UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
INSTITUTO MULTIDISCIPLINAR

FELIPE BEZERRA DE MELO

Reconhecimento de Dígitos Utilizando Redes Neurais em GPU

Prof. Fellipe Ribeiro Duarte, D.Sc.
Orientador

Prof. Marcelo Panaro de Moraes Zamith, D.Sc.
Co-orientador

Rio de Janeiro, dezembro de 2017

Agradecimentos

Quero agradecer, em primeiro lugar, a Deus, pela força e coragem durante toda esta longa caminhada.

Agradeço também a todos os professores que me acompanharam durante a graduação, em especial ao Prof. Dr. Fellipe Ribeiro Duarte e ao Prof. Dr. Marcelo Panaro de Moraes Zamith, responsáveis pela orientação na realização deste trabalho.

Dedico esta, bem como todas as minhas demais conquistas, a minha mãe, que sempre esteve ao meu lado me dando força e apoio desde sempre. Agradeço também a minha namorada Bianca, que sempre me ajudou e teve paciência nos momentos mais estressantes. Agradeço aos meus colegas de graduação que estiveram presentes ao longo das matérias da minha graduação.

RESUMO

Reconhecimento de Dígitos Utilizando Redes Neurais em GPU

Felipe Bezerra de Melo

dezembro/2017

Orientador: Fellipe Ribeiro Duarte, D.Sc.
COPPE - UFRRJ

Neste trabalho de conclusão de curso tivemos como objetivo explorar técnicas de classificação de dígitos utilizando redes neurais. Com base na *Lennet5* foi desenvolvido uma rede neural convolucional capaz de detectar imagens de dígitos da base de dados MNIST, um outro ponto explorado, foi a aceleração do processo de aprendizado com placas gráficas bem como os prós e contras que se fazer uso das mesmas. Por fim, são apresentados os experimentos e conclusão do trabalho.

ABSTRACT

Reconhecimento de Dígitos Utilizando Redes Neurais em GPU

Felipe Bezerra de Melo

dezembro/2017

Advisor: Fellipe Ribeiro Duarte, D.Sc.
COPPE - UFRRJ

In this work of course completion we aimed to explore techniques of digit classification using neural networks. Based on the Lennet5, a convolutional neural network capable of detecting digit images of the MNIST database was developed. Another point explored was the acceleration of the learning process with graphic cards as well as the pros and cons to be made their use. Finally, the experiments and conclusion of the work are presented.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

NN	Neural Networks
GPU	Graphic Processor Unit
RNAs	Redes Neurais Artificiais
RNA	Rede Neural Artificial
ANN	Artificial Neural Networks
CUDA	Compute Unified Device Architecture
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
CPU	Computing processor unit
HPC	High-Performance Computing
OCR	Optical Character Recognition
MNIST	Modified National Institute of Standards and Technology
NIST	National Institute of Standards and Technology
RAM	Random-Access Memory

Sumário

Capítulo 1

Introdução

A visão humana é o sentidos mais complexo dentre o conjunto que possuímos. Portanto, a facilidade com que percebemos o mundo a nossa volta mascara a complexidade por trás do funcionamento desse mecanismo de percepção tão complexo. Por exemplo, o ser humano é capazes de detectar nuances suaves em uma imagem ou perceber detalhes em um objeto parcialmente amostra. Essas capacidades mostram o quão complexa e poderosa a visão pode ser (Szeliski, 2010; Nixon and Aguado, 2008).

Uma vez que a visão humana possui tamanha capacidade, faz sentido tentar aprender o seu funcionamento e desenvolver soluções de visão artificial baseadas nela. Porém, na prática não é possível reproduzí-la totalmente, pois se por um lado não se tem o conhecimento total do funcionamento do sistema de visão humana, por outro nem todas as suas características são desejáveis (Nixon and Aguado, 2008). Uma exemplo segundo (Nixon and Aguado, 2008) é a incapacidade que o sistema de visão humana tem em avaliar uma distância de dois objetos de forma absoluta, comumente usa-se adjetivos como perto ou longe, sem saber exatamente a distância entre eles, para um computador o oposto ocorre, são as distâncias quantitativas que temos como medidas.

A visão computacional deseja capturar e interpretar o mundo a nossa volta, baseando-se no modelo de visão humana. Porém, como já foi dito, replicar todos os

detalhes contidos no processo do sistema de visão humana é difícil, tornando a visão computacional um desafio, abrindo caminho para diferentes propostas de soluções (Nixon and Aguado, 2008). Durante algum tempo tentou-se desenvolver algoritmos para reproduzir o que é feito pelo cérebro humano, as primeiras tentativas buscavam determinar regras e padrões fixos para detectar objetos em uma determinada imagem, porém isso logo tornou-se inviável, dada as combinações e possibilidades que podem existir em uma imagem. Ademais, esta tarefa é difícil por tratar-se de um problema de inversão (Szeliski, 2010).

De acordo com Szeliski (2010); Sebaa et al. (2006) um problema de inversão consiste em utilizar o resultado de alguma medida e, a partir deste, inferir os parâmetros que caracterizam a medida observada. Esta definição retrata o que a visão computacional deseja realizar, pois já é de conhecimento do sistema o resultado final da observação, i.e as imagens, e o objetivo é descobrir fatores que representam aquela imagem como o seu contexto e o seu conteúdo, o que caracteriza um problema de inversão.

Problemas relacionados com visão computacional não faltam, diversas propostas na literatura atacam problemas diferentes com soluções variadas como: Reconhecimentos de objetos para carros autônomos (Teichman and Thrun, 2011), detecção de placas de carros em (Nagare, 2011), reconhecimento de impressão digital em (Kaur et al., 2008), reconhecimento de dígitos em (LeCun et al., 1998) e detecção de fraude em documentos em (Bertrand et al., 2013). Em Rowley et al. (1998) é apresentado uma proposta de detecção de rosto em imagens. Antes de explicar os detalhes sobre os desafios relacionados ao problema uma distinção é feita, entre detecção e reconhecimento, no reconhecimento todas as imagens contem rostos e o desafio e tentar identificar e aprendendo quem está na imagem, já na detecção, (Rowley et al., 1998) exalta que o problema é primeiramente determinar se existe um rosto em determinada imagem, o que adiciona um componente de maior complexidade ao problema. Logo, a dificuldade está em apresentar ao sistema exemplos suficientes para que ele consiga aprender essa diferença, isso pode aumentar de maneira drástica a quantidade de amostras. A maneira como esse problema foi contornado por (Rowley et al., 1998) foi adicionando seletivamente e gradativamente imagens de um con-

junto grande de imagens. De fato, em problemas desse tipo, muitos fatores podem influenciar no sucesso dos resultados, pois em uma imagem o rosto de uma pessoa pode estar de diversas formas, como estar de lado, inclinado, apontando para cima ou para baixo. Sem contar na escala (Rowley et al., 1998).

Reconhecimento automático de objetos é uma das áreas mais abrangentes da visão computacional que está presente em muitos problemas ligados a sistemas robóticos. Por exemplo, para um sistema de carro autônomo é de suma importância que o carro seja capaz de detectar pedestres, ciclistas, veículos e outros elementos ao seu redor (Teichman and Thrun, 2011). Além disso, existem outras motivações para o desenvolvimento de sistemas de visão para carros pois, em larga escala os carros autônomos podem: diminuir o número de acidentes no trânsito; otimizar o uso de combustível, o que implicaria em uma baixa na poluição emitida pelos mesmos e diminuir o tempo gasto pelas pessoas no transito (Teichman and Thrun, 2011).

Outro exemplo de aplicação que mistura tecnologia com meio de transporte é o reconhecimento automático de placas de carro, para controle automático de multa por excesso de velocidade, controle de tráfego e identificação de automática (Nagare, 2011; Patel Smt Chandaben Mohanbhai et al., 2013). A motivação para sistemas desse tipo é o grande volume de carros envolvidos, o que torna esse processo ineficiente caso fosse realizados de forma manual. Uma vez que os carros são fotografados tem início o processo de detecção, a próxima etapa é a identificação da placa (em (Nagare, 2011) são apresentados soluções para esta etapa). Após realizado o recorte da região onde está localizada a placa do carro operações de conversão para escala de cinza e aumento de contraste são aplicadas para destacar o conteúdo da placa das demais cores ainda remanescentes da imagem, o objetivo é ter um recorte binário da placa onde as letras são totalmente pretas e o fundo totalmente branco (Nagare, 2011). Com o recorte da placa deve-se fazer a extração de características de tais elementos, este passo é importante pois ele extrairá informações que representam as imagens dos caracteres presentes na placa original, porém de forma a ser melhor trabalhada matematicamente do que o dado das imagens crua. Em (Nagare, 2011) são apresentados as representações que utilizam a transformada de Fan-beam a uma proposta baseada em geometria.

O problema de detecção de fraude em documentos digitalizados busca encontrar alterações em documentos feitos por meios físicos ou por intermédio da utilização de softwares de processamento de imagens (Bertrand et al., 2013). Existem alguns desafios quando se lida com este tipo de problema, pois dependendo de diversos fatores como o resultado da digitalização do documento, a qualidade do papel, o tipo de documento, o tipo de coloração da impressão e até o tipo de informação contida no mesmo afetam a detecção (Bertrand et al., 2013).

Algumas fraudes não podem ser percebidas a olho nu (Bertrand et al., 2013), portanto um sistema de visão inteligente com tal capacidade extrapola as capacidades de visão humana, por conta disso as soluções para este problema devem extrair características do documento que passam despercebidos pelo sistema visual humano. Existem técnicas que podem ser aplicadas a documentos que dificultam a realização de fraude, porém nem todos os documentos que circulam no dia a dia possuem tais características (Clarisse MANDRIDAKE, Amine OUDDAN, Mathieu HOARAU).

Em problemas desse tipo muitos fatores podem influenciar no sucesso dos resultados e, portanto, este trabalho tem como objetivo apresentar um estudo da técnica clássicas de reconhecimento de dígitos baseados no trabalho de (LeCun et al., 1998) onde é apresentada a Lenet5. A Lenet5 é uma rede neural convolucional que, assim como as outras redes do mesmo tipo, apresentam um custo de tempo computacional para aprender e extrapolar a partir de um conjunto conhecido de dados, o que é conhecido como aprendizado supervisionado. Com o intuito de abordar este problema de custo estudamos, também, o impacto de utilizar paralelismo nos dados da tarefa utilizando GPU's.

1.1 Contribuições do trabalho

este trabalho apresenta as seguintes contribuições:

- desenvolvimento de uma rede neural capaz de aprender a classificar dígitos;
- usar GPU para fazer o paralelismo do treinamento da rede;

- validar o speedup do problema em placas gráficas diferente;
- avaliar o desempenho na corretude das classificações entre CPU e GPU;
- um artigo sobre o overhead gerado pela biblioteca Theano ao realizar operações básicas de uma rede neural;

1.2 Organização do trabalho

Este trabalho está dividido em 6 capítulos onde o capítulo 2 apresenta mais detalhes sobre o problema de reconhecimento de dígitos e discute os passos necessários para determinada tarefa, no capítulo 3 são apresentados os conceitos de programação para GPU's e como esse paradigma de programação difere da programação sequencial, no capítulo 4 são discutidos conceitos, características e arquiteturas que envolvem o estudo das redes neurais artificiais bem como a forma com que essas redes aprendem, já o capítulo 5 são descritos os experimentos realizados e são apresentados os resultados obtidos e no capítulo 6 os resultados são interpretados e são apresentados algumas propostas de trabalhos futuros.

Capítulo 2

Reconhecimento de Dígitos

O Reconhecimento de dígitos é um campo de pesquisa dentro de visão computacional que busca extrair informação através de imagens contendo caracteres (Vamvakas et al., 2010). O Reconhecimento de dígitos é um problema bem difundido e estudado, possuindo uma literatura diversa sobre o assunto. Segundo Vamvakas et al. (2010) o reconhecimento de caracteres de máquina pode ser considerado um problema resolvido. No entanto, com caracteres escritos a mão não se pode dizer o mesmo, visto que a escrita cursiva varia de pessoa para pessoa, como ilustrado figura 2.1, o que torna este um problema mais difícil.

O reconhecimento de dígitos escritos a mão possui diferentes abordagens que dependem do objetivo da aplicação e dos dados disponíveis. Por exemplo, Bertrand et al. (2013) lida com a extração de conteúdo em um documento e Tay et al. (2001) lida com a identificação de endereço de cartas, nestes conjuntos, onde os dados são textos corridos, é comum efetuar a segmentação das letras para depois realizar o

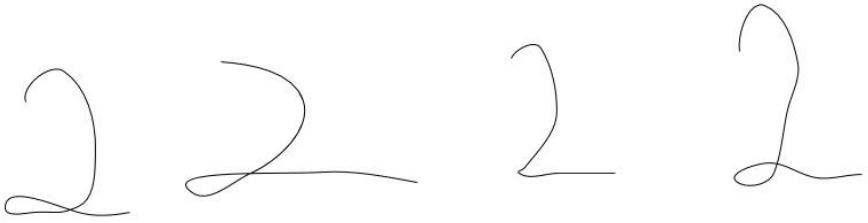


Figura 2.1: Representação da variedade de formas da escrita a mão

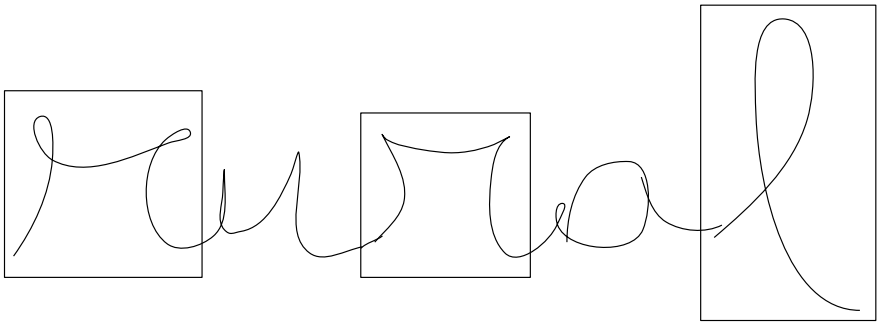


Figura 2.2: Segmentação de caracteres cursivos

reconhecimento das palavras e, portanto, a ausência de informação sobre o tamanho que cada letra ocupa aumenta mais a complexidade desta tarefa. Este é um dos fatores que podem influenciar no desempenho de um sistema de reconhecimento de dígitos (Camastra and Vinciarelli, 2001).

O conhecimento sobre os dados os quais serão trabalhados é muito importante para o sucesso de um sistema desse tipo. Logo, existem algumas bases de dados próprias para reconhecimento de dígitos, como o MNIST, que foi primeiramente proposta por LeCun et al. (1998), esta base é composta de imagens de dígitos de 0 a 9, conforme apresentado no capítulo 5. LeCun et al. (1998) utilizou o MNIST para aprender a representação dos dígitos e, posteriormente, serem aplicados no reconhecimento de conteúdo de documentos digitalizados, mesmo quando os documentos não foram preenchidos pelas mesmas pessoas que contribuíram para o MNIST, onde ele foi capaz de utilizar essa informação para generalizar a representação dos dígitos no seu trabalho.

Se considerarmos caracteres cursivos como na imagem 2.2, pode-se observar que a área que cada caractere ocupa varia, no caso dos dígitos esse fator pode ter uma variação menor, mas, ainda assim, deve ser levado em consideração. Duas técnicas são muito utilizadas para fazer essa segmentação como tentativa e erro, ou seja, selecionar várias áreas e guardar aquelas que conseguiram um bom resultado, e fazer uso de técnicas de processamento de imagem como filtro de segmentação (Vamvakas et al., 2010; Reddy et al., 2013).

Como visto no capítulo 1, interpretar o mundo visualmente como nosso cérebro

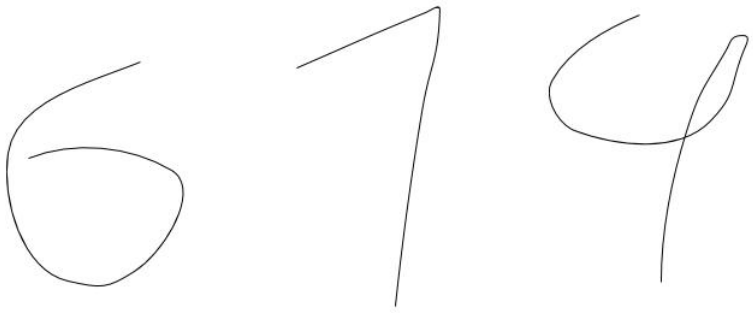


Figura 2.3: Imagens ambíguas de dígitos

faz não é uma tarefa fácil, diversos problemas podem surgir no desenvolvimento de um método automático de reconhecimento de dígitos. Alguns já foram mencionados como: tamanho da letra e como a variação de forma dos dígitos varia de pessoa para pessoa. Mas um problema mais sério pode ser a ambiguidade dos dígitos, essa situação pode atrapalhar até mesmo a visão humana a identificar os dígitos apresentados, como visto na figura 2.3, onde pode-se dizer que o primeiro dígito é um 6 ou 0, o segundo como 1 ou 7 e o último como 4 ou 9.

Para isso, o processo de reconhecimento de texto manuscrito pode ser dividido em etapas, que buscam melhorar o resultado final, ou seja, a classificação do digito de forma automatizada.

2.1 Etapas de reconhecimento

Segundo Reddy et al. (2013); Processing and Recognition (1999), um sistema de visão computacional é composto pelas seguintes etapas: pré-processamento e segmentação, extração de características, classificação e pós processamento. Acredita-se que etapas parecidas ocorrem no cérebro humano, um exemplo disso é a insuficiente capacidade de transmissão total da informação captada através do olho ao cérebro pelo nervo óptico, indicando alguma etapa de pré-processamento dos dados que são registrados pelos olhos (Nixon and Aguado, 2008).

- **Pré-processamento e Segmentação** - Muitas vezes o dado crú possui características que podem atrapalhar a detecção. Portanto, uma etapa de pré-processamento necessita ser executada a fim de fazer com que esse dado fique com características que auxiliem na manipulação computacional do mesmo (Reddy et al., 2013). Um exemplo destas técnicas são a binarização e a redução de ruídos.

  A segmentação é um tipo de pré-processamento que envolve a separação de regiões em uma imagem, a própria binarização é um exemplo de segmentação, porém uma imagem pode conter mais de um grupo de pixels que fazem parte de um mesmo contexto, resultando em diversos segmentos em uma mesma imagem (Processing and Recognition, 1999). Portanto, uma abordagem mais elaborada pode ser proposta. Em Reddy et al. (2013) é mencionado o uso de histogramas para auxiliar neste tarefa.

  Outro motivador importante para esta etapa é a redução de variáveis, uma imagem com tamanho 100 x 100 pixels e 3 canais de cores possui 30000 informações associadas a ela, isso pode gerar um problema de dimensionalidade das entradas, pois as bases de dados de imagens podem ter milhares ou até milhões de imagens (Krizhevsky et al.). Portanto, qualquer processo que consiga reduzir o número variáveis é bem vinda para problema que lidam com tamanha quantidade de dados.

- **Extração de Características** - É nesta etapa onde as informações da imagem são transformadas em um conjunto de características que serão enviadas para o sistema de reconhecimento (Reddy et al., 2013) e (Processing and Recognition, 1999).

  Porém, existem algumas ressalvas em relação ao poder de generalização quando se trata de extração de características baseada em uma regra empírica, isso pode fazer com que o resultado final dependa muito da técnica de extração de característica.

  Segundo LeCun et al. (1998), a extração de características foi idealizada com o intuito de diminuir o número de parâmetros que seriam utilizadas pelo classifi-

cador, pelo baixo poder de computação das máquinas a algumas décadas atrás. Contudo, esse fator não se justifica mais, tanto por conta do aumento do poder computacional quanto pela disponibilidade de dados grandes para diversos problemas e também pelas diversas técnicas de inteligência computacional de hardware e softwares disponíveis hoje de forma gratuita e transparente. Por conta disso, esta etapa é mais sobre transformar o dado do que propriamente diminuí-lo por motivos de capacidade.

- **Classificação** - Segundo Nixon and Aguado (2008), classificação consistem em atribuir a um conjunto de entradas um rótulo (classe). Este passo pode ser considerado como o coração do método que está sendo desenvolvido. O sucesso, por assim dizer, desta etapa depende dos resultados, do método adotado para classificação e da qualidade das características extraídas dos objetos que se deseja classificar.

  Um ponto interessante levantado por Vamvakas et al. (2010) é que a classificação pode ser feita em etapas, em um primeiro momento podemos classificar objetos semelhantes como sendo da mesma classe e em um próximo passo extrair mais características dos objetos não classificados para eliminar uma possível ambiguidade, esse tipo de abordagem pode variar de acordo com a aplicação.

- **Pós-Processamento** - O pós-processamento é a fase final da detecção, podemos dizer que trata-se da apresentação dos resultados obtidos a partir da imagem, pode ser apresentada em diversas estruturas como tabelas, gráficos ou descrição textual (Reddy et al., 2013).

Dentro do reconhecimento de dígitos, o pré-processamento mais indicado é a binarização, que separa os *pixels* do fundo e do dígito, tornar as imagens preto e branco é a melhor opção, dessa forma uma imagem que possui 3 canais de cores pode ser reduzida para uma matriz binária de valores. Outra parte importante é a normalização das dimensões da imagem tornando todas NxN (Vamvakas et al., 2010).

A extração de características para reconhecimento de dígitos visa tentar selecionar, a partir das imagens pré-processadas, valores estruturados que representem aquele dado. Isto, porém, isso não é algo mandatório dado que deferentes trabalhos utilizam o mesmo dado gerado no pré-processamento como características extraídas das imagens, como LeCun et al. (1998) fez em um dos seus experimentos, enquanto outros podem extrair vetores a partir de regras de somas entre linhas e colunas como é feito por Vamvakas et al. (2010).

Diversas técnicas de aprendizado de máquina, como redes neurais, k vizinhos, máquina de vetor de suporte (SVM), dentre outras, podem ser utilizadas com o propósito de se classificar um conjunto de imagens contendo dígitos. Estas técnicas, utilizam aprendizado supervisionado, quando o sistema de classificação aprende a partir de exemplos onde já se é conhecido o resultado, para que assim o método possa aprender os padrões e, a partir disso, generalizar as entradas e ser capaz de classificar novos elementos (LeCun et al., 1998), no capítulo 4 esse assunto é discutido com mais detalhes.

É importante que um sistema de classificação de dígitos consiga, generalizar bem para que ele possa identificar imagens que nunca foram apresentadas a ele. Conseguir 100% de precisão pode ser utópico até mesmo para o cérebro humano, pois as variáveis podem ser muito grandes, mas é possível obter resultados a cima de 90% de precisão com classificadores relativamente simples (Nielsen, 2016).

Para se construir um classificador que reconheça dígitos, algumas características devem se levadas em consideração, como: as particularidades do dado disponível e o número de parâmetros necessários para determinado método (Vamvakas et al., 2010), por exemplo, uma rede neural costuma utilizar quantidade grande de parâmetros devido a como este método aprende, já um algoritmo de k vizinhos não possui tantos parâmetros, mas, este pode sofrer com uma perda na cobertura da capacidade de classificações quando se limita o número de vizinho escolhidos para o cálculo de previsão, o que acarreta em uma maior precisão.

O pós-processamento tem a sua importância em termos de análise do que foi proposto, pois a partir disso podemos tirar conclusões de onde o classificador está

acertando e errando mais, por exemplo, pode ser esperado que um classificador de dígitos escritos a mão confunda o número 1 com o número 7, já mostrado na figura 2.3, porém outras inferências não óbvias podem aparecer.

Tratar o problema de reconhecimento de dígitos como um problema de reconhecimento de padrão pode não ser fácil do ponto de vista descritivo, como tentar definir um número usando o português, como: O 6 é um círculo inferior que possuí uma curva saindo do canto superior esquerdo do círculo. Por outro lado, do ponto de vista computacional pode-se conseguir melhores resultados. (Nielsen, 2016).

Capítulo 3

GPU

3.1 GPU e CUDA

As Placas de vídeos (GPUs - *Graphics Processing Unit*) são componentes de hardwares voltados para execução de algumas operações gráficas básicas, que eram computadas pela CPU, como renderização de imagens, realização de cálculos de polígonos complexos, cálculo de iluminação em uma cena, dentre outras tarefas (Cook, 2013).

A evolução dos processadores dos últimos anos provocou uma mudança no paradigma de programação. Atualmente, há um interesse em programação paralela, permitindo resolver os problemas de maneira mais rápida (Cheng et al., 2013). Segundo Cheng et al. (2013) o paralelismo do ponto de vista operacional pode ser definido como uma forma de computação onde muitos cálculos podem ser executados ao mesmo tempo, isso implica que grandes problemas podem ser divididos em problemas menores e serem resolvidos simultaneamente.

Segundo Cheng et al. (2013), o desenvolvimento de soluções paralelas requer duas tecnologias distintas trabalhando juntas, arquitetura de hardware e uma programação voltada para soluções em paralelo. A arquitetura de hardware compõem toda a estrutura física utilizada como cores, memória, controladores e etc. A programação envolve linguagens, compiladores, técnicas de desenvolvimento de software,

bibliotecas e como elas farão uso da arquitetura de hardware disponível.

De acordo com Cheng et al. (2013), existem dois tipos de paralelismo: o paralelismo de tarefas e o paralelismo de dados. O paralelismo de tarefas busca paralelizar funções ou tarefas, dividindo-as e fazendo com que executem de forma mais independente possível. O paralelismo de dados é utilizado em aplicações que possuem grandes quantidades de dados que podem ser distribuídos e computados de forma independente. A programação em paralelo disponível nas placas gráficas, como a implementada pela NVIDIA, funciona melhor com o paralelismo de dados (Cheng et al., 2013).

Atualmente, as placas gráficas possuem centenas ou milhares de cores para executar o processamento do que será apresentado no sistema de saída de vídeo das máquinas. A principal característica do processamento gráfico é a independência entre o que está sendo processado, onde cada core executa exatamente as mesmas instruções, porém, sobre dados diferentes (Zamith and Clua, 2015).

O grande poder computacional das GPU's e um baixo custo, em comparação com a capacidade de processamento das CPU's, são dois fatores que fazem da GPU um excelente hardware de computação paralela. Permitindo a computadores caseiros tornarem-se pequenos clusters computacionais. Isto se deu, em parte, pela demanda da indústria de jogos, pois, a industria de jogos necessita cada vez mais de processamento paralelo por conta dos gráficos cada vez mais elaborados (Harris, 2005).

Alguns pesquisadores passaram a utilizar as GPU's para tarefas de propósito geral, estendendo a sua funcionalidade de processamento gráfico (Zamith et al., 2007). E, em 2008 a NVIDIA lançou a arquitetura unificada de GPU conhecida como CUDA - *Compute Unified Device Architecture*. O CUDA é um SDK de desenvolvimento para placas gráficas. Podendo ser considerada uma biblioteca de desenvolvimento feita para o C/C++, que permite ao desenvolvedor paralelizar seu código e enviá-lo para ser executado na GPU (Zamith and Clua, 2015).

Uma alternativa ao CUDA é a biblioteca OpenCL, ambas as bibliotecas são

voltadas para desenvolver soluções de alto desempenho. Porém, o CUDA é uma biblioteca mais madura, de propriedade da NVIDIA, voltado apenas para as placas da arquitetura unificada da NVIDIA. OpenCL, por outro lado, funciona tanto em GPU's da Intel, NVIDIA e ATI. Por ser uma biblioteca aberta e contemplar uma grande quantidade de hardwares, algumas funcionalidades presentes no CUDA ainda não estão presentes no OpenCL, como paralelismo dinâmico Cook (2013). Neste trabalho será utilizada o CUDA e duas placas da NVIDIA.

O SDK do CUDA disponibiliza um conjunto de ferramentas para o desenvolvedor, como o profiler, chamado neste trabalho de CUDA Profiler para fins didáticos. O CUDA Profiler permite o monitoramento da execução do código dentro da GPU, ajudando a entender onde o código pode ser melhorado e onde encontram-se gargalos na execução (Zamith and Clua, 2015).

A GPU não é um hardware independente, pois requer que a CPU gerencie todo o seu funcionamento. Assim, a alocação de memória e outros recursos da GPU fica sob responsabilidade da CPU. E, a CPU determina o momento e quais códigos irão ser chamados dentro da GPU Cheng et al. (2013).

O CUDA, que permite as GPU's executarem código de propósito geral, foi lançado em 2008 e desde então diversos recursos foram incorporados e melhorados no hardware gráfico. Ao final de 2008, as placas e o SDK do CUDA evoluíram, passando por diferente arquiteturas de placas com diferentes características implementadas em hardware e software, como precisão dupla em operações de ponto flutuante, operações atômicas, maior capacidade de memória cache L1 e L2, paralelismo dinâmico, que será discutido em detalhes nas seções futuras, entre outras características. Além disso, outras ferramentas e bibliotecas puderam ser desenvolvidas em um nível maior de abstração do CUDA e com possibilidade otimização de algumas funções.

A capacidade de paralelismo está inversamente relacionada com o quantidade de dependência entre os dados que serão calculados, se pensarmos em uma função matemática simples como $f(x) = x^2$ e determinarmos um intervalo de número inteiros de 1 até 100, o cálculo de cada posição é independente, tornando esse problema altamente paralelizável (Cook, 2013). Porém, muitas vezes apenas parte da solução

possui esta característica, portanto, o problema é parcialmente paralelizável o que limita desempenho global desta execução (Cook, 2013).

Caso parte da aplicação em execução tenha uma dependência maior de dados do que de processamento, este será um gargalo na solução paralela. Por conta das particularidades do paralelismo da GPU e da capacidade das placas de vídeo, a GPU funciona melhor com solução totalmente paralelas, porque o custo de gerenciar essas dependências, pode prejudicar o desempenho da solução.

A GPU execução de um código CUDA em GPU deve ser organizada em 3 elementos diferentes, grades, blocos e threads como mostrado na figura 3.1. Uma grade contém um conjunto de blocos enquanto os blocos são formados por um conjunto de threads. As threads de um mesmo bloco compartilham um contexto de execução e executam a mesmo instrução a cada ciclo, um bloco deve ser alocado e executar suas instruções até o fim sem interrupções (Cook, 2013)

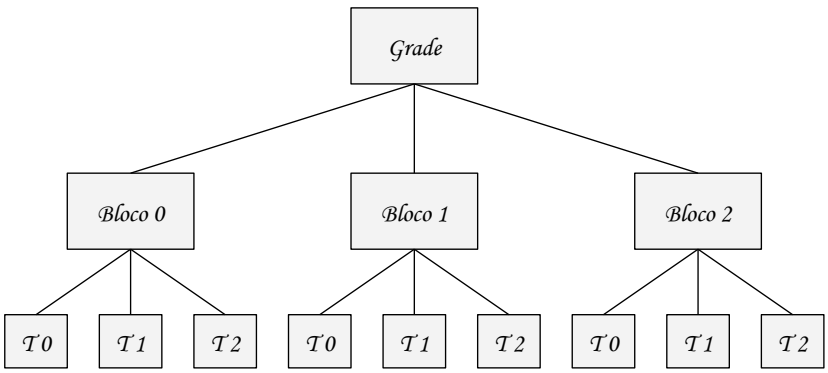


Figura 3.1: Estrutura de grade, bloco e threads da GPU

As threads pertencentes ao mesmo bloco compartilham uma área de memória que pertence ao processador. Essa área compartilhada é útil em algumas aplicações e pode ajudar a resolver certos problemas de dependência de dados para execuções diferentes, pois permite a comunicação entre as threads com custo zero. Por outro lado, a comunicação entre threads de blocos diferentes exige a gerência da CPU, o que pode tornar essa comunicação extremamente custosa para o desempenho do código (Cook, 2013).

Por conta dessas características da GPU, o correto mapeamento do problema em grade, blocos e threads deve ser pensado a fim de tirar o máximo proveito do paralelismo disponível das GPU's (Cook, 2013).

3.2 Arquitetura da GPU

Para obter o melhor desempenho do poder de processamento das placas de vídeo, é importante entender o funcionamento de sua arquitetura. Embora as placas gráficas possuam elementos nominalmente iguais à arquitetura das processadores de propósito geral, as suas relações e funcionamento são diferentes. E, é de suma importância o conhecimento deste funcionamento para que se possa tirar o melhor proveito do que elas têm a oferecer.

3.2.1 Taxonomia de Flynn

A taxonomia de Flynn é um modelo de classificação arquitetural de computadores. Desde a sua proposição em 1966 por Michael J. Flynn, ela vem sendo utilizada e atualizada conforme a evolução do hardware. Seu objetivo é classificar as diferente abordagem dos processadores em relação às instruções e os dados que são computados. Com os processadores multi cores e placas de vídeos a taxonomia Flynn ganhou uma complexidade maior nos últimos anos. Porém, neste trabalho foram discutidas as classificações existentes na taxonomia de Flynn relevantes ao trabalho desenvolvidos.

- **MIMD (Multiple Instructions Multiple Data)** - Esta arquitetura faz parte do funcionamento dos processadores multi cores modernos. Cada core possui uma independência sobre as instruções e dados a serem acessados pelos diversos processos executados pelo ele, em processadores multi tarefa existe a necessidade de trocas de contexto entre os processos em execução, o que pode acarretar um overhead caso existam muitos processos em execução (Cook, 2013).

- **SIMD (Single Instruction Multiple Data)** - Esta arquitetura trata dos processadores vetoriais, uma mesma instrução de ser executada para dados distintos (Cheng et al., 2013). Essa característica faz da arquitetura SIMD uma excelente escolha quando se trabalha com problemas onde a mesma instrução deve ser aplicada sobre um conjunto de dados, como o problema de aplicação de filtros em processamento de imagem, pois as imagens são estruturas vetoriais que favorecem o paralelismo.

  Originalmente essa arquitetura era a arquitetura das placas de vídeo. Com a utilização das placas de vídeo para computação de propósito geral, foram feitas alteração no hardware desses processadores. Desta forma, as placas de vídeo atuais trabalham com conjuntos de 1024 threads por bloco, onde threads de um mesmo bloco executam a mesma instrução, essa abordagem fez surgir uma nova classificação conhecida como *SIMT (Single Instruction Multiple Threads)* (Cook, 2013).

3.2.2 Arquitetura Kepler

Em 2012, a NVIDIA anunciou a arquitetura Kepler. Dentre as inovações que esta trouxe destaca-se a execução de kernels concorrentes proporcionando que a CPU dispare mais de uma chamada a GPU. Isto permite que diferentes códigos de GPU sejam executados concomitantemente, ainda que estejam em processos distintos (NVIDIA, a; Zamith and Clua, 2015). Uma outra inovação apresentada por esta arquitetura, foi a chamada recursiva de kernels, ou seja, um kernel pode lançar outro kernel sem transferir o contexto para a CPU, esta característica é chamada de paralelismo dinâmico, e permite ao programador uma maior flexibilidade no uso da GPU (Zamith and Clua, 2015).

3.2.3 Arquitetura Maxwell

Em 2014, a NVIDIA lançou a arquitetura Maxwell, que trouxe algumas melhorias em relação a arquitetura anterior, tanto em desempenho de cores quanto de memória. Dentre os avanços desenvolvidos, a eficiência energética teve destaque, pois a quantidade de operações de ponto flutuante por watts dobrou

em comparação com a arquitetura Kepler (Zamith and Clua, 2015; NVIDIA, b).

3.3 Hierarquia de Memória

As GPU's contam como uma hierarquia de memória semelhante a das CPU's em alguns pontos. Desde do surgimento do CUDA, as GPU's contam com memória principal ou RAM, memória de textura, dois níveis de cache, memória de constantes e memória compartilhada (Hennessy and Patterson, 2011).

As principais diferenças entre as memórias da CPU e GPU são em relação a latência e capacidade. A seguir será apresentado um pequeno resumo dos principais tipos de memórias utilizadas na para GPU's: (i) Memória global ou a memória RAM. Essa memória pode ser acessada por qualquer thread a qualquer momento. É uma memória com baixa latência e sua gerência fica sob responsabilidade da CPU. (ii) Memória de textura, em geral usada para computação gráfica. Como é uma memória apenas de leitura, sua latência é menor que a memória global. Para a computação de propósito geral, os dados devem ser modelados como texturas. Além disso, as threads fazem acesso apenas de leitura, podendo acessar qualquer dado da textura. (iii) Memória de constantes. O CUDA armazena dados constantes nesta memória que é apenas de leitura. Sua latência é semelhante a da memória de textura. Qualquer thread pode acessar qualquer posição da memória de constantes. (iv) Memória compartilhada tem escopo de bloco e sua latência é semelhante a dos registradores. Como tem escopo de bloco, apenas as threads de um mesmo bloco podem acessar os dados armazenados nessa memória. É uma memória utilizada para fazer a comunicação entre as threads. (v) Memória local tem escopo da thread, é nessa memória que as variáveis locais são armazenadas e sua latência é de registradores. Vale ressaltar como a memória local é alocada nos registradores, há um limite de variáveis que podem ser declaradas. Quando esse limite é ultrapassado, as variáveis passam a ser armazenadas na memória global, o que aumentando o tempo de acesso.

Uma diferença interessante nas hierarquias de memória da CPU e GPU é a organização dos processos e a forma de acessar os recursos dos disponíveis. A CPU executa muitas threads e faz diversas trocas de contexto tendo que armazenar os valores presentes nos registradores para depois restaurá-los quando necessário, isso não acontece na GPU, pois, não se faz necessário realizar trocas de contexto. (Cook, 2013; Zamith and Clua, 2015).

Capítulo 4

Redes Neurais Artificiais

4.1 Definição

Rede Neural Artificial (RNA) é uma classe de algoritmo de aprendizado de máquina baseado no funcionamento do cérebro humano que, por sua vez, funciona a partir de uma rede de neurônios. Os neurônios fazem conexões trocando informações, através de sinapses, e, analogamente, as redes neurais artificiais são compostas de unidades de processamento que realizam algum cálculo e trocam dados através de conexões similares as sinapses (Processing and Recognition, 1999).

As primeiras propostas de redes neurais artificiais surgiram entre os anos de 1950 e 1960, onde os principais cientistas de tal estudo são Warren McCulloh e Walter Pitts (Nielsen, 2016). Inspirado pelo trabalho destes, Frank Rosenblatt desenvolveu a ideia do neurônio artificial chamado *perceptron* que serve de inspiração para os modelos de neurônios utilizados atualmente em diversas arquiteturas de redes neurais (Nielsen, 2016). Como já foi dito o cérebro é a inspiração para as RNA's, ele funciona disparando sinais elétricos, conhecidos como sinapses, através da combinações de neurônios. Logo, o cérebro possui milhões de neurônio que são capazes de realizar bilhões de conexões, e esses neurônios pertencem a um sistema que vem evoluindo a centenas de milhares de anos e, portanto, desenvolver um sistema tão preciso quando esse não é uma tarefa simples (Nielsen, 2016).

As redes neurais vêm conseguindo se destacar no campo do aprendizado de máquina pela sua capacidade de aprender padrões em diversos tipos de problemas (LeCun et al., 1998). O grande sucesso das RNA's se deu em parte graças a sua parceria com o gradiente descendente como método de aprendizagem (LeCun et al., 1998). Nos últimos anos são vários os trabalhos que foram desenvolvidos com soluções utilizando rede neurais artificiais como: detecção de dígitos (LeCun et al., 1995, 1998), classificação de imagens em alta definição (Krizhevsky et al.), composição de música (Eck and Schmidhuber, 2002), reconhecimento de voz (MOHAMMAd et al., 2014) e detecção de rostos em imagens (Rowley et al., 1998).

Os computadores funcionam executando algoritmos formados por sequências de passos que transformam entradas em saídas determinísticas. Porém, existem problemas para os quais não existem algoritmos diretos para resolvê-los, pois todos os fatores envolvidos não são claros, um bom exemplos disso são os preços de imóveis de uma cidade. Logo, pode-se imaginar variáveis que influenciam nos valores como o m^2, o bairro, a localidade e os acessos a transportes, mas não podemos definir esses fatores como absolutos. Contudo, nosso cérebro, a partir de experiências passadas, consegue muitas vezes estimar um preço socialmente aceitável para tal problema. Portanto, pode-se dizer que as redes neurais artificiais são uma tentativa de propor soluções para problemas deste tipo, problemas que precisam ser aprendidos (Kriesel, 2013).

4.2 *perceptron*

O *perceptron* foi o primeiro modelo de representação do neurônio presente em redes neurais artificiais (Nielsen, 2016). Seu funcionamento é bem simples, na figura 4.1 é apresentado o modelo visual do *perceptron*. Ele funciona recebendo entradas de valores binários como: x_1, x_2, x_3, ..., x_n e produz uma saída binária y onde, cada conexão feita pelas entradas possui um valor real associado, w_1, w_2, w_3, ..., w_n normalmente chamados de pesos. Logo, os pesos representam a força daquela entrada, onde a saída y é formada pela aplicação de uma regra simples conforme apresentado na equação 4.1 (Nielsen, 2016).

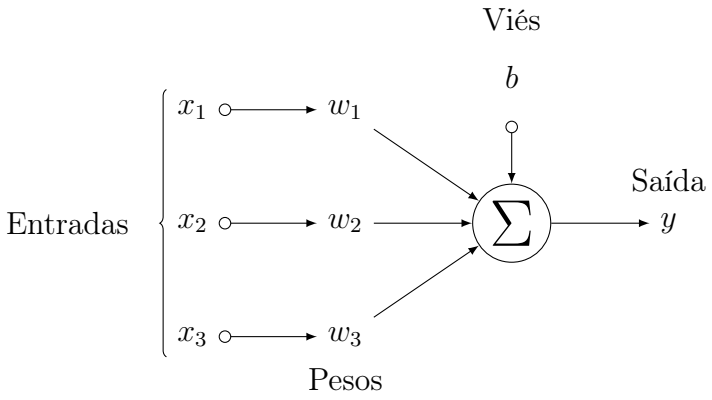


Figura 4.1: Perceptroon

$$f(x) = \begin{cases} 0, & \text{se} \sum_{j=0}^{n} w_j \times x_j < \text{threshold} \\ 1, & \text{se} \sum_{j=0}^{n} w_j \times x_j \geq \text{threshold} \end{cases} \tag{4.1}$$

$$f(x) = \begin{cases} 0, & \text{se} \sum_{j=0}^{n} w_j \times x_j + b < 0 \\ 1, & \text{se} \sum_{j=0}^{n} w_j \times x_j + b \geq 0 \end{cases} \tag{4.2}$$

Posteriormente a equação 4.1 foi redefinida para a equação 4.2 onde $b = -\text{threshold}$, o operador b comumente é chamado de viés, essa redefinição trouxe um novo fator para a equação do *perceptron*, um fator linear, que pode ser interpretado como um grau de facilidade com que determinada entrada é capaz de ativar o *perceptron*, tal que quanto maior o valor de b mais fácil é para aquela entrada ativar o neurônio.

Um *perceptron* pode não fazer muita coisa, porém vários *perceptrons* interconectados, como mostrado na figura 4.2, fizeram parte das primeiras redes neurais propostas. Nesse contexto, as entrada podem ser interpretadas como afirmações, sobre determinada questão, e a saída como sendo uma reação ou consequência que representa as entradas em conjunto. Logo, um conjunto maior de *perceptrons* pôde abstrair e expressar um conjunto complexo de informações e produzir uma saída adequada (Nielsen, 2016).

Observando o funcionando do *perceptron*, a saída produzida por ele depende dos pesos e do threshold, já que as entradas são informações externas ao mesmo. Dessa forma, as decisões dependem desses valores. Por exemplo, as entradas podem re-

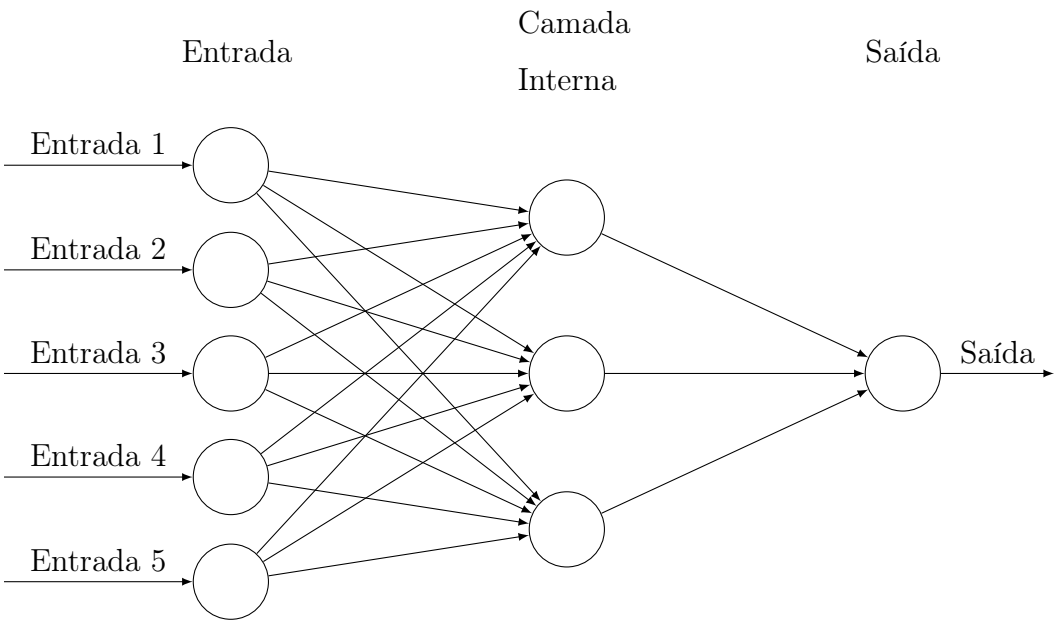


Figura 4.2: Rede neural de *perceptrons*

presentar notas de um aluno, os pesos são literalmente os pesos que valem cada avaliação e os threshold a média de aprovação, uma analogia similar pode ser feito com o viés já que as equações 4.2 e 4.1 são equivalentes. Contudo, como afirma Nielsen (2016), este não é o modelo completo de funcionamento do cérebro humano, mas, ainda assim, é capaz de reproduzir algumas tomadas de decisões bem complexas.

4.3 Neurônios Sigmoid

A saída de um neurônio artificial representa o quando suas entradas, combinadas com os pesos associados as suas conexões, o ativaram. O neurônio biológico, segundo Kriesel (2013), está sempre ativo de alguma forma, já os *perceptrons* possuem apenas dois estados, ativo, quando sua saída é 1, e inativo, quando sua saída é 0. As funções de ativação presentes nas equações 4.1 e 4.2 funcionam bem para problemas com variáveis binárias e com decisões finais também binárias, porém para certos problemas, principalmente com várias classes de classificação final, é interessante que pequenos ajustes nas variáveis da rede neural influencie de forma proporcional na saída da rede, isso não acontece em redes formadas por *perceptrons*, pois, como

eles possuem apenas dois tipos de saídas uma pequena mudança em um peso pode alterar totalmente o resultado final (Nielsen, 2016).

Os neurônios sigmoid e os *perceptrons* possuem diversas características similares, como um conjunto de entradas, um conjunto de pesos associados a cada ligação feita entre as camadas da rede e uma saída que representa o processamento desses elementos. A diferença está na forma como este processamento é realizado. Nos neurônios sigmoid as entradas não são estritamente binárias, isto é, elas podem assumir valores reais, e a função de ativação, como o nome do neurônio sugere, é uma função sigmoid, conforme representada na equação 4.5, posteriormente outras funções de ativação se tornaram comuns como a função de tangente hiperbólica, equação 4.4, que proporciona uma saída entre -1 e 1. Essas funções tornam a ativação do neurônio mais suave além de serem funções matematicamente convenientes para o método de aprendizado com gradiente descendente.

$$z(x) = \sum_j w_j \times x_j + b \tag{4.3}$$

$$a(x) = \frac{e^{-z(x)} - e^{-z(x)}}{e^{-z(x)} + e^{-z(x)}} \tag{4.4}$$

$$a(x) = \frac{1}{1 + e^{-z(x)}} \tag{4.5}$$

4.4 Arquitetura de uma Rede Neural

Segundo Processing and Recognition (1999) uma Rede Neural possui os seguintes elementos básicos: (i) Um conjunto finito de neurônios agrupados em camadas, sendo a primeira camada a entrada dos dados, uma ou mais camadas internas (escondidas) e uma camada de saída da rede; (ii) um conjunto finito de conexões entre os neurônios, onde cada conexão é associada a um valor que representa a força daquela conexão, que é aprendido pela rede; (iii) uma função de propagação que diz como são feitos os cálculos em cada camada de neurônios; (iv) uma função de ativação

que faz uso da função de propagação para gerar a saída dos neurônios.

Não se vê muita modificação no *design* das camadas de entrada e saída das redes neurais, porém é nas camadas escondidas onde os diversos tipos de arquiteturas surgiram, com a intenção de melhor representar e solucionar os problemas que determinada rede se propõe a resolver (Nielsen, 2016).

4.4.1 *Feed Forward*

As redes do tipo *Feed Forward* são as mais utilizadas e mais simples, sua ideia é que as camadas propaguem as informações sempre em um sentido partindo da camada de entrada em direção a camada de saída e nunca no sentido contrário (Nielsen, 2016). Uma das razões para o sucesso das redes *feed forward* é a simplicidade da sua implementação e os bons resultados obtidos nos últimos anos (Nielsen, 2016), a figura 4.2 apresenta uma rede neural *feed forward*.

4.4.2 Rede Recorrentes

As redes neurais recorrentes embora não possuam a mesma influência em trabalhos como as redes *feed forward* são redes que tem maior proximidade com o tipo de conexão observadas no cérebro humano. Sua principal diferença é a capacidade da saída de um neurônio servir de entrada em uma camada anterior, situação que não é permitida nas redes *feed forwards* (Nielsen, 2016; Kriesel, 2013). Por essa característica, os neurônios das redes recorrentes podem fazer conexões com neurônios de camadas anteriores, com isso, pode acontecer desses neurônios se auto influenciarem, pois o valor que compõem a sua saída pode ser transmitido de volta e fazer parte do valor que será enviado como entrada para ele mesmo, como ilustrado na figura 4.3. Uma variação das redes neurais recorrentes pode ser vista na figura 4.4, onde as conexões ocorrem em neurônios de uma mesma camada, Kriesel (2013) define este tipo de rede como Rede Recorrente Lateral.

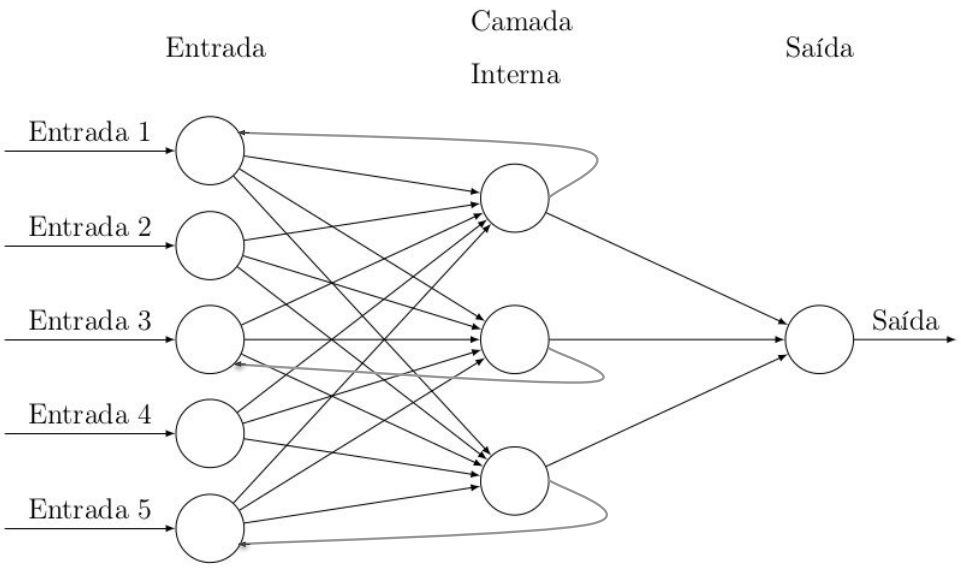


Figura 4.3: Rede neural Recorrente

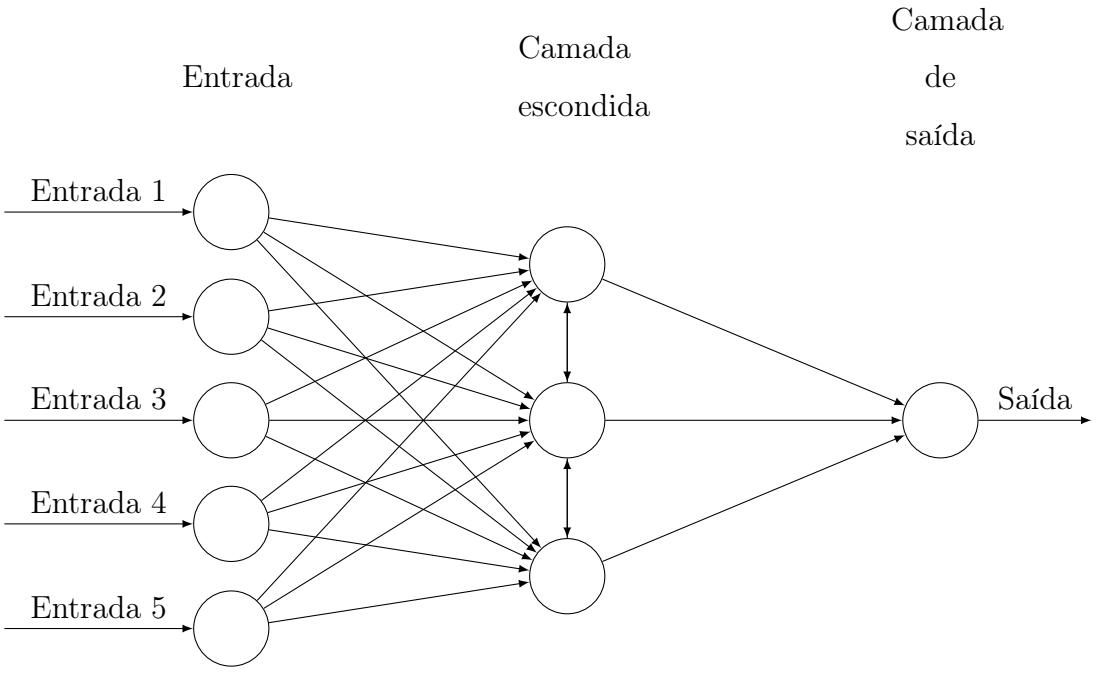


Figura 4.4: Rede neural Recorrente Lateral

4.5 Redes Convolucionais

Redes neurais convolucionais são redes idealizadas com o objectivo de classificar imagens e, para isso, faz uso de conexões específica que buscam tirar vantagem da estrutura espacial de imagens. Sua arquitetura se encaixa em uma rede *feed forward* (Nielsen, 2016). As principais característica das redes neurais convolucionais são as operações de convolução e *pooling* em suas camadas, diferente das redes *feed forward* convencionais totalmente conectadas, as redes convolucionais buscam aprender através de conexões feitas por conjuntos próximos de *pixels*, fazendo com que cada neurônio lide com uma pequena parcela da informação da imagem. Segundo Nielsen (2016); LeCun et al. (1998); CS231n isso acelera o aprendizado além de diminuir o número de parâmetros da rede.

4.5.1 Operação de Convolução

A operação de convolução pode ser considerada um filtro que é aplicado em diferentes regiões da imagem, os valores desse filtro formam os pesos das conexões dos neurônios da rede; cada neurônio está associado a uma pequena matriz de filtro/ pesos que são aplicados em uma região da imagem gerando um valor de saída. Essa operação faz com que os neurônios da camada de convolução não sejam totalmente conectados com os neurônios das camadas vizinhas (LeCun et al., 1998; CS231n).

Em uma rede neural *feed forward* convencional totalmente conectada todos os neurônios de camadas vizinhas conectam-se, porém quando as entradas representam uma imagem essa pode não ser a melhor abordagem, pois, isso faz com que *pixels* de regiões extremas da imagem tenham o mesmo grau de influência através das conexões, mas é razoável pensar que pixels mais próximos compartilham um contexto maior de correlação se comparados com pixels muitos distantes (Nielsen, 2016). Segundo Nielsen (2016) essas relações, de pouca influência entre pixels distantes, podem ser aprendidas em uma rede totalmente conectada, mas ao limitar tais conexões as redes convolucionais podem aprender mais rápido e com menor número de parâmetros.

Na figura 4.5 é possível ver uma representação de como um neurônio de uma rede convolucional opera sobre uma imagem. Um ponto importe sobre a operação de convolução é que a submatriz que serve como filtro para um neurônio pode ser compartilhada por todos os neurônios de uma mesma camada, porém cada neurônio opera em partes diferentes da imagem, fazendo com que todos esses neurônios reconheçam, através deste filtro, o mesmo padrão em partes distintas da imagem (Nielsen, 2016).

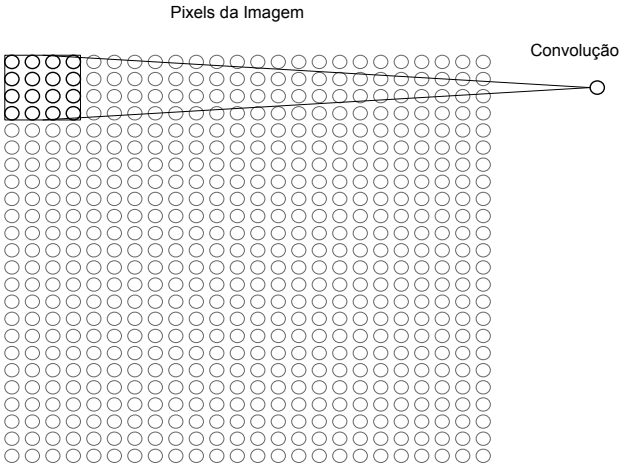


Figura 4.5: Representação da operação convolução de redes neurais convolucionais (Imagem alterada de (Nielsen, 2016))

Segundo Nielsen (2016) ao aplicar o mesmo filtro em partes diferentes da imagem é possível detectar o mesmo padrão, mas pode-se ir além e detectar mais de um padrão em uma mesma camada da rede neural, basta aplicar outro(s) filtro(s) e produzir diferentes saídas, como é ilustrado na figura 4.6, permitindo que uma mesma camada possa detectar mais de um tipo de padrão, por exemplos, podemos ter uma camada que detecta linhas horizontais, linha verticais e linha diagonais. Outro detalhe é que esses filtros podem ser calculados de forma independentes um do outro.

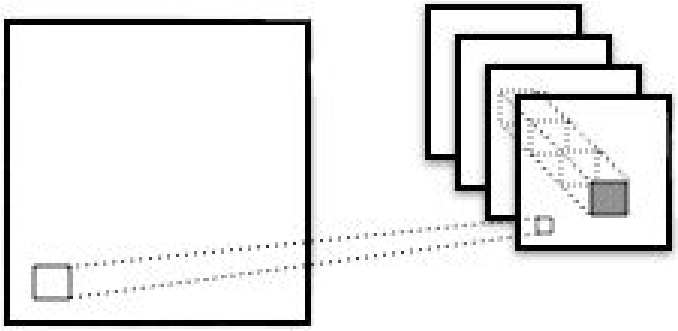


Figura 4.6: Representação da operação convolução com 4 filtros (Imagem alterada de (lab))

4.5.2 Operação de *pooling*

Em redes neurais convolucionais é comum a aplicação de uma função de *pooling* logo após a realização da operação de convolução, esta operação tem como objectivo simplificar a informação gerada por cada saída da operação de convolução, além de diminuir o número de parâmetros para as camadas posteriores, esta operação ajuda a abstrair um pouca às informações da imagem original. Portanto, para cada resultado gerado a partir dos filtros de convolução ocorre uma diminuição de dimensionamento como pode visto na figura 4.7 onde quatro valores são resumidos em um (Nielsen, 2016).

Existem algumas função de *pooling* como: (i) Max-pooling que consistem em retornar apenas o maior valor dentro um conjunto de valores que serão resumidos, (ii) L2 pooling consiste em uma operação de raiz quadrada do somatório dos valores a serem resumidos e (iii) a operação de média consistem em realizar a média aritmética dos valores (Nielsen, 2016).

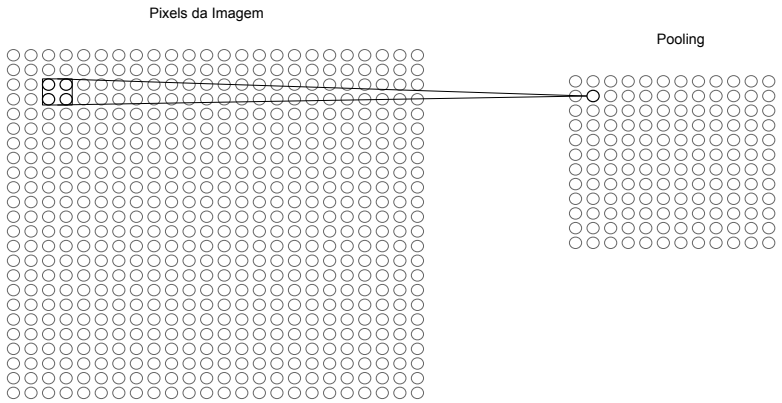


Figura 4.7: Representação da operação de *pooling* (Imagem alterada de (Nielsen, 2016))

4.6 *Backpropagation*

Segundo Nielsen (2016) *backpropagation* foi introduzido pela primeira vez nos anos de 1970, mas ganhou reconhecimento em 1986 em um trabalho desenvolvido por David Rumelhart, Geoffrey Hinton e Ronald Williams, mostrando que o *backpropagation* era um método de aprendizado rápido e que conseguia resolver problemas antes considerados impossíveis. Hoje, ele é com certeza o método mais utilizado em redes neurais artificias (Nielsen, 2016).

Como visto ao longo deste capítulo as redes neurais dependem dos valores associados as ligações que os neurônios fazem, que são chamados de pesos e viés, esses são os valores que devem ser aprendidos pelas redes neurais. O *backpropagation* é um algoritmo de gradiente descendente que busca encontrar os valores que consigam generalizar a entrada em relação a classe(saída correta) que a ela pertença (Nielsen, 2016; Kriesel, 2013).

4.6.1 Função de Custo

Quando se utiliza o *backpropagation* como algoritmo de aprendizado de uma rede neural artificial, é necessário ter uma função de custo que medirá o quão ajustado os parâmetros da rede estão em relação ao problema. A função de erro quadrático

médio 4.6 é a função de custo mais utilizada na literatura Nielsen (2016); LeCun et al. (1998); Nixon and Aguado (2008); Sebaa et al. (2006).

$$C = \frac{1}{2n} \sum_{x} \|y(x) - a(x)\|^2 \tag{4.6}$$

Na equação 4.6 n representa a quantidade de exemplos que a rede possui, $y(x)$ representa a classe a qual exemplo x pertence, $a(x)$ é a classe a qual a rede neural encontrou para o exemplo x. A função de erro quadrático médio calcula a média do erro dos valores encontrados pela rede em relação a classe correta das entradas disponíveis (Nielsen, 2016).

4.6.2 Aprendizado

O coração do *backpropagation* está em usar a função de custo para aprender e ajustar os melhores valores para os pesos e viés da rede neural. Ele faz isso calculando as derivadas parciais da função de custo em relação a todos os parâmetros que influenciam na classificação final da rede, $a(x)$ na equação 4.5 (Nielsen, 2016). Para cada camada da rede neural, deve-se calcular um valor de erro para cada parâmetro dos neurônios.

Deseja-se que a função de custo 4.6 seja o mais próximo possível de 0 e por isso a derivada parcial é utilizada, pois a derivada representa a taxa de variação de uma função. Por tanto, calculando esses valores podemos determinar o quanto cada parâmetro está influenciando na previsão da rede neural e corrigir os valores a partir desse erro, que é dado pelas equações 4.7 e 4.8 (Nielsen, 2016).

$$\Delta w^L = \frac{\partial C}{\partial w^L} \tag{4.7}$$

$$\Delta b^L = \frac{\partial C}{\partial b^L} \tag{4.8}$$

Nas equações acima os termos Δw^L e Δb^L representam o erro do peso w na camada L e o erro do viés b na camada L respectivamente. Como a saída de uma

camada serve de entrada na camada subsequente e os valores são submetidos a funções de propagação e funções de ativação, a aplicação da regra de cadeia a partir da saída voltando até os primeiros parâmetros é fundamental para realizar todos os cálculos necessários para se determinar os correspondentes erros de cada parâmetro (Nielsen, 2016).

Essa ideia dos erros serem calculados no sentido contrário ao fluxo de propagação da rede dá o nome *backpropagation* a esse método, pois os erros encontrados pela função de custo são propagados de trás para frente com a intenção de corrigir os valores dos parâmetros da forma apresentada pelas equações 4.9 e 4.10, onde η corresponde a uma parâmetro de velocidade de aprendizado, que deve ser ajustado de forma a tentar acelerar o aprendizado.

$$w^L = w^L - \eta \times \Delta w^L \tag{4.9}$$

$$b^L = b^L - \eta \times \Delta b^L \tag{4.10}$$

Capítulo 5

Experimentos

Neste trabalho foi desenvolvido uma rede neural convolucional para a classificação de dígitos do dataset MNIST utilizando GPU para acelerar o processo de aprendizagem. A rede neural proposta é baseada na Lenet5 presente no trabalho de LeCun et al. (1998), que também fez uso do mesmo conjunto de dados.

A utilização de GPU para acelerar o aprendizado de redes neurais é uma técnica bastante explorada nos últimos anos, diversos trabalhos foram desenvolvidos utilizando placas gráficas como: Johnson (2015); Krizhevsky et al.; Nielsen (2016) dentre outros. Junto com a GPU, foi utilizada outra ferramenta para auxiliar no desenvolvimento da construção da rede neural, a biblioteca Theano, que auxilia tanto na parte matemática quanto na comunicação com a GPU.

5.1 Biblioteca Theano

Theano é uma biblioteca de código aberto criada em 2007 pela universidade de Montreal com um foco em programação científica, desenvolvida em python, ela permite a otimização e paralelização de funções e expressões matemáticas, principalmente de álgebra linear. A paralelização pode ser feita tanto na CPU quando na GPU de forma transparente para o desenvolvedor, sem muitas mudanças na implementação (Theano Development Team, 2016).

Dentre as principais características da Theano tem-se (i) uma fácil compatibilidade com as biblioteca numpy e scipy, outras biblioteca científica para python, (ii) um nível maior de abstração na paralelização de códigos para a GPU, (iii) a realização de operações de diferenciação de forma rápida e eficiente, o que auxilia no desenvolvimento de soluções que fazem uso desse tipo de aprendizado e (iv) a geração de código dinâmico de python para a linguagem C, requisito para a aceleração de operações matemáticas tanto para CPU quanto para GPU. Através de uma representação de variáveis simbólica a biblioteca Theano consegue otimizar expressões matemáticas, o que pode acelerar ainda mais o desempenho do código proposto. Esse tipo implementação é conhecida como avaliação atrasada (lazy evaluation) (Theano Development Team, 2016).

Por conta da forma como a biblioteca Theano transforma a implementação em uma aplicação que será executada em paralelo em placas gráficas, o programador perde um pouco do controle em relação a alguns parâmetros vistos no capítulo 3 como quantidade de blocos e de threads que serão alocadas, bem como quando os dados serão transferidos da memória RAM para a memória da GPU. Na documentação da biblioteca é descrito um tipo de variável simbólica chamada shared (compartilhada) que é transferida para a memória da GPU, quando existe uma placa disponível, esse é um recurso útil quando se deseja otimizar a execução da aplicação, mesmo sem o controle total dessas operações (Theano Development Team, 2016).

Nos últimos anos diversas bibliotecas similares surgiram com características próximas as da Theano como: Tensor Flow do Google, Caffe, Keras, CNTK da Microsoft, dentre outras. Todas elas possuem suas facilidades e objetivos, mesmo assim todas essas de alguma forma oferecem uma forma de executar código em paralelo nas GPU's.

5.2 Base de Dados

A base de dados utilizada neste trabalho foi o MNIST, que foi apresentado pela primeiro ver por LeCun et al. (1998), ele foi construído a partir de uma base de dados

maior chamado NIST. O NIST por sua vez é formado por dois conjuntos de imagens de dígitos, números de 0 a 9, escritos a mão por mais de 500 pessoas diferentes, cada um desses conjuntos formam os conjuntos de treino e teste do NIST, porém como cada conjunto desse veio de grupos de escritores distintos, isso poderia enviesar os resultados, uma vez que no conjunto de treino não tinha nenhuma exemplo de caligrafia dos escritores do conjunto de texto. Por esse motivo, uma nova base de dados foi proposta por LeCun et al. (1998) misturando-se as imagens de ambos os conjuntos nas bases de treino e teste do MNIST.

O MNIST é composto por 60 mil imagens de treino e 10 mil imagens para teste, todas as imagens são binárias e tiveram seus tamanhos normalizados para 20x20 pixels e posteriormente foram centralizadas para ocupar um tamanho de 28 x 28 pixels (LeCun et al., 1998). Este tipo de padronização facilita muito e já pode ser considerada como parte da etapa pré-processamento necessário na maioria dos trabalhos de visão computacional.

LeCun et al. (1998) em um dos trabalhos mais referenciados sobre reconhecimento de dígitos apresenta uma tabela, que foi adaptada na figura 5.1, com diversas técnicas que já foram usadas para tentar solucionar o problema de reconhecimento de dígitos utilizando a base de dados MNIST.

5.3 Métricas

As métricas utilizadas neste trabalho podem ser separadas em dois grupos, um para medir a desempenho da rede neural em relação a aprendizado e acertos de classificação e outro em relação ao speedup e desempenho da GPU's. Em relação ao primeiro grupo de medidas nós medimos (i) a acurácia total dos conjunto de teste e treino ao longo das épocas de treino; (ii) o macro F1 score, que consegue juntar o F1 de todas em classes em um único valor (iii) e a matriz de confusão final, que é um quadro final de previsões para todas as instâncias de treino e teste onde pode-se ver tanto a quantidade de erros quanto a de acertos. As fórmulas utilizadas para o macro F1 score podem ser vistas nas equações 5.1, 5.2 e 5.3. Já o speedup é a razão

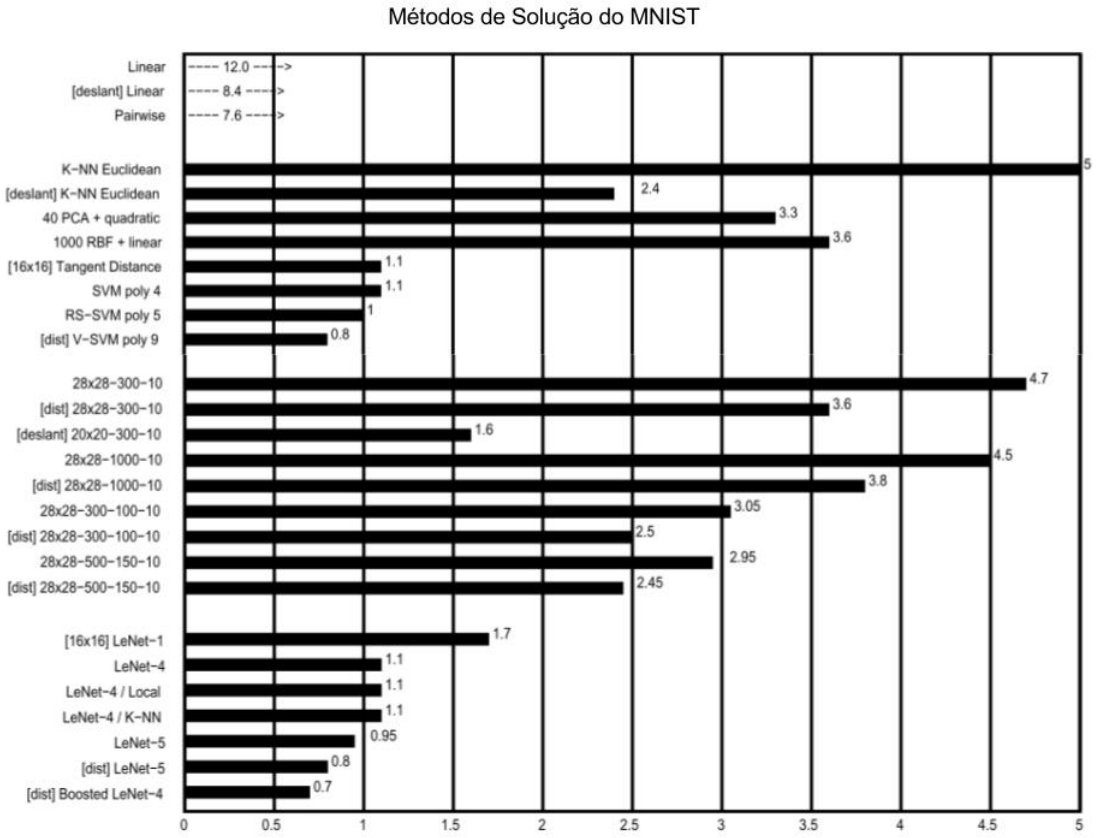


Figura 5.1: Erro médio dos Métodos que foram aplicados ao MNIST (Imagem alterada de (LeCun et al., 1998)).

entre o tempo sequencial e o tempo do paralelo, indicando quantas vezes a execução em paralelo é mais rápido.

$$MMP = 2 \times \frac{\sum_n VP_i}{\sum_n VP_i + FP_i} \tag{5.1}$$

$$MMR = 2 \times \frac{\sum_n VP_i}{\sum_n VP_i + FN_i} \tag{5.2}$$

$$MF1 = 2 \times \frac{MMP + MMR}{MMP \times MMR} \tag{5.3}$$

onde,

VP : verdadeiro positivo

VN : verdadeiro negativo

FP : falso positivo

FN : falso negativo

MMP : Média macro da precisão

MMR : Média macro do recall

$MF1$: Macro do F1

5.4 Implementação

Neste trabalho foi implementado uma rede neural convolucional baseada na Lenet5. Ela foi implementada em python3 utilizando as bibliotecas Theano e numby. A rede desenvolvida possui 6 camadas como pode ser visto na figura 5.2, os dados de entrada são imagens da base de dados MNIST que foram redimensionadas de 28x28 *pixels* para 32x32 *pixels*.

Na primeira camada é aplicada uma operação de convolução com um tamanho de 5x5 *pixels*, ou seja, cada neurônio aplica um filtro de tamanho 5x5 em um recorte da imagem, como as dimensões da imagem de entrada são de 32x32 *pixels* são necessários 784 neurônios para aplicar tal operação em uma janela de 5x5 *pixels*, como visto no capítulo 4, é possível compartilhar o mesmo filtro através dos neurônios e aplicar mais de um filtro numa mesma camada, por isso foram aplicadas nesta camada 6 filtros de convolução diferentes, que geram como saída um cubo de 6x28x28.

Após a aplicação da operação de convolução e de geradas as 6 saídas com dimensões 28x28 *pixels*, é aplicado a operação de *pooling* para resumir os valores encontrados, a função escolhida foi a de *max pooling* com uma janela de aplicação de 2x2 *pixels* o que reduz pela metade as dimensões de cada uma das saídas geradas pela operação de convolução anterior.

Como pode ser vista na figura 5.2, o mesmo processo dos parágrafos anteriores se repete, com variação no aumento na quantidade de filtros de convolução aplicados terminando com saída com as dimensões 16x5x5. As próximas duas camadas são

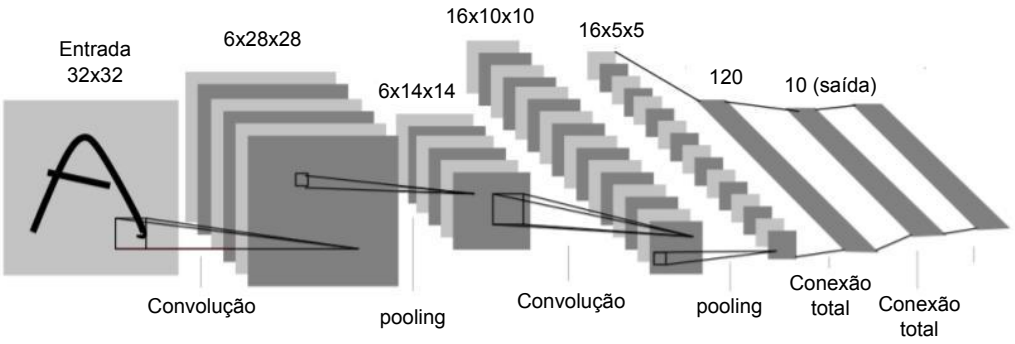


Figura 5.2: Representação da rede neural convolucional implementada. Imagem alterada de (LeCun et al., 1998)

compostas de neurônios totalmente conectados que geram 10 saída, essas 10 saída correspondem a probabilidades para cada uma das 10 classes de números (0 até 9), que as imagens do MNIST podem assumir. Essas probabilidade são geradas pela função de softmax descrita na equação 5.4

$$p(C_k|\mathbf{x}) = \frac{e^{\ a_k(\mathbf{x})}}{\sum_{j=1}^{K} e^{\ a_j(\mathbf{x})}} \tag{5.4}$$

Na equação de softmax 5.4 $p(C_k|\mathbf{x})$ representa a probabilidade de um dados x pertencer a classe C_k em um universo de K classes onde $a_k(x)$ representa a k-ésima saída da rede neural dado a entrada x

Como parâmetros da rede neural foram utilizados 30 épocas de treino com um rate de aprendizado com valor de 0.3, este valor foi encontrado depois vários teste, e este foi o que gerou os melhores resultado. No treino da rede neural foi utilizado o gradiente descendente estocástico, com pacote de 200 imagens, essa é uma forma de manter um baixo custo de execução que ao mesmo tempo melhora a generalização da rede fornecendo poucos exemplos a cada iteração para evitar que a rede decorre as características das imagens de treino. Como função de custo foi escolhida a função de entropia cruzada (*cross entropy*), que segundo Nielsen (2016) é uma função que consegue se ajustar mais rapidamente quando a rede comete um erro grande e com isso aprende de forma mais rápida. A inicialização dos pesos e viés da rede neural

foram baseados no capítulo 3 do livro (Nielsen, 2016) onde são escolhidos valores aleatórios que possuem um distribuição normal de valores com média 0 e desvio padrão 1.

Na equação 5.5, onde é calculado o custo da rede neural. n é a quantidade de amostras fornecidas para a rede neural e $a_j(x)$ é probabilidade encontrada pela rede neural para a classe j, que é a classe correta para a entrada x, por tanto, é desejável que esse valor se aproxime ao máximo de 1 fazendo o somatório do *log* tender a 0.

$$c = \frac{-1}{n} \times \sum_{n} \log a_j(x) \tag{5.5}$$

A cada época de treino da rede os parâmetros de peso e viés são atualizados com a derivada parcial da função de custo 5.5 em relação a cada um dos parâmetros na forma descrita nas equações 4.9 e 4.10 no capítulo 4.

5.5 Resultados

Neste trabalho foi feito uso de duas placas gráficas para acelerar o processo de aprendizado das rede neural e a biblioteca Theano oferece suporte a geração de código para CUDA, que proporciona ganhos em tempo de execução.

As placas utilizadas foram as placas Tesla K40c e a GTX 980, ambas produzidas pela NVIDIA. A Tesla K40c é uma placa dedicada e da arquitetura Kepler. A GTX 980 é de uma arquitetura mais moderna (Maxwell), atinge uma frequência dos cores maior e taxa de transferência maior em relação a Tesla K40c. Por outro lado, a GTX 980 não é uma placa dedicada, compartilhando a tarefa de visualização com de computação de propósito geral. A tabela 5.1 resume as características de cada placa.

Quanto aos resultados obtidos tanto na CPU quanto na GPU em termos de acurácia conseguiram atingir a casa dos 99% e 98% ao final da etapa de treino, para ambos o conjuntos de dados em ambas as placas, mostrando que não houve perda quanto a qualidade do aprendizado como pode ser visto nos gráficos das Figuras

Característica/GPU	**Tesla K40c**	**GTX 980**
Arquitetura	Kepler	Maxwell
CUDA cores	2.880	2.048
Freq. core	745MHz	1.241 MHz
Freq. memória	3.004GHz	3.505GHz
RAM (Capacidade)	12GB	4GB
Cache L2	1,5MB	2MB
Banda passante	288GB/s	224GB/s

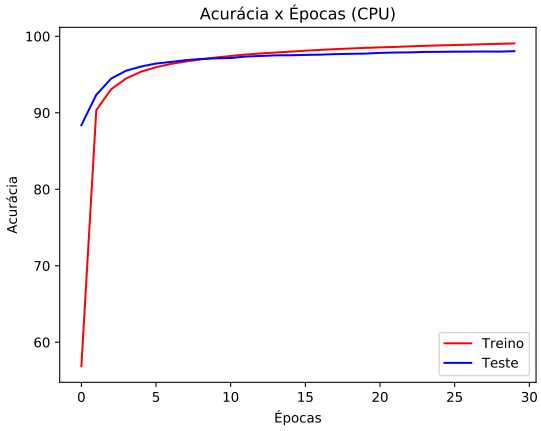
Tabela 5.1: Características das GPUs. (Tabela extraída de (Melo et al., 2017))

5.3a, 5.3c e 5.3e. As Figuras 5.3b, 5.3d e 5.3f apresentam o macro F1 score. E, as Figuras 5.3a, 5.3b, 5.3c, 5.3d, 5.3e e 5.3f mostram as matrizes de confusão de treino e teste ao final do treinamento.

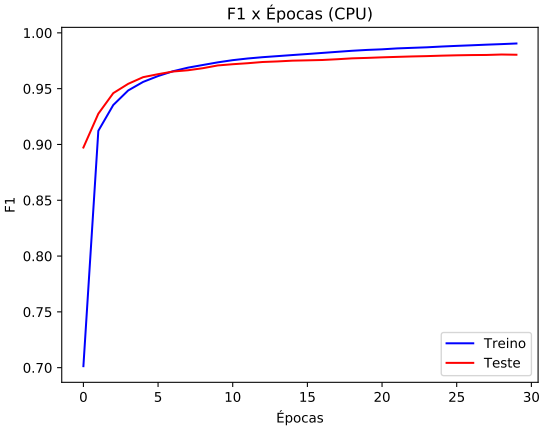
O ambiente para os testes consiste de duas CPUs, onde a primeira é máquina com a placa de vídeo GTX 980, como sistema operacional Linux Mint 18 64-bit com 8 GB de memória RAM, e processador Intel Core i5 3.30GHZ. A segunda máquina tem a placa gráfica Tesla K40c também, com sistema operacional Linux Mint 18 64-bit com 12 GB de memória RAM e processador Intel Core i7 3.40GHZ. A biblioteca Theano foi utilizada na versão 0.9 e versão do CUDA é 8.0.

Para a métrica de speedup, foi considerada a versão sequencial da execução do treinamento da rede neural em CPU com uma única thread. Foram duas versões paralelas, uma para cada GPU. A tabela 5.2 mostra o speedup alcançado com as GPU's.

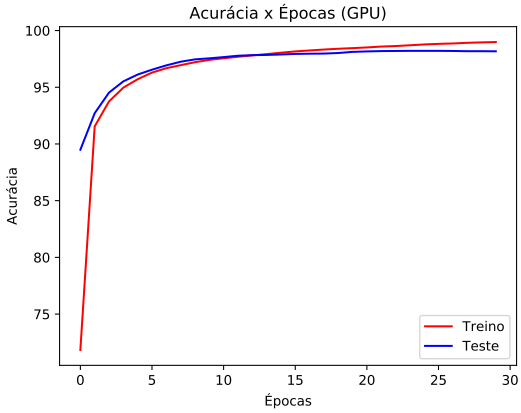
A coluna **Speedup com carga** apresenta os speedups alcançados pelas GPU's, considerando a inicialização feita pela Theano, que inclui a carga da base de dados, a inicialização de alguns objetos e a inicialização da placa gráfica. Vale ressaltar que essas inicializações são executadas em CPU sequencialmente. A coluna **Speedup sem carga** mostra o ganho alcançado sem considerar a etapa de inicialização.



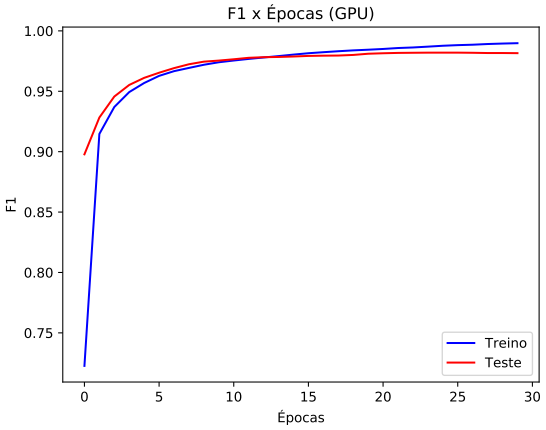
(a) Acurácia da rede neural CPU



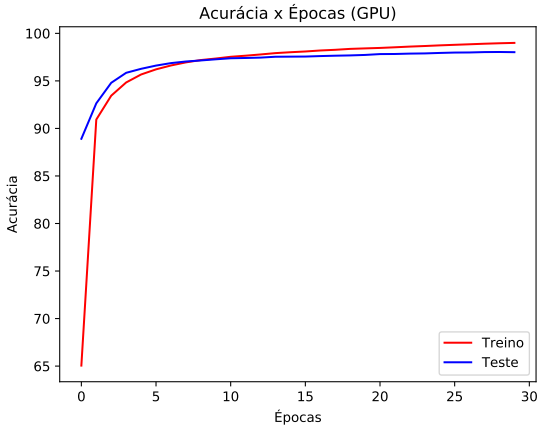
(b) F1 da rede neural CPU



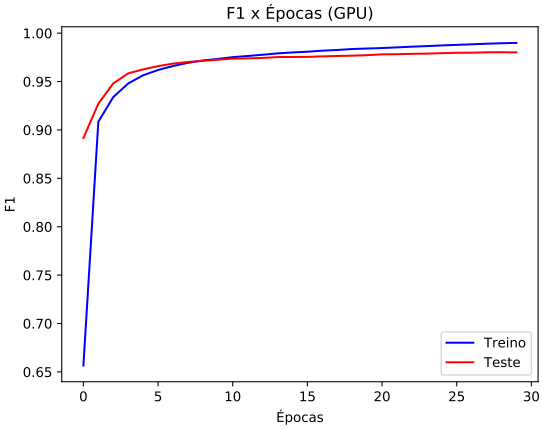
(c) Acurácia da rede neural GPU na placa GTX 980



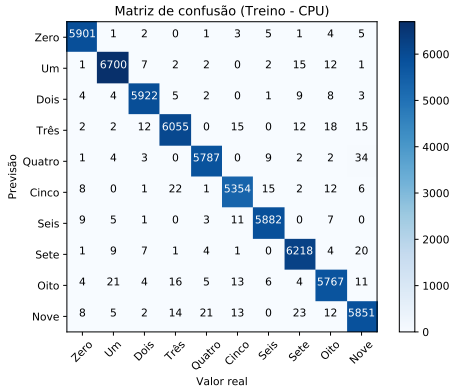
(d) F1 da rede neural GPU na placa GTX 980



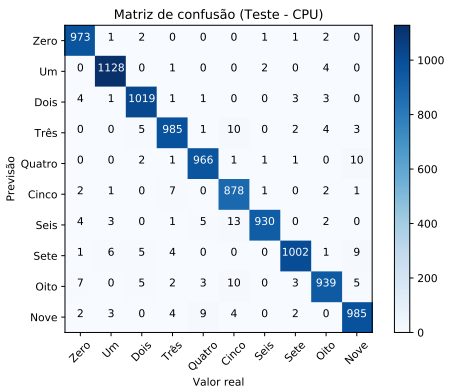
(e) Acurácia da rede neural GPU na placa Tesla K40



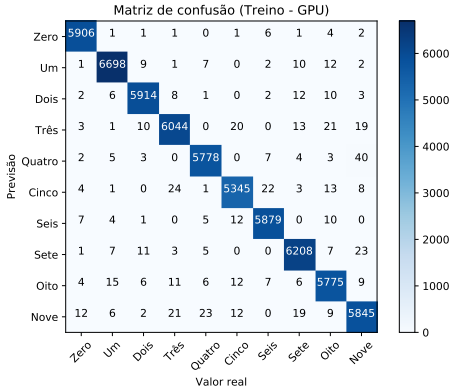
(f) F1 da rede neural GPU na placa Tesla K40



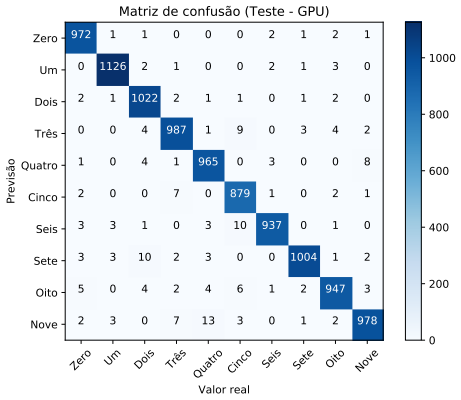
(a) Matriz de confusão na CPU - Treino



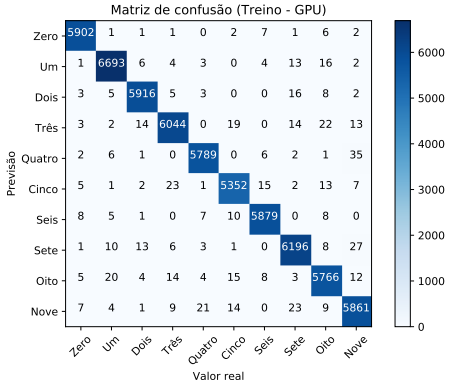
(b) Matriz de confusão na CPU - Teste



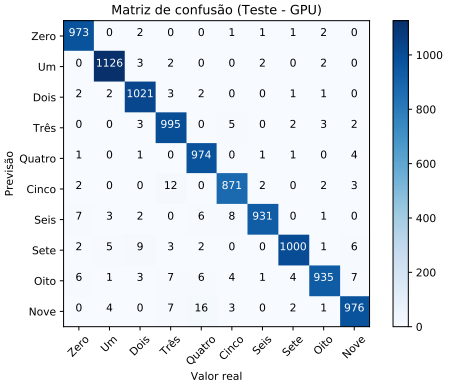
(c) Matriz de confusão na GTX 980 - Treino



(d) Matriz de confusão na GTX 980 - Teste



(e) Matriz de confusão na Tesla k40 - Treino



(f) Matriz de confusão na Tesla k40 - Teste

Máquinas	Tipo	Carregar dados(s)	Execução(s)	**Speedup c/ carga**	**Speedup s/ carga**
GTX 980	CPU	1.90(s)	180.43(s)	5.1×	15.84×
	GPU	24.38 (s)	11.39(s)		
K40c	CPU	6.47(s)	186.77(s)	3,69×	9,52×
	GPU	32.77(s)	19.62(s)		

Tabela 5.2: Tabela de tempos de execução, em segundos, e speedup.

Capítulo 6

Conclusão

Concluímos que os objetivos deste trabalho foram alcançados, visto que conseguimos atingir bons resultados no aprendizado da rede neural desenvolvida para a base de dados MNIST tanto nas execuções da CPU quanto da GPU e com uma aceleração de 9 a 15 vezes no treinamento com o uso das placas gráficas, levando-se em consideração a parte paralelizável da execução.

Sem dúvida o uso da biblioteca *Theano* facilita no desenvolvimento de soluções de programação científica, por auxiliar tanto na representação de expressões matemáticas quanto na otimização da paralelização em placas de vídeo programáveis. Porém, existe um limite até onde podemos controlar as ações de configuração dos parâmetros para tal, como nos parâmetros de estrutura de execução do CUDA, grades, blocos e *threads*.

Analisando os resultados de acurácia e F1 score no capítulo 5 pode ser percebido uma diferença de menos de 0.5% entre os valores obtidos pela CPU com os da GPU, que representa uma margem de erro bem pequena, portanto, podemos dizer que não houveram perdas relevantes nos resultados finais dessas métricas. Já em relação ao *speedup* mesmo levando-se em consideração a fase de carregamento de dados e inicialização das os objetos e expressões do *Theano*, ainda foi possível obter um aumento na performance no tempo de execução, o tempo de carregamento para a GPU acaba fazendo com que o *speedup* seja a terça parte do speedup onde o

carregamento é desconsiderado. Esse tempo de carregamento se deve ao fato do tipo de variável utilizada no *Theano*, *shared* (compartilhada), vista no capítulo 5, variáveis deste tipo são transferidas automaticamente para a GPU pela *Theano*, caso a execução esteja configurada para executar em uma placa gráfica. Portanto, uma vez que os dados estejam alocados na memória da GPU o ganho em termos de tempo é maior.

Levando em consideração os resultados mostrado, posso dizer que vale a pena fazer uso de placas gráficas para aplicações deste tipo, pois é possível obter ganhos em aprendizado equiparáveis aos ganhou em aplicações da CPU, com um tempo menor de execução.

6.1 Trabalhos Futuros

No trabalho que serviu como inspiração para esta dissertação LeCun et al. (1998) apresenta uma forma estendida da sua base de dados onde ele distorce as imagens do MNIST para aumentar a quantidade de amostras, e conclui que esse aumentou proporcionou a Lenet5 uma pequena melhora nos dados apresentados. Levando isso em consideração, outras base de dados foram pensadas para serem testadas.

Duas bases de dados chegaram a serem consideradas para este trabalho o fashion MNIST (SE) e o notMNIST (Dey). O primeiro, é uma base de dados proposta pela empresa Zalando Research como uma alternativa para o MNIST, que ao invés de conter imagens de números contém imagens de peças de vestuário que podem assumir 10 classes, assim como o MNIST. Já o notMNIST, é uma base de dados composta por letras de diversas fontes diferentes, também com 10 classes as quais as imagens podem assumir, a até j, porém um diferencial desta base é a quantidade de imagens que ela possui e o fato dela ser formada por caracteres com fontes muito diversas.

A realização de teste com outras bases de dados como as supra citadas seriam interessantes para ser o comportamento da rede bem como quais ajustes seriam necessários para a rede conseguisse um bom desempenho. Outra questão que foi levanta durante a fase final dos experimentos, foi a ideia de juntar essas base de

dados como uma só, para que assim pudéssemos avaliar o desempenho da rede em uma quantidade grande de dados e com 30 classes diferentes. (LeCun et al., 1998) aborda em seu trabalho sobre um problema que surge nessa situação, onde classes diferentes possuem características muito próximas como o número 1 e a letra I, a letra O e o número 0, dentro outras colisões.

Referências

Romain Bertrand, Petra Gomez-Kramer, Oriol Ramos Terrades, Patrick Franco, and Jean Marc Ogier. A system based on intrinsic features for fraudulent document detection. *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR*, pages 106–110, 2013. ISSN 15205363. doi: 10.1109/ICDAR.2013.29.

Francesco Camastra and Alessandro Vinciarelli. Combining Neural Gas and Learning Vector Quantization For Cursive Character Recopnition. *Idiar Research Report*, 2001.

John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*, volume 53. 2013. ISBN 978-1-118-73932-7. doi: 10.1017/CBO9781107415324.004.

Kévin WIN-LIME Clarisse MANDRIDAKE, Amine OUDDAN, Mathieu HOARAU. Towards Fully Automatic ID Document frauds detection. pages 1–6.

Shane Cook. *Cuda Programming*. 2013. ISBN 9780124159334.

CS231n. CS231n Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/convolutional-networks/`. [Online; accessed 20-Octuber-2017].

Sandipan Dey. Deep Learning with TensorFlow in Python. `https://www.datasciencecentral.com/profiles/blogs/deep-learning-with-tensorflow-in-python?utm_content=buffer8f865&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer`. [Online; accessed 20-Octuber-2017].

Douglas Eck and Jürgen Schmidhuber. A First Look at Music Composition using LSTM Recurrent Neural Networks. *Idsia*, pages 1–11, 2002. URL `http://www.idsia.ch/{~}juergen/blues/IDSIA-07-02.pdf`.

M. Harris. Mapping computational concepts to gpus. *M. Pharr(Ed.), GPU Gems (2), Addison-Wesley, Boston, USA*, pages 493–508, 2005.

John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

Justin Johnson. neural-style. `https://github.com/jcjohnson/neural-style`, 2015.

Manvjeet Kaur, Mukhwinder Singh, Akshay Girdhar, and Parvinder S Sandhu. Fingerprint Verification System using Minutiae Extraction Technique. *Engineering and Technology*, pages 497–502, 2008.

David Kriesel. Neural Networks. page 286, 2013. URL `http://www.dkriesel.com/en/science/neural{_}networks`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks.

LISA lab. Convolutional Neural Networks (LeNet). `http://deeplearning.net/tutorial/lenet.html`. [Online; accessed 2-December-2017].

Y. LeCun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Müller, E. Säckinger, P. Simard, and V. Vapnik. Comparison of Learning Algorithms for Handwritten Digit Recognition. *Proc. International Conference on Artificial Neural Networks*, pages 53–60, 1995. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.4023`.

Yann LeCun, Léon Buttou, Bengio Yoshua, and Haffner Patrick. Gradient-Based Leaning Applied to Document Recognition. 1998.

Felipe Melo, Raul S Ferreira, Fellipe Duarte, and Marcelo Zamith. Análise de desempenho da biblioteca Theano com GPU sob a ótica do CUDA Profiler. 2017.

AL MOHAMMAd, AL AbdUSAMAd, and EYA MA. Artificial Intelligence Technique for Speech Recognition based on Neural Networks. (May):66–72, 2014. URL http://www.computerscijournal.org/pdf/vol7no3/vol7no3{_}331-336.pdf.

Anuja P Nagare. License Plate Character Recognition System using Neural Network. *International Journal of Computer Application*, 25(10):36–39, 2011. ISSN 09758887. doi: 10.5120/3147-4345.

Michael Nielsen. 2016. URL http://neuralnetworksanddeeplearning.com/chap1.html.

M S Nixon and A S Aguado. Feature Extraction and Image Processing. *Academic Press*, page 88, 2008. doi: 10.1016/B978-0-12-396549-3.00001-X. URL http://www.amazon.com/dp/0750650788.

NVIDIA. KEPLER ARCHITECTURE. http://www.nvidia.com/object/nvidia-kepler.html, a. [Online; accessed 9-Octuber-2017].

NVIDIA. Maxwell Architecture. https://developer.nvidia.com/maxwell-compute-architecture, b. [Online; accessed 9-Octuber-2017].

Chirag Patel Smt Chandaben Mohanbhai, Dipti Shah, G H Patel PG, Atul Patel, and Smt Chandaben Mohanbhai. Automatic Number Plate Recognition System (ANPR): A Survey. *International Journal of Computer Applications*, 69(9):975–8887, 2013.

Signal Processing and Pattern Recognition. *Computer Vision and*, volume 2 Signal P. 1999. ISBN 0123797705. doi: 10.1007/s00138-006-0021-7. URL http://dl.acm.org/citation.cfm?id=553575.

Kandula Venkata Reddy, D Rajeswara Rao, and K Rajesh. Hand Written Character Detection by Using Fuzzy Logic Techniques. 3(3):256–260, 2013.

Henry A Rowley, Student Member, Shumeet Baluja, and Takeo Kanade. Neural Network-Based Face Detection. 20(1):23–38, 1998.

Zalando SE. fashion-mnist. `https://github.com/zalandoresearch/fashion-mnist`. [Online; accessed 20-Octuber-2017].

N Sebaa, N Sebaa, Z E a Fellah, Z E a Fellah, M Fellah, M Fellah, E Ogam, E Ogam, a Wirgin, a Wirgin, F G Mitri, F G Mitri, C Depollier, C Depollier, W Lauriks, and W Lauriks. *Ultrasonic characterization of human cancellous bone using the Biot theory: inverse problem.*, volume 120. 2006. ISBN 0898715725. doi: 10.1137/1.9780898717921. URL `http://www.ncbi.nlm.nih.gov/pubmed/17069280`.

Richard Szeliski. *Computer Vision : Algorithms and Applications*, volume 5. 2010. ISBN 1848829345. doi: 10.1007/978-1-84882-935-0. URL `http://research.microsoft.com/en-us/um/people/szeliski/book/drafts/szelski{_}20080330am{_}draft.pdf`.

Yong Haur Tay Yong Haur Tay, P.M. Lallican, M. Khalid, C. Viard-Gaudin, and S. Kneer. An offline cursive handwritten word recognition system. *Proceedings of IEEE Region 10 International Conference on Electrical and Electronic Technology. TENCON 2001 (Cat. No.01CH37239)*, 2, 2001. doi: 10.1109/TENCON.2001.949649.

Alex Teichman and Sebastian Thrun. Practical object recognition in autonomous driving and beyond. In *Proceedings of IEEE Workshop on Advanced Robotics and its Social Impacts, ARSO*, 2011. ISBN 9781467307963. doi: 10.1109/ARSO.2011.6301978.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

Georgios Vamvakas, Basilis Gatos, and Stavros J. Perantonis. Handwritten character recognition through two-stage foreground sub-sampling. *Pattern Recognition*, 43 (8):2807–2816, 2010. ISSN 00313203. doi: 10.1016/j.patcog.2010.02.018. URL `http://dx.doi.org/10.1016/j.patcog.2010.02.018`.

Marcelo Zamith and Esteban G. W. Clua. Programming in CUDA for Kepler and

Maxwell Architecture. *Revista de Informática Teórica e Aplicada 22*, 2:233–257, 2015. ISSN 21752745.

M.P.M. Zamith, E.W.G. Clua, A. Conci, and A. Montenegro. Parallel processing between gpu and cpu: Concepts in a game architecture. In *Computer Graphics, Imaging and Visualisation, 2007. CGIV '07*, pages 115–120, 2007. doi: 10.1109/CGIV.2007.64.