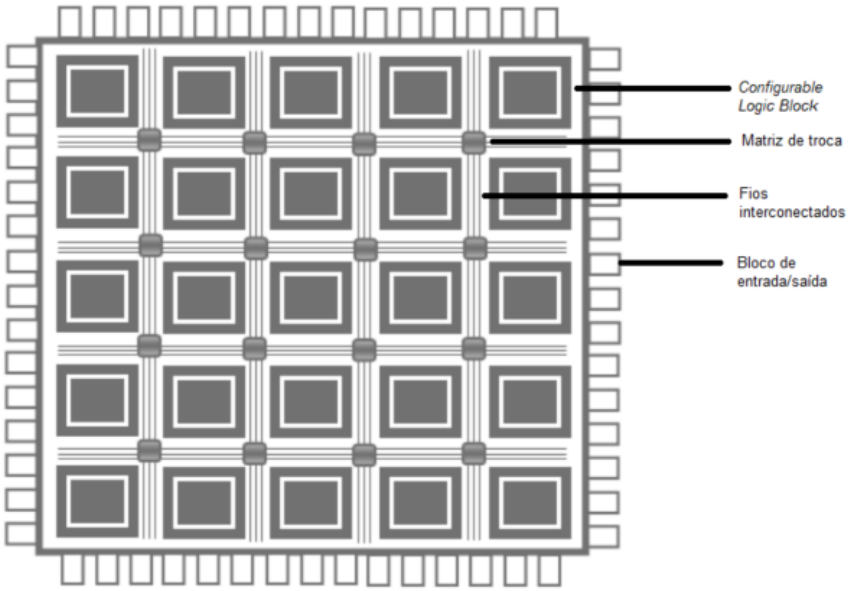


Figura 2.14: Arquitetura de uma FPGA [5]

O CLB consiste num bloco dividido em fatias, o número de fatias depende da família da FPGA, cada fatia está dividida em dois elementos lógicos [40].

Os elementos lógicos do CLB são compostos principalmente por três componentes; os flip-flops, tabelas de consulta e multiplexadores.

- Flip-flps -> O flip flop é o menor recurso de armazenamentos da FPGA. Cada flip-flop em um CLB é um registo binário usado para guardar estados lógicos entre ciclos do *clock* na FPGA;
- Tabelas de consulta -> Uma colecção de portas conectadas ao FPGA. A tabela de consulta armazena uma lista predefinida de saídas para cada combinação de entradas;
- Multiplexadores -> Um circuito que seleciona duas ou mais entrada e retorna a entrada desejada.

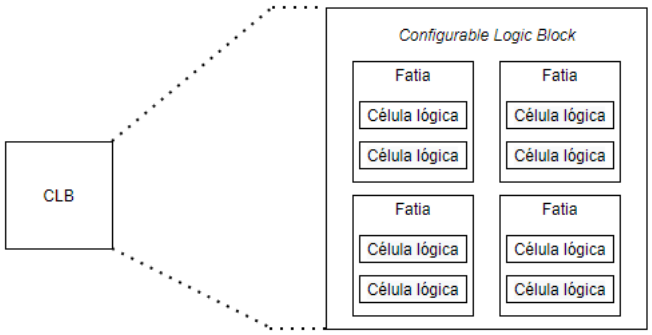


Figura 2.15: CLB com 4 fatias

A matriz de troca realiza a lógica de interconexão da FPGA configurando os fios interconectados [41]. Portanto a matriz de troca permite a construção de pontes específicas entres os fios.

Os Blocos de entrada/saída implementam as funções de entrada e saída de uma FPGA.

2.3.1.5 Conclusões

Analisando as três arquiteturas para as multiplicações de matrizes, é possível concluir que o CPU lida com a multiplicação de matrizes de uma através de operações escalares. A GPU trabalha com a multiplicação de matrizes na forma de um vetor e por fim a NPU lida com isso na forma de tensores.

Tendo em conta a forma como os três sistemas lidam com cálculos matriciais, a NPU parece ser o sistema mais rápido para *machine learning*, seguido da GPU e por fim o CPU.

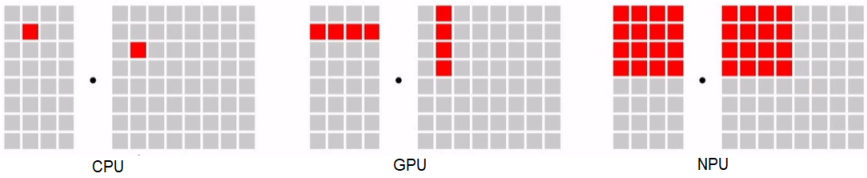


Figura 2.16: Cálculo de matrizes nas diferentes arquiteturas [6]

A FPGA não foi considerada como uma arquitetura a ser testada, no entanto a estrutura privilegia o processamento de tarefas de forma paralela, sendo algo que beneficia o *machine learning*.

2.3.2 Sistemas de visão embebida

Nos últimos anos, a miniaturização tem afetado diversas áreas, como computadores, telemóveis e electrónicos no geral, os sistemas de visão artificial também seguiram essa tendência. Um sistema clássico de visão artificial consiste numa câmara industrial e um computador, como está representado na figura 2.17. No começo da utilização da visão artificial, os sistemas eram ainda maiores e mais caros.

Figura 2.17: Sistema clássico de visão artificial [7]

Ao longo do tempo, os sistemas foram ficando mais pequenos e poderosos, os computadores foram substituídos por placas de processamento e as câmaras industriais foram trocadas por câmaras de nível de placa. Estes dois sistemas permitiram criar um sistema de visão compacto para uma aplicação específica, ou seja, um sistema de visão embebida, representado na figura 2.18, sendo que este sistema é posteriormente conectado a um sistema maior [42].

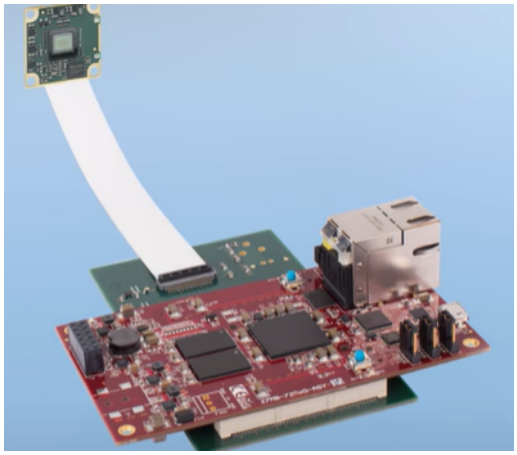


Figura 2.18: Sistemas de visão embebida [7]

As vantagens destes sistemas são:

- Tamanho reduzido;
- Peso reduzido;
- Baixo custo;
- Baixo consumo de energia.

2.3.3 Sistemas embebidos que suportam Machine Learning

Um sistema embebido que suporte aplicações em visão computacional, necessita de no mínimo conseguir correr uma versão reduzida de um sistema operativo e as bibliotecas para o *machine learning*.

Neste secção irão ser apresentadas algumas opções de *hardware* que suportam tais requisitos.

2.3.3.1 NVIDIA Jetson

Os sistemas NVIDIA Jetson são placas de computação embebidas, que incorporam um processador com uma arquitetura *Advanced RISC Machine (ARM)*. São sistemas de baixo consumo energético e foram desenvolvidas para aplicações em *machine lerarning*. Existem diferentes modelos desta placa, mas para o projeto em questão a Jetson Nano Developer kit é o suficiente, pois já suporta *machine learning*.

Características:

- CPU Quad-Core ARM Cortex;
- GPU NVIDIA Maxwell w/ 128 CUDA *cores*;
- 4 *Gigabyte (GB)* de *Random Access Memory (RAM)*;
- 40 pinos *General Purpose Input/Output (GPIO)*;
- Sistemas operativos: CUDA, Linux para Tegra e *Real Time Operating System (RTOS)*.

2.3.3.2 Coral Dev Board

A Coral Dev Board é uma placa de computação embebida desenvolvida pela *Google* que promete uma alta *performance* em *machine learning*, com um baixo custo energético.

Características:

- CPU Quad-Core ARM Cortex;
- GPU integrado GC7000 Lite Graphics;
- NPU Google Edge TPU coprocessor;
- 1, 2 ou 4GB de RAM;
- 40 pinos GPIO;
- Sistemas operativos: derivado de Debian.

2.3.3.3 RaspBerry Pi 4B

A Rapberry PI 4B é um modelo recente da gama Rapberry, de todos os sistemas enunciados, é o mais conhecido.

Características:

- CPU Quad-Core ARM Cortex;
- GPU integrada Broadcom VideoCore VI;
- 1 ou 4GB de RAM;
- 40 pinos GPIO;
- Sistemas operativos: Arch Linux, Fedora, Raspbian, entre outros.

2.3.3.4 Apalis iMX8

A Apalis iMX8 é um modelo da TORADEX construída para aplicações em visão computacional.

Características:

- CPU 8QuadMax;
- Dual GPU GC7000 3D;
- 2 ou 4GB de RAM;
- Sistemas operativos: Linux, FreeRTROS, Android e QNX.

2.3.3.5 Rock Pi N10

A Rock Pi N10 é um placa criada para aplicações de inteligência artificial e *deep learning*.

Características:

- CPU *six-core*;
- GPU Mali T860MP4 GPU;
- NPU;
- 4, 6 ou 8GB de RAM;
- 40 pinos GPIO;
- Sistemas operativos: Debian ou android.

2.3.3.6 Conclusões

De todas as placas apresentadas, as mais promissoras são a Jetson Nano Developer kit, Coral Dev Board e RaspBerry Pi 4B, pois é possível fazer uma comparação entre 3 placas com pontos fortes diferentes.

A Jetson Nano com o seu GPU NVIDIA que tem suporte para múltiplas bibliotecas de *machine learning*, a Coral Dev Board com o TPU que promete um alto número de operações por segundo com um baixo consumo energético e por fim, a RaspBerry Pi que das três placas é a que tem o CPU mais recente.

Um outro ponto que torna estas placas uma opção mais relevante, é já estarem bem documentadas para *machine learning*, o que facilita na fase de implementação.

Tabela 2.1: Pontos principais de cada placa

Hardware	CPU?	GPU NVIDIA?	NPU?	RAM?
Jetson Nano	sim	sim	não	4 GB
Coral Dev board	sim	não	não	1, 2 e 4 GB
RaspBerry Pi	sim (o melhor)	não	sim	1 e 4 GB

2.3.4 Diferentes tipos de programação

Neste sub secção irão ser abordados diferentes tipos de programação, mais precisamente a ordem de como as instruções são executadas, dependendo do tipo de programação.

2.3.4.1 Programação sequencial

A programação sequencial, tal como o próprio nome sugere, é um conjunto de instruções executadas em sequência num único processo (por exemplo um único *core*).

Na figura 2.19 está um pequeno exemplo de um processo sequencial. O programa começa por definir que a variável max é igual a 20, depois é realizada uma comparação com a variável n, onde é verificado se a mesma é maior que 20. Se for menor nada muda e o programa termina, se for maior, a variável max fica com o mesmo valor de n e o programa termina.

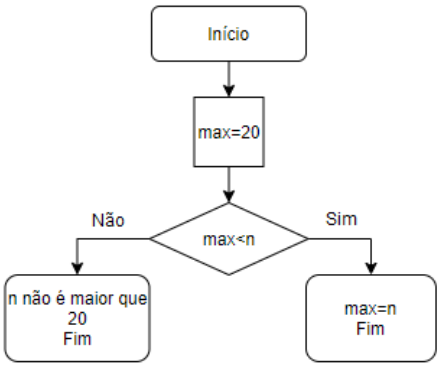


Figura 2.19: Simples exemplo de instruções sequenciais

2.3.4.2 Programação paralela

A programação é considerada paralela quando existe um conjunto de tarefas sendo executadas concorrentemente. No entanto, dentro de cada tarefa, as instruções são executadas sequencialmente [43].

A computação paralela é uma forma de usar vários recursos computacionais em simultâneo, de forma a diminuir o tempo de resolução de um problema.

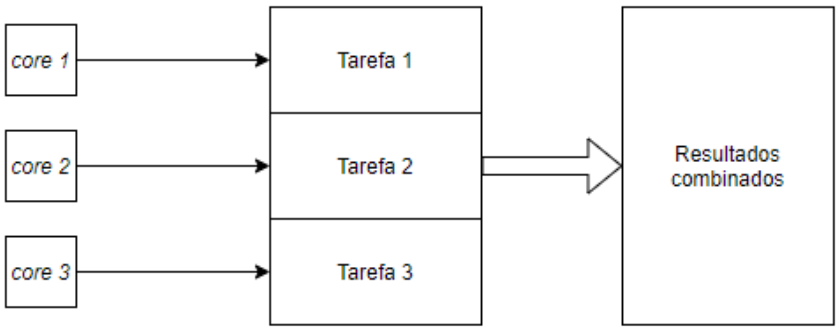


Figura 2.20: Processamento paralelo

Este tipo de programação pode ser executada de duas formas:

- As tarefas são executadas sem qualquer ordem e isso não altera o resultado final;
- As tarefas têm de ser executadas em simultâneo no processador, pois existem algum tipo de dependência entre as mesmas.

2.3.5 Bibliotecas embebidas

Nesta secção irá ser feita uma comparação entre diferentes bibliotecas para o *machine learning* em sistemas embebidos.

Na tabela 2.2 está uma comparação entre 7 bibliotecas entre diferentes requisitos.

Tabela 2.2: Comparação entre diferentes bibliotecas [10]

	Pré-requisitos	Linguagem para construção do modelo	Treino	Execução	Implementação em GPU	Instalação	Modelos Pré-treinados
tiny-dnn	Nenhum	C++	C++	C++	Não	Não	Sim
caffe	BLAS, Boost, protobuf, glog, gflags, hdf5	ficheiro de configuração	C++, Python	C++, Python	Sim	Sim	Sim
Theano	Numpy, Scipy, BLAS	Python	Python	Python	Sim	Sim	Não (1)
Tensor-Flow lite	Numpy, six, protubf	C++, Python	C++, Python	C++, Python	Sim	Sim	Sim
Mxnet	BLAS	Python, C++, R, Julia	Python, C++, R, Julia	Python, C++, R, Julia	Sim	Sim	Sim
Py-Thorch	Numpy, Scipy	Python, C++	Python, C++	Python, C++	Sim	Sim	Sim

1 (Existe uma versão não oficial)

Das sete bibliotecas, foram seleccionadas três para uma análise mais profunda.

2.3.5.1 Tensor-Flow lite

A TensorFlow lite é uma biblioteca *open source* baseada na TensorFlow, foi criada para dispositivos limitados em termos de recursos computacionais.

Esta biblioteca baseia-se em modelos pré-treinados que podem ser exportados da Tensor-Flow original, que depois podem ser adaptados para diferentes tarefas.

Principais vantagens são:

- É possível converter modelos da TensorFlow para a TensorFlow lite;
- Otimizada para sistemas com baixos recursos;
- Bem documentada e *open source*;
- Suporta a TPU de Coral Dev Board.

2.3.5.2 PyTorch

PyTorch é uma biblioteca *open source* para *machine learning* utilizada em aplicações de visão computacional.

Tem duas interfaces para desenvolvimento, Python e C++, no entanto a interface mais polida e com mais suporte é a de Python. É uma bibloteca bastante utilizada para o desenvolvimento de *software*, sendo até utilizada no *Autopilot* da tesla [44].

Principais vantagens:

- Permite paralelismo de dados declarativo, ou seja facilmente podem ser usadas várias GPUs em paralelo;
- De fácil implementação por causa da interface em Python;
- Bem documentada para um biblioteca tão recente;
- O comportamento da rede pode ser alterado programaticamente em tempo de execução.

2.3.5.3 Theano

Theano é uma biblioteca de *deep learning* criada no *Montreal Institute for Learning Algorithms (MILA)* na universidade de Montreal. Na sua essência, Theano é um compilador para expressões matemáticas em Python [45].

Esta biblioteca foi projetada especificamente para lidar com grandes redes neurais. Sendo que foi uma das primeiras do tipo (desenvolvimento iniciado em 2007), hoje em dia é considerada um padrão na indústria para a pesquisa e desenvolvimento de *deep learning* [46].

Principais vantagens:

- Bem otimizada;
- Suporte para GPUs e CPUs;
- Diferenciação Simbólica, que cria automaticamente gráficos simbólicos para gradientes de computação

2.3.5.4 Conclusões

As bibliotecas mais promissoras para o projeto são a Tensor-Flow lite, PyTorch e Theano.

A Tensor-Flow lite é uma escolha pertinente pois a Coral Dev Board é otimizada para a mesma, sendo interessante observar até que ponto existem diferenças de performance.

A PyTorch para conseguir obter o máximo desempenho, necessita de uma GPU da NVIDIA logo será relevante observar o quanto a GPU da NVIDIA faz a diferença.

A Theano irá ser uma boa comparação, pois como foi uma das primeiras bibliotecas de *deep learning*, será possível observar se fica atrás da Tensor-Flow lite e PyTorch, que são bibliotecas mais recentes.

2.3.6 Machine Learning com Python

Como é possível observar nas bibliotecas descritas anteriormente, o Python está muito presente em *machine learning*.

Os projetos para *machine learning* devem utilizar linguagens estáveis, flexíveis e com várias ferramentas disponíveis e o Python é capaz de oferecer tudo isso [47].

É simples e consistente, pois oferece um código conciso e legível, o que permite aos *developers* focarem-se na resolução de um problema de *machine learning*, em vez de se preocuparem com as nuances técnicas da linguagem.

Oferece uma vasta selecção de bibliotecas e *frameworks*, que facilitam e diminuem o tempo de implementação de diferentes algoritmos. Pois se o *developer* não tiver de se preocupar em codificar tarefas comuns como a análise de dados de uso geral, ganha muito tempo e garante menos erros por essas ferramentas já estarem devidamente testadas.

Tem independência das diferentes plataformas pois é suportado pelos principais sistemas operativos (Linux, Windows e macOS), o que garante uma fácil distribuição entre as diferentes plataformas.

Para finalizar, o Python é uma das linguagens mais populares, como tal existe uma grande comunidade para suporte.

Capítulo 3

Arquitetura de sistema

A arquitetura de uma rede neural tem uma grande influência na sua precisão e velocidade de treino. O número de camadas e a forma como estão estão organizadas, mudam o resultados obtidos. Como tal, é importante testar diferentes estruturas para perceber as que melhor se adequam.

Neste capítulo irão ser apresentadas as diferentes estruturas de CNN utilizadas para teste dos sistemas embebidos.

3.1 Modelos de CNN

O problema abordado com o *machine learning* foi a classificação de imagens. Como tal, existem algumas especificações a ser definidas:

- **Tarefa** : classificação de imagens;
- **Tipo de *machine learning*** : *supervised*;
- **Tipo de rede neural** : CNN;
- **Linguagem de programação** : Python;
- **Bibliotecas** : PyTorch, TensorFlow e Theano;
- **Arquitetura das Redes** : As três redes para esta dissertação são simples, não tendo em vista a otimização.

Tabela 3.1: Rede 1

Número de camadas	17
Distribuição de Camadas	3 camadas convulsionais+ 3 camadas de *pooling*+ 5 camadas $ReLU$+ 3 camadas de normalização+ 3 camadas totalmente conectadas+ 2 camadas de *dropout*

Tabela 3.2: Rede 2

Número de camadas	13
Distribuição de Camadas	2 camadas convulsionais+ 2 camadas de *pooling*+ 4 camadas *ReLU*+ 2 camadas de normalização+ 3 camadas totalmente conectadas+ 2 camadas de *dropout*

Tabela 3.3: Rede 3

Número de camadas	12
Distribuição de Camadas	2 camadas convulsionais+ 2 camadas de *pooling*+ 3 camadas *ReLU*+ 2 camadas de normalização+ 2 camadas totalmente conectadas+ 1 camadas de *dropout*

Capítulo 4

Implementação

Todo o trabalho realizado na tese tem como objetivo responder à seguinte pergunta:

"Qual o melhor *hardware* para *machine learning* utilizando sistemas com baixos recursos computacionais?"

Como tal, este capítulo tem como principal objetivo apresentadar a implementação das CNNs utilizando as diferentes bibliotecas referidas referidas na secção 2.3.5 .

Irá ser apresentado o *dataset* utilizado e detalhar todo o processo desde a configuração do sistema operativo para cada sistema embebido, bem como detalhar o código desenvolvido.

4.1 Dataset

Inicialmente para a criação do *datasets* foi criado um *script* em *Python*.

O *script* é divido em 4 partes.

Em primeiro, é necessário seleccionar o tipo de imagens que vão ser guardadas.

De seguida, procede-se ao *download* das imagens, em que basicamente é realizado *web scraping*, utilizando uma biblioteca chamada Selenium, que permite automatizar diferentes tipos de tarefas no *browser*.

As imagens que irão servir para o treino e teste da rede neural, têm que ter um tamanho definido (altura e largura). Como tal, é necessário ter uma forma de redimensionar as imagens para que os tamanhos sejam todos os mesmos.

Por fim, é criado um ficheiro *Comma-Separated Values (CSV)* onde é armazenado o nome da imagem e a sua classificação, num valor inteiro. Por exemplo, se for um gato terá um valor correspondente de zero e se for um cão terá um valor de um.

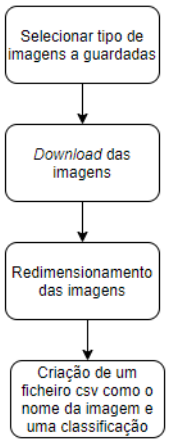


Figura 4.1: Criação do *dataset*

No final, acabou por ser utilizado um *dataset* já existente, CIFAR-10, pela melhor fiabilidade das imagens, pois algumas das imagens recolhidas com o *script* não eram referentes ao tema pesquisado.

O *dataset* CIFAR10 consiste em sessenta mil imagens 32X32, divididas por 10 classes, cada classe com dez mil imagens. Das sessenta mil imagens, cinquenta são para a fase de treino e dez mil para testes do modelo [8]. No total as sessenta mil imagens ocupam um espaço em memória de 163 *Megabyte (MB)* para a versão de python.

As dez classes do *dataset* estão divididas em: aviões, automóveis, pássaros, gatos, veados, cães, sapos, cavalos, navios e camiões.

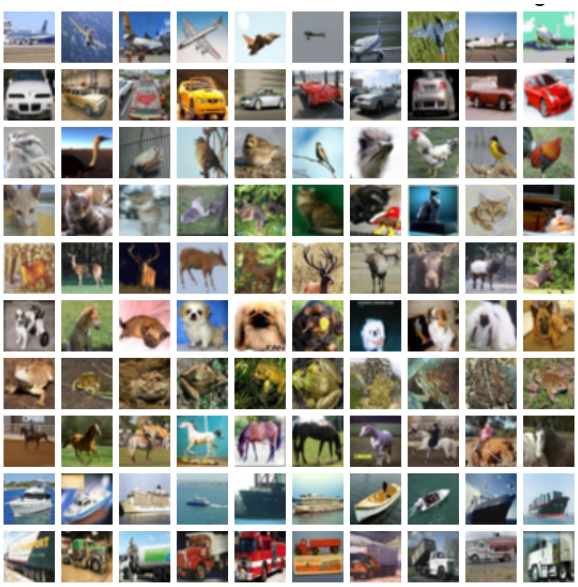


Figura 4.2: Exemplos de imagens do *dataset* CIFAR0-10 [8]

4.2 Instalação de Sistemas operativos

Para o treino de uma CNN, os diferentes sistemas embebidos, referidos nos fundamentos, necessitam de um sistema operativo que pelo menos suporte o *Python*.

Nesta secção irão ser apresentados os sistemas operativos instalados em cada um dos sistemas, assim como os pacotes necessários para executar cada uma das bibliotecas.

4.2.1 Jetson Nano (4GB)

O sistema operativo instalado na Jetson Nano é o Tegra linux, é uma distribuição da Nvidia para processadores da serie Tegra. A imagem para está disponivel no *site* da Nvidia (https://developer.nvidia.com).

Para instalar o linux na Jetson Nano, em primeiro é necessário fazer o *flash* do cartão de memória utilizando o programa balenaEtcher.

Quando o *flash* do cartão de memória estiver finalizado, é só inserir o cartão de memória na Jetson Nano e ligar a mesma à corrente.

A primeira inicialização do linux é muito semelhante à de uma distribuição normal, onde é necessário fazer a configuração do utilizador, *password*, fuso horário e o *layout* do teclado.

Depois da configuração do linux é necessário instalar os pacotes para a utilização da bibliotecas.

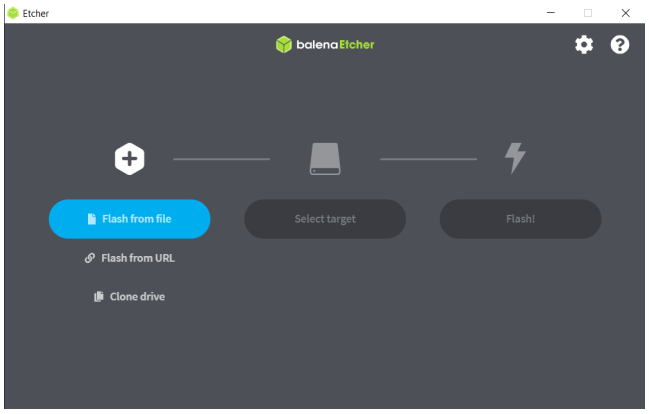


Figura 4.3: Programa para gerar imagem para a Jetson Nano

Lista de comandos para instalar as bibliotecas e suas dependências:

- sudo apt-get install python3-pip libopenblas-dev libopenmpi-dev libomp-dev
- sudo -H pip3 install future
- sudo -H pip3 install –upgrade setuptools
- sudo -H pip3 install Cython
- sudo -H pip3 install torch-1.7.0a0+56b43f4-cp36-cp36m-linux_aarch64.whl

- sudo -H pip3 install torchvision-0.8.0a0+8fb5838-cp36-cp36m-linux_aarch64.whl
- sudo apt-get install libjpeg-dev zlib1g-dev libpython3-dev
- sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
- sudo apt install python3-keras
- sudo apt install python3-theano
- sudo pip3 install -U pip testresources setuptools numpy==1.16.1 future==0.17.1 mock==3.0.5 h5py==2.9.0 keras_preprocessing==1.0.5 keras_applications==1.0.8 gast==0.2.2 futures protobuf pybind11
- sudo pip3 install –pre –extra-index-url https://developer.download.nvidia.com /compute/redist/jp/v45 'tensorflow<2'

4.2.2 Coral Dev board (1 GB)

O sistema operativo instalado na coral DEV board é um derivado do Debian Linux. A imagens foi descarregada do *site* coral.ai.

Para instalar a imagem na coral, é necessário fazer o *flash* do cartão de memória com o programa balenaEtcher. Depois da imagem estar *flashed* no cartão de memória, com a *board* desligada, é necessário mudar a posição dos interruptores, como é mostrado na figura 4.4.

Com os interruptores na posição certa é só adicionar o cartão de memória à *board* e ligar a mesma à corrente.

A *board* vai demorar entre 5 a 10 minutos a ler e instalar a imagem do cartão de memória.

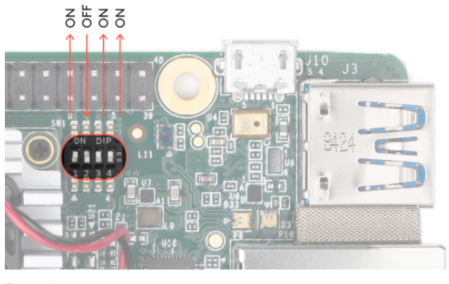


Figura 4.4: Posição dos interruptores para a placa inicializar com o cartão de memória [9]

Quando o processo de instalação do sistema operativo estiver finalizado, é necessário desligar a *board* da corrente, remover o cartão de memória e alterar o modo de inicialização, mudando a posição dos interruptores, como demonstrado na figura 4.5.

Por fim é só ligar a *board* à corrente para iniciar com o sistema operativo.

Com o sistema operativo configurado, o próximo passo é a instalação de vários pacotes para habilitar as diferentes bibliotecas.

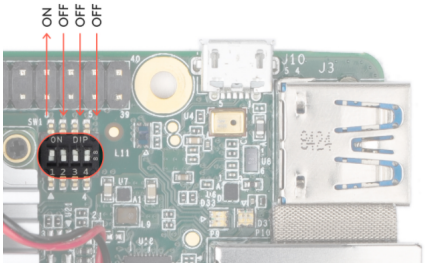


Figura 4.5: Posição dos interruptores para a placa inicializar com o sistema operativo [9]

É necessário o *download* de alguns ficheiros para a instalação do PyTorch (https://mathinf.eu/pytorch/arm64/).

Lista de comandos para instalar as bibliotecas e suas dependências:

- sudo apt install libopenblas-dev libblas-dev m4 cmake cython python3-dev python3-yaml python3-setuptools python3-wheel python3-pillow python3-numpy
- sudo apt-get install libopenblas-base libopenmpi-dev
- sudo apt-get install libatlas-base-dev
- sudo apt install python3-h5py
- sudo apt install python3-keras
- sudo apt install python3-theano
- pip3 install torch-1.7.0a0-cp37-cp37m-linux_aarch64.whl
- pip3 install torchvision-0.8.0a0+45f960c-cp37-cp37m-linux_aarch64.whl

Por incompatibilidades de alguns pacotes, não foi possível instalar a TensorFlow. No entanto, a TensorFlow lite já vem instalada de origem no sistema operativo.

4.2.3 RaspBerry Pi 4B (4GB)

O sistema operativo instalado na RaspBerry é o RaspBerry Pi OS 32 bits. Para gerar a imagem foi utilizado o programa RaspBerry Pi Imager, em que só é necessário seleccionar o sistema operativo e seleccionar o cartão de memória.

Com a imagem gerada, a instalação do sistema operativo na RaspBerry é simples, apenas é necessário inserir o cartão à RaspBerry e ligar à corrente, sistema operativo vai inicializar.

No fim da instalação do sistema operativo, são instalados os pacotes necessários para utilizar as bibliotecas para *machine learning* .

É necessário transferir alguns ficheiros de instalação do Pytorch (https://github.com/sungjuGit/PyTorch-and-Vision-for-RaspBerry-Pi-4B).

Figura 4.6: Programa para gerar a imagem para o RaspBerry

Lista de comandos para instalar as bibliotecas e suas dependências:

- sudo apt install libopenblas-dev libblas-dev m4 cmake cython python3-dev python3-yaml python3-setuptools python3-wheel python3-pillow python3-numpy
- sudo apt-get install libatlas-base-dev
- pip3 install torch-1.7.0a0+f43194e-cp37-cp37m-linux_armv7l.whl
- pip3 install torchvision-0.8.0a0+9cdc814-cp37-cp37m-linux_armv7l.whl
- pip3 install tensorflow
- pip3 install keras
- pip3 install jupyter
- pip3 install theano

4.3 Algoritmos para teste

Nesta secção, irá ser abordada a forma como os algoritmos foram criados, utilizando as diferentes bibliotecas, escolhidas nos fundamentos (Pytorch, TensoFlow, Theano).

Irá ser somente apresentada a rede 1, citada no capítulo 3, para cada uma das bibliotecas, visto que os restantes modelos utilizam as mesmas funções, exceptuando o facto da estrutura da rede ser diferente.

4.3.1 CNN Pytorch

Nesta subseção é explicado passo a passo como a rede neural é criada utilizando a Pytorch.

Existem duas variáveis de entrada (`in_channels, num_classes`) no construtor da classe CNN. A `in_channels` tem como objetivo indicar a gama de cores das imagens, como o valor é 3, indica que é uma imagem RGB. A `num_classes` corresponde ao número de saídas da rede, ou seja, se a rede for treinada para 10 tipos de objetos diferentes, este valor tem de ser igual a 10.

Ainda dentro do constructor são criadas as camadas de convolução (`conv1, conv2, conv3`), *polling* (`pool, pool2, pool3`), camadas totalmente conectadas (`fc1, fc2, out`), normalização (`bn1, bn2, bn3`) e o *dropout* (`drop`).

Mudando agora para a função `forward`, ela recebe de parâmetro de entrada a variável `x` que corresponde à imagem a ser processada. De seguida está a organização da rede, em cada camada de convolução e nas camadas totalmente conectadas é aplicado um ReLU, importante para introduzir a não linearidade na rede. É interessante notar que entre cada camada convolucional e camadas totalmente conectadas, o número de saídas da camada anterior é igual ao número de entradas da camada logo a seguir.

Isto aplica-se em todas as camadas, excepto na transição das camadas de convolução para as camadas totalmente conectadas, onde para além desse valor, é necessário multiplicar a largura e altura da imagem naquela fase do processamento.

Para calcular a largura e altura da imagem em cada fase de processamento é utilizada a seguinte fórmula:

$$W' = \frac{W - F + 2P}{S} + 1 \tag{4.1}$$

Onde:

- W -> Largura/altura de uma imagem quadrada.
- F (*Kernel*) -> O filtro usado para extrair os recursos das imagens.
- P (*Padding*) -> Ajusta a imagem caso o filtro não se encaixe perfeitamente com a imagem.
- S (*Stride*) -> O número de pixeis deslocados na matriz de entrada.

Entre as camadas de convolução e as camadas totalmente conectadas existem duas funções: `BatchNorm2d` e `Dropout`.

`BatchNorm2d` tem como finalidade tornar a rede mais rápida e estável, normalizando a camada de entrada através recentralização e redimensionamento.

O `Dropout` é uma técnica de regularização que tem como ideia desativar alguns neurónios da rede, de forma a evitar o problema do *overfitting*, ou seja, evitar que a rede se torne muito específica, para aquele conjunto de imagens.

Depois de todas as fases estarem completas é retornada uma nova matriz.

```
class CNN(nn.Module):
    def __init__(self, in_channels=3, num_classes=10):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=8,
                               kernel_size=(3,3), stride=(1,1), padding=(1,1))
        self.bn1 = nn.BatchNorm2d(num_features=8)
        self.pool = nn.MaxPool2d(kernel_size=(2,2), stride=(2,2))

        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,
                               kernel_size=(3,3), stride=(1,1), padding=(1,1))
        self.bn2 = nn.BatchNorm2d(num_features=16)
        self.pool2 = nn.MaxPool2d(kernel_size=(3,3), stride=(6,6))
```

```
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=144,
                    kernel_size=(3,3), stride=(1,1), padding=(1,1))
        self.bn3 = nn.BatchNorm2d(num_features=144)
        self.pool3 = nn.MaxPool2d(kernel_size=(3,3), stride=(6,6))

        self.fc1 = nn.Linear(in_features=144*1*1, out_features=100)
        self.fc2 = nn.Linear(in_features=100, out_features=50)
        self.out = nn.Linear(in_features=50, out_features=num_classes)
        self.drop = torch.nn.Dropout(0.2)


    def forward(self, x):
        #hidden conv layer
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(x)

        #hidden conv layer
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool2(x)

        #hidden conv layer
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.pool3(x)

        #hidden linear layers
        x = x.reshape(-1, 144*1*1)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = F.relu(self.fc2(x))
        x = self.drop(x)
        x = self.out(x)

        return x
```

Com rede definida, o próximo passo é treinar a mesma. Para a fase de treino foi criado um *script* definido pela figura 4.7.

No começo do *script* é selecionado o tipo de dispositivo onde a rede vai ser treinada. Caso a GPU esteja disponível, será na mesma, senão é escolhido o CPU.

Imediatemente a seguir, o *dataset* é carregado (CIFAR10) e as imagens são convertidas para tensores.

O próximo passo consiste na inicialização da rede, a classe acima, e o carregamento da *loss* e *optimizer*.

A *loss* é utilizada para medir o erro entre a saída da rede e o valor alvo fornecido, a função utilizada foi `CrossEntropyLoss` e é particularmente útil para conjuntos de treino desequilibrados [48].

O *optimizer* é uma função que implementa vários algoritmos de optimização no treino da rede.

O último passo é o treino da rede que está que está fraccionado em algumas etapas:

- Carregar dados para a GPU;
- *Foward propagation*;
- *Backward propagation*;
- Gradiente descendente.

A *Foward propagation* calcula o resultado de acordo com uma função e guarda os dados necessários para o cálculo do gradiente na memória.

A *backward propagation* utiliza o valor do erro, esse valor volta a ser propagado na rede neural para recalcular o peso e a tendência.

O Gradiente é utilizado para medir a mudança em todos os pesos em relação à mudança no erro. O gradiente pode ser visto como a inclinação de uma função. Quanto maior for o gradiente, maior será a sua inclinação e mais rápida será a aprendizagem do modelo, que caso seja zero, o modelo deixa de aprender. Enquanto o gradiente vai subindo a inclinação, ou seja, do fundo para o topo, o gradiente descendente faz o caminho inverso.

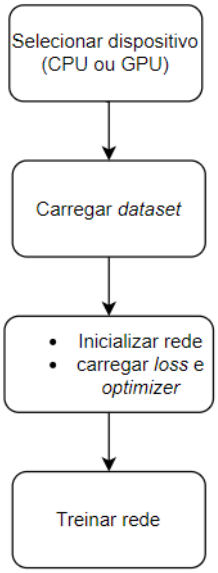


Figura 4.7: Estrutura do programa

Depois da rede estar treinada, a mesma é guardada num ficheiro .pth .

4.3.2 CNN TensorFlow e Theano

Para trabalhar com TensorFlow e Theano é utilizada uma outra biblioteca, Keras.

A Keras é uma biblioteca que foi criada para ser "amiga" do utilizador e ao mesmo tempo eficiente. Este efeito é obtido tendo uma sintaxe simples para o utilizador mas utilizando mecanismos de *backend* poderosos, onde a TensorFlow e Theano estão inseridos.

4.3.2.1 TensorFlow

O modelo criado usando a TensoFlow é semelhante ao anterior, sendo que entre cada camada é passado o mesmo número de saídas, as funções utilizadas têm o mesmo propósito e o número de camadas é o mesmo.

```
def my_model():
    inputs = keras.Input(shape=(32, 32, 3))
    x = layers.Conv2D(8, 3, padding="same", kernel_regularizer=
                      regularizers.l2(0.01),)(inputs)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.MaxPooling2D(2)(x)

    x = layers.Conv2D(16, 3, padding="same", kernel_regularizer=
                      regularizers.l2(0.01),)(x)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.MaxPooling2D(2)(x)

    x = layers.Conv2D(288, 3, padding="same", kernel_regularizer=
                      regularizers.l2(0.01),)(x)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.Flatten()(x)

    x = layers.Dense(100, activation="relu", kernel_regularizer=
                     regularizers.l2(0.01),)(x)
    x = layers.Dropout(0.2)(x)
    x = layers.Dense(50, activation="relu", kernel_regularizer=
                     regularizers.l2(0.01),)(x)
    x = layers.Dropout(0.2)(x)
    outputs = layers.Dense(10)(x)

    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

A fase de treino é também bastante semelhante, a diferença é que o modelo no fim do treino é guardado e convertido para a TensorFlow lite, para posteriormente ser utilizado nos sistemas embebidos.

Depois do modelo treinado e convertido, foi necessário criar um novo código para a inferência do modelo no TensorFlow lite.

Na figura 4.8 é apresentado a estrutura do *script* para inferência do modelo na TensorFlow lite.

Inicialmente, o dispositivo onde o modelo vai correr é seleccionado, após isso, os dados e o modelo pré treinado são carregados para o sistema. Em seguida, é necessário redimensionar os tensores para estarem de acordo com os dados de teste, é utilizada a função `resize_tensor_input` para tal.

Por fim, a rede é ativada e é verificada a precisão dos resultados.

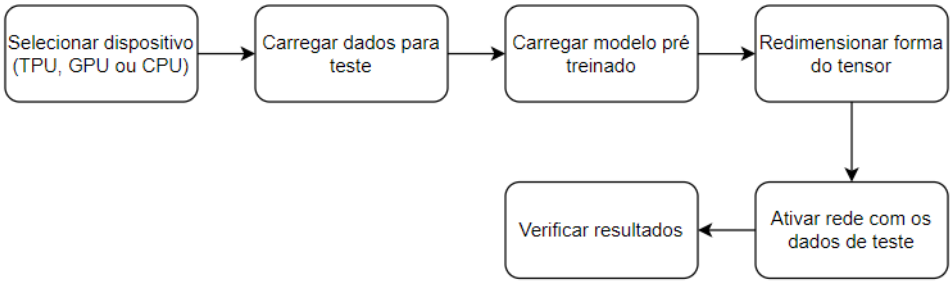


Figura 4.8: Inferência do modelo

4.3.2.2 Theano

O modelo para Theano é semelhante aos modelo anteriores, sendo especialmente semelhante ao modelo da TensorFlow, visto que ambos utilizam a Keras, só que com mecanismos *backend* diferentes.

A principal diferença visível no código em relação ao modelo da TensorFlow é o uso do comando `Sequential`. O uso deste comando não afeta em nada o resultado final, é apenas uma abstracção para executar as instruções sequencialmente sem ter que passar o estado anterior à instrução que vai ser executada.

```
model = Sequential([
        layers.Conv2D(8, 3, padding="same", input_shape=(32, 32, 3)),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPooling2D(2),

        layers.Conv2D(16, 3, padding="same"),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPooling2D(2),

        layers.Conv2D(144, 3, padding="same"),
        layers.BatchNormalization(),
        layers.Activation('relu'),
```

```
        layers.Flatten(),


        layers.Dense(100, activation="relu"),
        layers.Dropout(0.2),
        layers.Dense(50, activation="relu"),
        layers.Dropout(0.2),
        layers.Dense(10),
    ])
```

No geral, a estrutura do código é muito semelhante aos modelos anteriores. Existe um passo para seleccionar o dispositivo onde a rede vai ser treinada e executada, o *dataset* é carregado, o modelo criado é inicializado e por fim a rede é treinada.

No fim de todo este processo, o modelo treinado é guardado.

4.4 Problemas durante o desenvolvimento

O principal problema durante o desenvolvimento dos algoritmos, é o sistema onde foram desenvolvidos.

Foram codificados num computador que utiliza um processador x86 (*Complex Instruction Set Computer (CISC)*) e os sistemas embebidos utilizados para a dissertação utilizam um processador com uma arquitetura ARM (*Reduced Instruction Set Computer (RISC)*).

Os processadores ARM têm uma arquitetura mais simples, sendo de um tamanho reduzido e possuem um baixo gasto energético. Os processadores x86, geralmente têm uma arquitetura interna mais complexa, com consumo energético superior.

A diferença mais relevante entre as duas arquiteturas, é as instruções para ARM operam apenas em registos, com algumas instruções para carregar e guardar os dados de / para a memória, enquanto o x86 pode operar diretamente na memória.

O ponto problemático desta diferença, é certos pacotes utilizados durante a fase de desenvolvimento, não terem suporte para os processadores ARM dos sistemas embebidos.

Capítulo 5

Testes e validação de resultados

Neste capítulo são descritos os teste realizados, de forma a perceber qual dos sistemas obtém a melhor *performance*.

Os testes estão divididos em duas fases principais; fase de treino e fase de inferência.

Durante a fase de treino são utilizadas três redes diferentes, para verificar como a estrutura das redes afeta os resultados. O parâmetro verificado durante esta fase é o tempo de treino da rede.

Durante a fase de inferência utiliza-se os modelos já treinados, para verificar a velocidade dos modelos nas diferentes plataformas e a precisão da rede com um novo conjunto de dados nunca visto pelo modelo.

Para obter os resultados, foi utilizado cada um dos modelos citados no capítulo 3. Cada modelo foi executado em todos os sistemas embebidos utilizando as diferentes bibliotecas. Em cada rede testada, o número de *epochs* do modelo também variou.

O número de *epochs* é um parâmetro que define o número de vezes o algoritmo funcionará durante a fase de treino.

5.1 Jetson Nano

Na tabela 5.1 são apresentados os resultados obtidos na Jetson Nano, à primeira vista é possível concluir que a biblioteca que obteve melhores resultados de uma maneira geral é a PyTorch.

Também é possível observar que a o biblioteca Theano tem tempos de treino e inferência muito superiores. Isto deve-se ao não uso da GPU para executar o programa.

Na figura 5.1 está o erro que não permitiu a utilização da GPU, pois mesmo com a versão da pygpu atualizada, o erro manteve-se.

```
ERROR (theano.gpuarray): pygpu was configured but could not be imported or is to
o old (version 0.6 or higher required)
NoneType: None
Using Theano backend.
```

Figura 5.1: Erro para o uso da GPU com a Theano

Tabela 5.1: Resultados na Jetson nano

Amostra	Biblioteca	Rede	*Num Epochs*	*Batch size*	fase de treino (s)	fase de inferência (s) (10000 imagens)	Precisão (%)
1	PyTorch	1	20	32	≈ 1314.2	≈ 10.1	≈ 64.1
2	PyTorch	1	40	32	≈ 2602.2	≈ 8.8	≈ 65.5
3	PyTorch	2	20	32	≈ 1139.6	≈ 8.3	≈ 62.2
4	PyTorch	2	40	32	≈ 2286.7	≈ 9.9	≈ 64.3
5	PyTorch	3	20	32	≈ 1071.6	≈ 8.7	≈ 61.9
6	PyTorch	3	40	32	≈ 2088.5	≈ 8.6	≈ 63.1
1	TensorFlow	1	20	32	≈ 1159.7	≈ 34.8	≈ 50.9
2	TensorFlow	1	40	32	≈ 2291.9	≈ 35.6	≈ 59.6
3	TensorFlow	2	20	32	≈ 796.4	≈ 14.3	≈ 58.6
4	TensorFlow	2	40	32	≈ 1649.3	≈ 13.7	≈ 53.9
5	TensorFlow	3	20	32	≈ 674.4	≈ 13.7	≈ 60.1
6	TensorFlow	3	40	32	≈ 1305.9	≈ 15.4	≈ 61.6
1	TensorFlow lite	1	-	-	-	≈ 10.3	≈ 52.9
2	TensorFlow lite	1	-	-	-	≈ 10.5	≈ 50.4
3	TensorFlow lite	2	-	-	-	≈ 5.7	≈ 61.1
4	TensorFlow lite	2	-	-	-	≈ 5.8	≈ 55.0
5	TensorFlow lite	3	-	-	-	≈ 4.9	≈ 62.9
6	TensorFlow lite	3	-	-	-	≈ 4.9	≈ 61.3
1	Theano	1	20	32	≈ 25916.1	≈ 92.8	≈ 10.2
2	Theano	1	40	32	≈ 49720.3	≈ 94.1	≈ 9.8
3	Theano	2	20	32	≈ 10126.1	≈ 40.9	≈ 9.9
4	Theano	2	40	32	≈ 19887.9	≈ 37.9	≈ 7.1
5	Theano	3	20	32	≈ 7164.3	≈ 32.3	≈ 12.7
6	Theano	3	40	32	≈ 14281	≈ 29.6	≈ 10.3

No gráfico da figura 5.2 é exposta a comparação entre tempos de treino para as três bibliotecas.

O que salta mais à vista é a grande diferença da Theano no tempo de treino para as restantes, o motivo disto é a Theano estar a utilizar o CPU para treinar a rede, o que mostra a grande diferença que uma GPU faz para *machine learning*.

A TensorFlow e a PyTorch têm um tempo de treino mais próximos, com a TensorFlow a ter um tempo de treino inferior de 13% a 60%. É interessante reparar que quanto mais

complexa a rede, menor a diferença de tempo de treino entre a PyTorch e a TensorFlow.

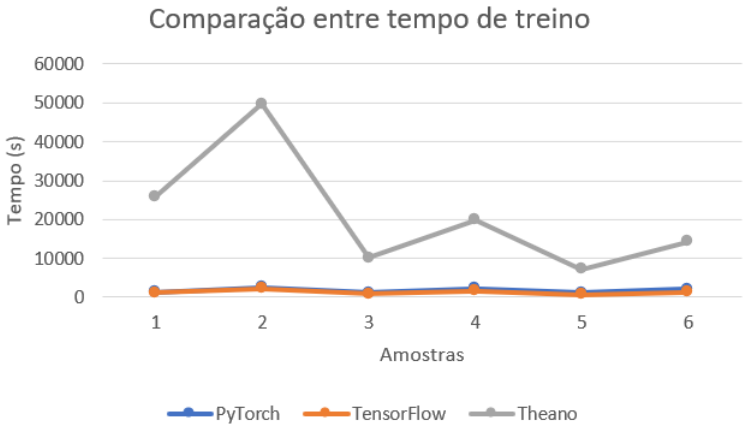


Figura 5.2: Comparação entre tempo de treino (jetson)

O gráfico da figura 5.3 mostra comparação entre os tempos de inferência.

As bibliotecas com os melhores tempos são a PyTorch e a TensorFlow lite. Com a PyTorch a ser mais rápida para modelos mais complexos e a TensorFlow lite a ser mais rápida para modelos mais simples.

A TensorFlow demora mais de o dobro para os modelos mais complexos, mas para os modelos simples tem um tempo próximo com a PyTorch e TensorFlow lite. É importante salientar que a TensorFlow lite não tem suporte para a GPU da NVIDIA, como tal foi utilizado o CPU para o processamento.

Por fim a Theano é a que tem os tempos mais elevados para todos os modelos.

Durante a fase de inferência, acontece o mesmo que durante a fase de treino com a PyTorch, quanto mais complexa a rede, melhor o seu tempo em relação às restantes bibliotecas.

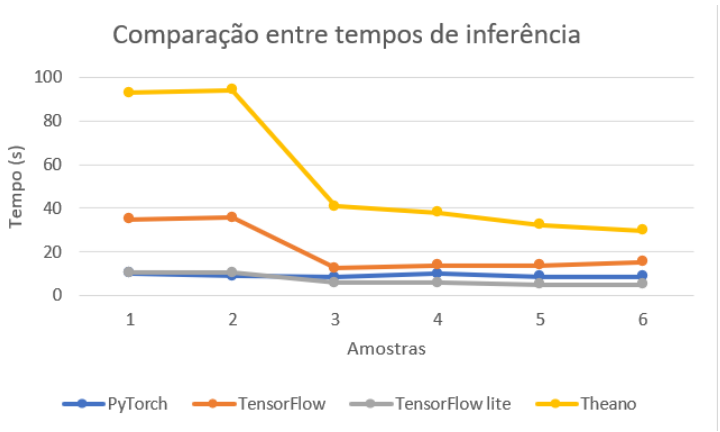


Figura 5.3: Comparação entre tempo de Inferência (Jetson)

Na figura 5.4 está representada a comparação entres as precisões dos diferentes modelos.

A Theano tem um precisão péssima, cerca de 10 %, uma precisão que não é de todo aproveitável.

As restantes bibliotecas têm valores relativamente próximos, sendo a PyTorch a que obtém os melhores valores.

A TensorFlow e a TensorFlow lite apesar de também terem bons valores, são um pouco irregulares. Os modelos mais simples ou com menor número de *epochs* conseguem uma precisão superior aos modelos mais complexos ou com um maior número de *epochs*, algo que não acontece na PyTorch.

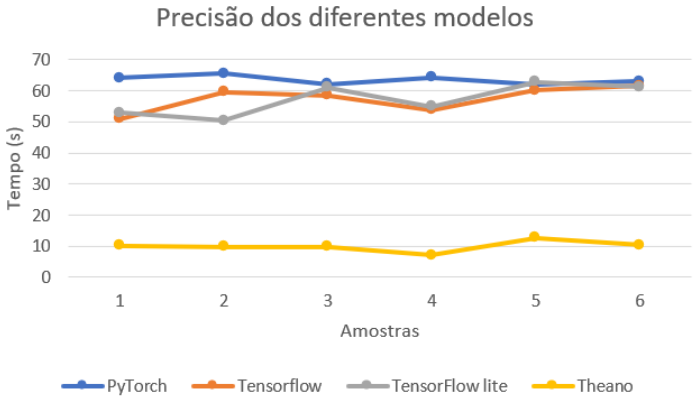


Figura 5.4: Comparação entre precisões (Jetson)

5.1.1 Conclusão

Com base nos resultados obtidos para a Jetson Nano, as bibliotecas PyTorch, TensorFlow e TensorFlow lite obtém resultados satisfatórios.

No entanto a PyTorch parece ser a que melhor se adequa ao sistema, apesar de ter tempos superiores durante a fase de treino e inferência, quanto mais complexa se torna a rede, melhores se tornam os tempos.

Sendo que a PyTorch também precisão superior e resultados regulares.

5.2 Coral Dev board

Na tabela 5.2 são apresentados os resultados da Coral Dev board.

Através de uma análise superficial dos resultados na Coral Dev board, fica claro que a mesma não é de todo ideal para treinar redes neurais e a precisão obtida fica muito a desejar, sendo aproximadamente 10%.

No entanto com a TensorFlow lite foram obtidos resultados satisfatórios, com um bom tempo de inferência e uma precisão aceitável, entre os 50 e 60%, que foram os resultados obtidos com as outras *boards*.

Tabela 5.2: Resultados na Coral board

Amostra	Biblioteca	Rede	*Num Epochs*	*Batch size*	Fase de treino (s)	Fase de inferência (s) (10000 imagens)	Precisão (%)
1	PyTorch	1	20	32	≈ 10300.5	≈ 69.9	≈ 9.5
2	PyTorch	1	40	32	≈ 20549	≈ 71.8	≈ 10.1
3	PyTorch	2	20	32	≈ 6496.6	≈ 66.6	≈ 9.7
4	PyTorch	2	40	32	≈ 12989.3	≈ 64.8	≈ 11.1
5	PyTorch	3	20	32	≈ 6347.7	≈ 64.5	≈ 10.2
6	PyTorch	3	40	32	≈ 12675.1	≈ 67.1	≈ 9.9
1	TensorFlow	1	20	32	NA	NA	NA
2	TensorFlow	1	40	32	NA	NA	NA
3	TensorFlow	2	20	32	NA	NA	NA
4	TensorFlow	2	40	32	NA	NA	NA
5	TensorFlow	3	20	32	NA	NA	NA
6	TensorFlow	3	40	32	NA	NA	NA
1	TensorFlow lite	1	20	32	-	≈ 18.4	≈ 53.2
2	TensorFlow lite	1	40	32	-	≈ 18	≈ 50.4
3	TensorFlow lite	2	20	32	-	≈ 10.2	≈ 61.2
4	TensorFlow lite	2	40	32	-	≈ 10.1	≈ 55
5	TensorFlow lite	3	20	32	-	≈ 8.7	≈ 62.9
6	TensorFlow lite	3	40	32	-	≈ 8.9	≈ 60.9
1	Theano	1	20	32	≈ 1287.5	≈ 77.8	≈ 9.3
2	Theano	1	40	32	≈ 2543.6	≈ 76.1	≈ 10
3	Theano	2	20	32	≈ 791.9	≈ 37.8	≈ 10
4	Theano	2	40	32	≈ 1461.7	≈ 36.7	≈ 9.9
5	Theano	3	20	32	≈ 673.8	≈ 39.3	≈ 10
6	Theano	3	40	32	≈ 1301.5	≈ 40.3	≈ 9.9

Na figura 5.5 é apresentado o gráfico com a comparação dos tempos de treino das bibliotecas PyTorch e Theano na Coral Dev board.

O tempo de treino com a biblioteca PyTorch é muito superior à Theano, o que pode indicar uma menor otimização da biblioteca PyTorch em CPUs ou o volume de cálculos é consideravelmente superior à Theano.

Não foi possível treinar um modelo utilizando a TensorFlow, pois durante a instalação, existiram algumas incompatibilidades de pacotes com a arquitetura do sistema operativo da Coral Dev board.

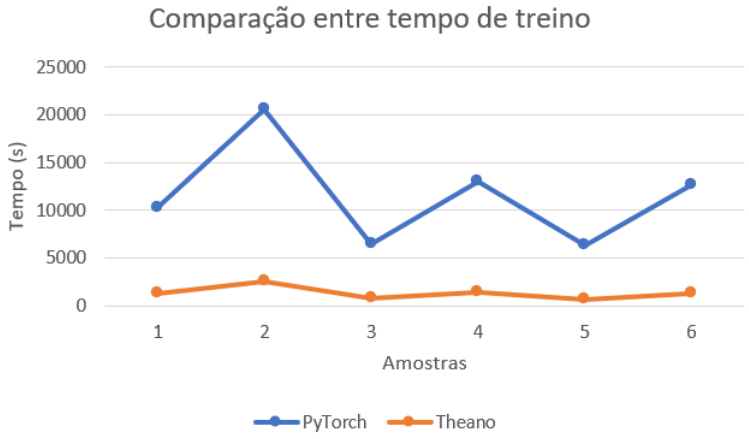


Figura 5.5: Comparação entre tempo de treino (Coral)

Durante a fase de inferência, claramente a biblioteca mais rápida é a TensorFlow lite. É um resultado que faz sentido, pois com a TensorFlow lite é possivel utilizar o TPU presente na *board*.

A PyTorch e Theano obtêm valores muito semelhantes para a primeira rede neural, com a Theano a exigir uns segundos a mais. Para a segunda e terceira rede neural a Theano reduziu consideravelmente o tempo de inferência e a PyTorch apenas teve uma ligeira descida.

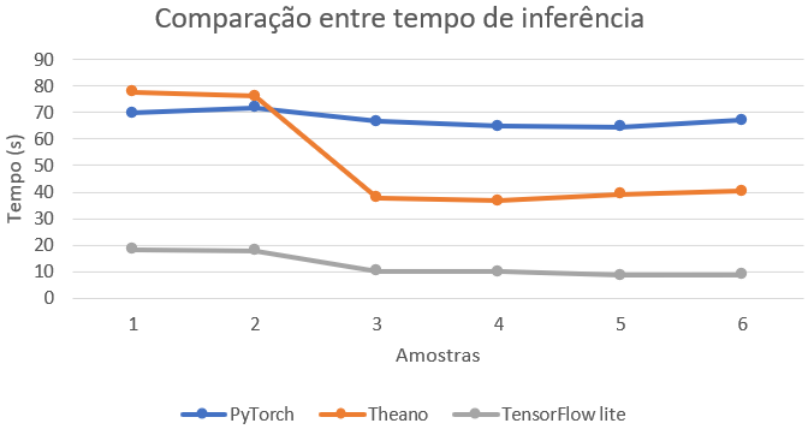


Figura 5.6: Comparação entre tempo de Inferência (Coral)

A precisão das redes é o resultado mais curioso.

As bibliotecas que foram treinadas utilizando a Coral Dev board obtiveram resultados péssimos, por volta dos 10% de precisão.

Um resultado estranho, pois nas outras *boards*, pelo menos a PyTorch teve resultados satisfatórios.

Uma possível explicação, é a memória RAM da placa, pois a mesma só tem 1 GB de RAM, enquanto as outras duas placas têm 4 GB.

A TensorFlow lite obteve resultados expectáveis.

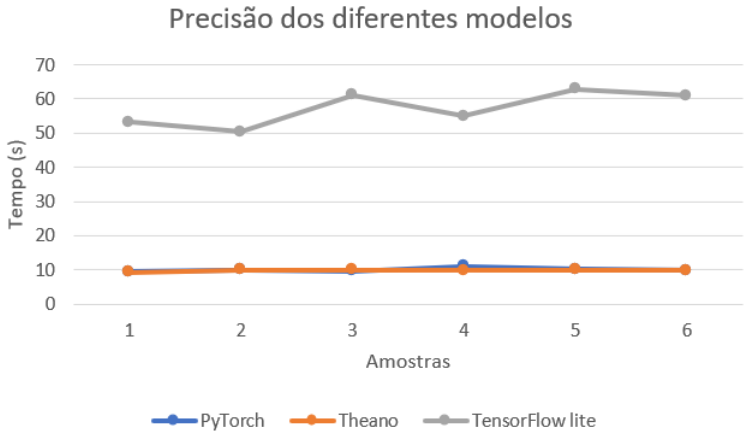


Figura 5.7: Comparação entre precisões (Coral)

5.2.1 Conclusão

Com base no resultados obtidos, a Coral Dev board obtém resultados satisfatórios para modelos previamente treinados, mas para treino de CNNs não é de todo ideal. A precisão das redes treinadas é muito má.

Como tal, a única biblioteca que se adequa é a TensorFlow lite.

5.3 RaspBerry Pi

Na tabela 5.3 são apresentados os resultados para a RaspBerry Pi.

Com a análise dos resultados obtidos na RaspBerry Pi nos diferente critérios, é possível observar que a biblioteca com o menos tempo de treino é a TensorFlow, o que pode indicar uma melhor otimização para CPUs.

Na fase de inferência a TensorFlow e Tensorflow lite têm resultados muitos bons, sendo que a TensorFlow lite tem tempos de inferência ligeiramente melhores.

Por fim, no fator precisão, a PyTorch obteve as taxas de precisão mais altas, com melhor consistência entre os resultados.

Tabela 5.3: Resultados na RaspBerry Pi

Amostra	Biblioteca	Rede	*Num Epochs*	*Batch size*	Fase de treino (s)	Fase de inferência (s) (10000 imagens)	Precisão (%)
1	PyTorch	1	20	32	≈ 17361.9	≈ 40.6	≈ 62
2	PyTorch	1	40	32	≈ 42027.5	≈ 44.4	≈ 65
3	PyTorch	2	20	32	≈ 12689.6	≈ 35.6	≈ 61.5
4	PyTorch	2	40	32	≈ 27854.1	≈ 35.8	≈ 63.7
5	PyTorch	3	20	32	≈ 13111.8	≈ 35.5	≈ 63.4
6	PyTorch	3	40	32	≈ 25486.5	≈ 35.9	≈ 64.1
1	TensorFlow	1	20	32	≈ 6627.4	≈ 10.1	≈ 51.5
2	TensorFlow	1	40	32	≈ 13354.8	≈ 10	≈ 61.1
3	TensorFlow	2	20	32	≈ 3969.2	≈ 5.8	≈ 31.5
4	TensorFlow	2	40	32	≈ 6878.4	≈ 5.7	≈ 61.5
5	TensorFlow	3	20	32	≈ 2817.9	≈ 5.4	≈ 62.6
6	TensorFlow	3	40	32	≈ 5371.6	≈ 5.3	≈ 48.5
1	TensorFlow lite	1	20	32	-	≈ 9.7	≈ 53.1
2	TensorFlow lite	1	40	32	-	≈ 9.7	≈ 50.4
3	TensorFlow lite	2	20	32	-	≈ 5.3	≈ 61.2
4	TensorFlow lite	2	40	32	-	≈ 5.3	≈ 55
5	TensorFlow lite	3	20	32	-	≈ 4.2	≈ 62.9
6	TensorFlow lite	3	40	32	-	≈ 4.1	≈ 61.3
1	Theano	1	20	32	≈ 14053	≈ 52.8	≈ 10.2
2	Theano	1	40	32	≈ 28015.1	≈ 52.7	≈ 10.1
3	Theano	2	20	32	≈ 6683.1	≈ 27.1	≈ 9.9
4	Theano	2	40	32	≈ 13554	≈ 27.1	≈ 9.9
5	Theano	3	20	32	≈ 5846.5	≈ 26.5	≈ 9.9
6	Theano	3	40	32	≈ 10908.5	≈ 26.3	≈ 8.5

Observando o gráfico de comparação entre tempos de treino representado na figura 5.8, é possível concluir que em todos os testes a biblioteca PyTorch exige um tempo de

treino muito superior, em relação às restantes bibliotecas.

A TensorFlow em todas as amostras obteve o melhor tempo de treino, tal como foi referido anteriormente, pode indicar uma melhor otimização da biblioteca para o treino em CPUs.

De uma maneira geral, é também possível concluir que com o aumento do número de *epochs*, o tempo de treino aumenta.

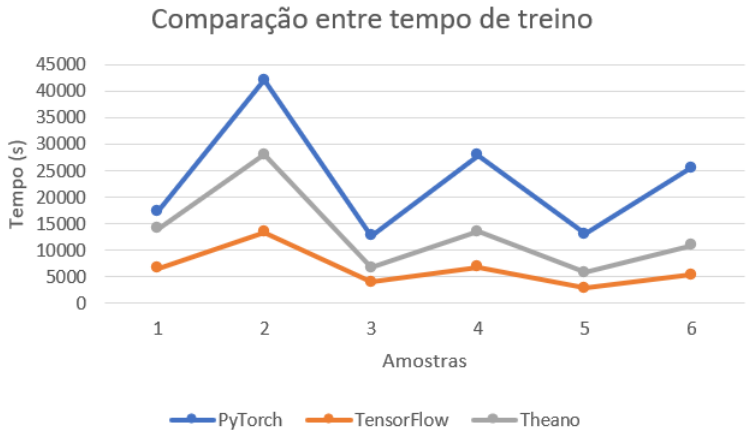


Figura 5.8: Comparação entre tempo de treino (RaspBerry Pi)

Examinando o gráfico da fase de inferência, as claras vencedoras no tempo de execução são a TensorFlow e TensorFlow lite. Com a TensorFlow lite obter tempos ligeiramente melhores, entre 3% a 24% de diferença, algo que faz bastante sentido pois a mesma é otimizada para dispositivos com baixos recursos.

A PyTorch e Theano partilham os piores resultados. Para redes neurais mais complexas, a Theano tem os tempos de inferência mais elevados, mas em redes com menor complexidade, a PyTorch fica com os valores de inferência superiores.

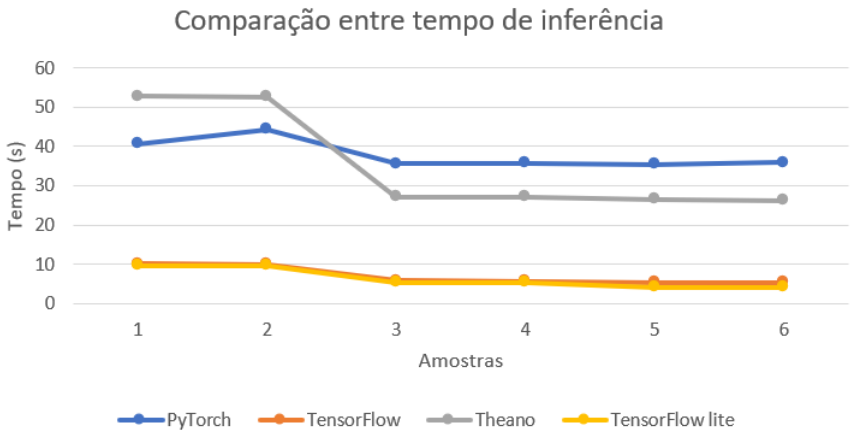


Figura 5.9: Comparação entre tempo de Inferência (RaspBerry Pi)

Em termos de precisão, a PyTorch domina em todas as amostras.

A TensorFlow e TensorFlow lite obtêm precisões ligeiramente inferiores à PyTorch em algumas amostras, embora exista alguma incoerência entre os valores, devido à existência de picos nos valores da precisão.

A Theano de todas é a que tem os piores resultados, valores entre os 10 e 8 %, o que é um aproveitamento muito baixo.

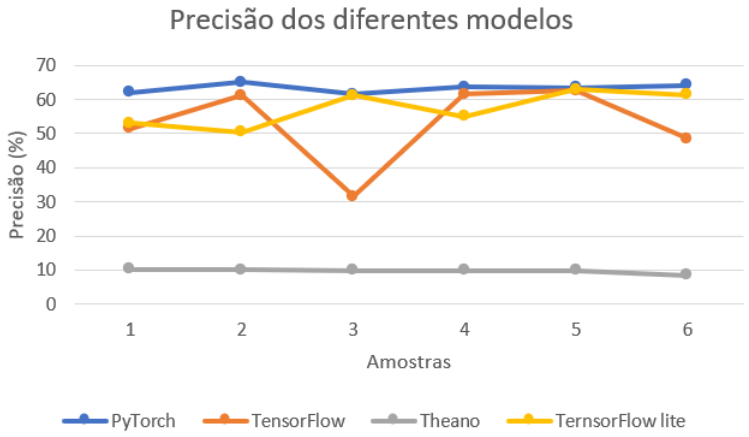


Figura 5.10: Comparação entre precisões (RaspBerry Pi)

5.3.1 Conclusão

Com base nos dados obtidos para a RaspBerry Pi, as bibliotecas que aparentam os melhores resultados são a TensorFlow e TensorFlow lite, uma vez que relação entre os três fatores analisados (tempo de treino, inferência e precisão) parece ser a mais favorável. Visto que tem um tempo de treino e execução mais baixo, com uma boa precisão dos modelos.

5.4 Comparação entre todos os resultados

Analisando todos os resultados, o sistema embebido que melhor *performance* obtém é a Jetson Nano, tanto em fase de treino como fase de inferência. A GPU presente na mesma, aparenta ser uma grande mais valia para *machine learning*.

A biblioteca que se adequa melhor à Jetson Nano é a PyThorch, pois apesar de ter tempos ligeiramente superiores à TensorFlow, apresenta resultados melhores e mais consistentes.

Em relação aos outros dois sistemas embebidos, Coral Dev board e RaspBerry Pi, apenas se adequam para a fase de inferência com a biblioteca TensorFlow lite.

Com a Coral Dev board não foram obtidos resultados relevantes durante a fase de treino, todas as redes treinadas tiveram uma precisão de aproximadamente 10 %.

A RaspBerry Pi demora demasiado tempo durante a fase de treino quando comparada à Jetson Nano.

Na tabela 5.4 são apresentadas as conclusões para cada um dos sistemas embebidos, onde se conclui se as diferentes *board*s servem para treino e/ou inferência e qual a melhor biblioteca para cada uma das *boards*.

Tabela 5.4: Resultados gerais

Hardware	Treino de rede	Modelos treinados	Biblioteca mais adequada
Jetson Nano	**Sim** (PyTorch e TensorFlow)	**Sim** (PyTorch e TensorFlow e TensorFlow lite)	**PyThorch**
Coral Dev board	**Não**	**Sim** (PyTorch e TensorFlow e TensorFlow lite)	**TensorFlow lite**
RaspBerry Pi	**Não**	**Sim** (PyTorch e TensorFlow e TensorFlow lite)	**TensorFlow lite**

Na tabela 5.5 é apresentada uma comparação do tempo de processamento em cada *frame* na fase de treino e após o modelo estar treinado.

A comparação é feita entre todas as bibliotecas e o diferente *hardware*. A rede utilizada para obter estes valores foi a rede 1, citada no capítulo 3, com vinte *epochs*.

Tabela 5.5: Tempo de processamento para cada *frame*

Tempo de processamento	PyTorch	TensorFlow	TensorFlow lite	Theano
Jetson Nano	Treino: 2.0 s Modelo treinado: 1 ms	Treino: 1.8 s Modelo treinado: 3.5 ms	Treino: NA Modelo treinado: 1 ms	Treino: 40.5 s Modelo treinado: 9.2ms
Coral Dev board	Treino: 16.1 s Modelo treinado: 6.9 ms	Treino: NA Modelo treinado: NA	Treino: NA Modelo treinado: 1.8 ms	Treino: 2 s Modelo treinado: 7.9 ms
RaspBerry Pi	Treino: 27.1 s Modelo treinado: 4 ms	Treino: 10.4 s Modelo treinado: 1 ms	Treino: NA Modelo treinado: 0.9 ms	Treino: 21.9 s Modelo treinado: 5.3 ms

5.5 Fatores que podem ter influenciado os resultados

Existem certos fatores que podem influenciar os resultados, como por exemplo, a temperatura.

As altas temperaturas tendem a diminuir a *performance* em sistemas electrónicos de processamento de dados [49].

Existem formas de lidar com o calor produzido pelo CPU, GPU ou outro componente (forma passiva ou ativa), algumas das placas tinham sistemas para lidar com o calor.

A Coral Dev Board é composta por um dissipador e um ventoinha enquanto que a Jetson Nano é apenas constituída por um dissipador. A RaspBerry não nenhuma forma ativa ou passiva de dissipar o calor.

Estas diferentes formas de lidar com a dissipação de calor, muito provavelmente influenciam os resultados.

Um outro fator que provavelmente afetou os resultados foi os diferentes valores de memória RAM entre as placas.

Uma vez que o machine learning é uma tarefa intensiva do ponto de vista da memória, um valor mais baixo de memória RAM vai certamente afetar a performance do sistema.

Capítulo 6

Conclusão

A presente dissertação procura apresentar um *benchmark* de sistemas embebidos para *machine learning*. O principal foco é entender se sistemas embebidos conseguem executar algoritmos *machine learning* e se é eficiente executar esse tipo de algoritmos nesses sistemas.

Depois da análise de alguns algoritmos de *machine learning*, de diferentes arquiteturas para processamento de dados e de bibliotecas para a criação de modelos de *machine learning*, foram escolhidas as opções que mais se adequavam aos objetivos.

O algoritmo seleccionado foi o CNN, pois é um algoritmo com um elevado volume cálculos, o que o torna ideal para testar a performance dos sistemas embebidos.

Os sistemas embebidos escolhidos têm com base a análise feita das diferentes arquiteturas de *hardware* para o processamento de dados (CPU, GPU e NPU). As plataformas escolhidas são Jetson Nano, Coral Dev board e RaspBerry Pi 4b. Cada um desses sistemas têm um ponto forte nas diferentes arquiteturas estudadas, a Jetson Nano tem uma GPU da NVIDIA, a Coral tem um NPU e a RaspBerry tem o processador mais recente de todos os sistemas.

Por fim, foram seleccionadas as bibliotecas para implementar o algoritmo e testar nos sistemas embebidos. As bibliotecas escolhidas são a PyTorch, TensorFlow Lite e Theano.

Cada biblioteca tem um motivo específico para ser seleccionada, a PyTorch obtém a melhor *performance* utilizando uma GPU da NVIDIA, a Tensor Flow lite oferece suporte para o TPU da Coral Dev board e a Theano por ser uma das primeiras bibliotecas para *deep learning*, permite uma comparação interessante com as outras duas bibliotecas que são mais recentes.

Durante o desenvolvimentos dos algoritmos nas três bibliotecas diferentes não existiram grandes problemas, a única parte problemática foi o tipo de processadores dos sistemas embebidos (ARM), que dificultaram a instalação de alguns pacotes necessários para as bibliotecas.

Em termos de resultados, foi possível concluir que a Jetson Nano é o melhor sistema embebido para *machine learning* tanto para treino quanto para inferência, mostrando que a GPU faz uma grande diferença.

A Coral Dev board e a RaspBerry Pi apenas têm uso relevantes para a inferência de modelos, uma vez que a Coral não obtém resultados relevantes no treino dos modelos e a RaspBerry tem um tempo de treino muito elevado.

De uma forma geral todos os objetivos propostos foram cumpridos .

6.1 Trabalhos futuros

Ao nível de desenvolvimentos futuros para este projeto, seria apenas testar os modelos desenvolvidos numa FPGA.

A arquitetura da FPGA favorece a execução de tarefas paralelas, algo que para o *machine learning* é muito benéfico.

Bibliography

[1] R. Prabhu, "Understanding of convolutional neural network (cnn) — deep learning." https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148. Acessado: 22-11-2020. [citado na p. v, 8, 9]

[2] P. Cunningham and S. J. Delany, "k-nearest neighbour classifiers," 2007. [citado na p. v, 10]

[3] J. Jordan, "Neural networks: representation.." https://www.jeremyjordan.me/intro-to-neural-networks/. Acessado: 15-11-2020. [citado na p. v, 15]

[4] https://www.youtube.com/watch?v=cmy7LBaWuZ8&ab_channel=MahmoodNaderan. Acessado: 21-01-21. [citado na p. v, 17]

[5] https://allaboutfpga.com/fpga-architecture/. Acessado: 30-06-21. [citado na p. v, 18]

[6] "What is cpu,gpu and tpu? understanding these 3 processing units using artificial neural networks.." https://www.youtube.com/watch?v=6ZDoFomU10A&list=WL&index=34&t=226s&ab_channel=DOYOUKNOW. Acessado: 24-01-21. [citado na p. v, 19]

[7] "What is embedded vision? – vision campus." https://www.youtube.com/watch?v=O_6IxjD86AE&ab_channel=BaslerAG. Acessado: 15-10-2020. [citado na p. v, 20]

[8] https://www.cs.toronto.edu/~kriz/cifar.html. Acessado: 13-02-21. [citado na p. v, 32]

[9] https://coral.ai/docs/dev-board/get-started/#flash-the-board. Acessado: 09-03-21. [citado na p. v, 34, 35]

[10] "Comparison with other libraries." https://github.com/tiny-dnn/tiny-dnn/wiki/Comparison-with-other-libraries. Acessado: 23-10-2020. [citado na p. vii, 25]

[11] A. Ng, *Machine Learning Yearning*. GitHub, 2018. [citado na p. 1]

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," 2014. [citado na p. 1]

[13] "Basic introduction to computer vision." https://kapernikov.com/basic-introduction-to-computer-vision/. Acessado: 28-10-2020. [citado na p. 1]

[14] A. Zhuravchak, "Machine learning fails when it comes to embedded system. here's why." https://towardsdatascience.com/machine-learning-fails-when-it-comes-to-embedded-system-9ce6def9ba75. Acessado: 28-10-2020. [citado na p. 1]

[15] V. Alcouffe, "On-device-ai: Machine learning on embedded systems, made easy.." https://medium.com/ai-in-plain-english/bringing-on-device-ai-to-off-the-shelf-mcus-made-easy-120c47e16228. Acessado: 11-11-2020. [citado na p. 1]

[16] G. Oppy and D. Dowe, "The turing test." https://plato.stanford.edu/entries/turing-test/. Acessado: 07-10-2020. [citado na p. 5]

[17] T. Cover and P. Hart, "Nearest neighbor pattern classification," in *IEEE Transactions on Information Theory, vol. 13*, pp. 21–27, IEEE, 1967. [citado na p. 6]

[18] Y. Hamdi, "The story of machine learning." https://medium.com/@yasminehamdi/the-story-of-machine-learning-7ac3e8a5eaf9. Acessado: 08-10-2020. [citado na p. 6]

[19] D. Fumo, "Types of machine learning algorithms you should know." https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861. Acessado: 08-10-2020. [citado na p. 6]

[20] I. Hasabo, "Image classification using machine learning and deep learning." https://medium.com/swlh/image-classification-using-machine-learning-and-deep-learning-2b18bfe4693f. Acessado: 10-10-2020. [citado na p. 7]

[21] A. Lee, "What is embedded vision?." https://www.bell-labs.com/var/articles/putting-ai-historical-perspective/. Acessado: 21-10-2020. [citado na p. 8]

[22] M. Sidana, "Intro to types of classification algorithms in machine learning." https://medium.com/sifium/machine-learning-types-of-classification-9497bd4f2e14. Acessado: 24-10-2020. [citado na p. 9]

[23] O. Harrison, "Machine learning basics with the k-nearest neighbors algorithm." https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761. Acessado: 24-10-2020. [citado na p. 10]

[24] M. Patel, "Naive bayes — machine learning algorithm." https://medium.com/@meetpatel12121995/naive-bayes-machine-learning-algorithm-aaf57bdc8d87. Acessado: 26-10-2020. [citado na p. 10]

[25] R. Alake, "Introduction to computer vision." `https://towardsdatascience.com/introduction-to-computer-vision-b44bb4c495ab`. Acessado: 12-10-2020. [citado na p. 11]

[26] J. Cohen, "Computer vision at tesla." `https://heartbeat.fritz.ai/computer-vision-at-tesla-cd5e88074376`. Acessado: 12-10-2020. [citado na p. 11]

[27] D. Serpanos and T. Wolf, *Architecture of Network Systems*. Morgan Kaufmann, 2012. [citado na p. 12]

[28] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of computer organization and architecture*. John Wiley Sons, 2004. [citado na p. 12]

[29] C. Watson, "How does a cpu work?." `https://www.techwalla.com/articles/what-are-the-functions-of-a-cpu-in-a-computer`. Acessado: 13-11-2020. [citado na p. 13]

[30] D. A. Patterson and J. L.Hennessy, *Computer Organization and Design: The Hardware/Software Interface, 4 edição*. Morgan Kaufmann, 2004. [citado na p. 13]

[31] M. Rouse, "Dram (dynamic random access memory)." `https://searchstorage.techtarget.com/definition/DRAM`. Acessado: 19-01-2021. [citado na p. 13]

[32] "Cpu, gpu, and tpu for fast computing in machine learning and neural networks." `https://svitla.com/blog/cpu-gpu-and-tpu-for-fast-computing-in-machine-learning-and-neural-networks`. Acessado: 29-10-2020. [citado na p. 14, 15]

[33] Z. A. Memon, F. Samad, Z. R. Awan, A. Aziz, and S. S. Siddiqi, "Cpu-gpu processing," 2017. [citado na p. 15]

[34] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," 2012. [citado na p. 15]

[35] B. Wheeler, "A new era of network processing," 2013. [citado na p. 15]

[36] R. P. Brent and H. T. Kung, "Systolic vlsi arrays for polynomial gcd computation," 1984. [citado na p. 16]

[37] "What is neural processing unit (npu)?." `https://iq.opengenus.org/neural-processing-unit-npu/`. Acessado: 15-11-2020. [citado na p. 17]

[38] A. Moltzau, "The google edge tpu." `https://alexmoltzau.medium.com/the-google-edge-tpu-61dd08229d5a`. Acessado: 30-10-2020. [citado na p. 17]

[39] `https://www.ni.com/pt-pt/innovations/white-papers/08/fpga-fundamentals.html`. Acessado: 30-06-21. [citado na p. 17]

[40] C. M. Maxfield, *FPGAs: Instant Access*. Newnes, 2008. [citado na p. 18]

[41] M. Ender, P. Swierczynski, S. Wallat, M. Wilhelm, P. M. Knopp, and C. Paar, "Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream," 2019. [citado na p. 19]

[42] "What is embedded vision?." https://www.visiononline.org/blog-article.cfm/What-is-Embedded-Vision/90. Acessado: 14-10-2020. [citado na p. 20]

[43] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," 2013. [citado na p. 24]

[44] Y. Sun, "Tesla and pytorch — pytorch developer conference highlights part." https://medium.com/data-science-bootcamp/tesla-and-pytorch-pytorch-developer-conference-highlights-part-3ed36f2c9d5e. Acessado: 04-11-2020. [citado na p. 26]

[45] J. Brownlee, "Introduction to the python deep learning library theano." https://machinelearningmastery.com/introduction-python-deep-learning-library-theano/. Acessado: 12-12-2020. [citado na p. 26]

[46] J. Brownlee, "Introduction to the python deep learning library theano." https://machinelearningmastery.com/introduction-python-deep-learning-library-theano/. Acessado: 13-12-2020. [citado na p. 26]

[47] A. Beklemysheva, "Why use python for ai and machine learning?." https://steelkiwi.com/blog/python-for-ai-and-machine-learning/. Acessado: 12-01-2021. [citado na p. 27]

[48] "Crossentropyloss." https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html. Acessado: 12-12-2020. [citado na p. 38]

[49] https://www.technipages.com/what-is-thermal-throttling. Acessado: 21-02-21. [citado na p. 53]