CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

CURSO DE BACHARELADO EM ENGENHARIA ELÉTRICA

TRABALHO DE CONCLUSÃO DE CURSO

SISTEMA AUTOMÁTICO DE TRADUÇÃO PARA GRAFIA BRAILLE:
Aplicação de redes neurais em tecnologia assistiva

Ítalo Eluezer Rodrigues Vicente

SISTEMA AUTOMÁTICO DE TRADUÇÃO PARA GRAFIA BRAILLE:

Aplicação de redes neurais em tecnologia assistiva

Trabalho apresentado como requisito parcial para a Conclusão do Curso de Bacharelado em Engenharia Elétrica do Centro Federal de Educação Tecnológica de Minas Gerais

COMISSÃO EXAMINADORA

_____

Professor Márcio Wladimir Santana

_____

Professora Rosana Áurea Tonetti Massahud

_____

Professora Zélia Maria Velloso Missagia

Nepomuceno, 30 de novembro de 2020

DEDICATÓRIA

Dedico este trabalho a minha família e amigos, que depositaram em mim toda confiança. Dedico também aos meus professores que durante a graduação me auxiliaram de maneira única

AGRADECIMENTOS

Ao meu professor-orientador e à professora coorientadora por toda dedicação e respeito.

A todos os meus amigos e colegas que de uma forma ou outra contribuíram na realização deste trabalho.

À professora Andréa Barra e ao professor Baltazar Jonas Ribeiro de Morais por todo apoio durante a graduação.

Ao CEFET-MG e em especial aos funcionários da biblioteca que sempre estavam dispostos a nos orientar em nossas pesquisas bibliográficas.

EPÍGRAFE

“*A imaginação é mais importante que o conhecimento.*”.

Albert Einstein

RESUMO

O presente trabalho tem como tema Sistema automático de tradução para grafia braille: Aplicação de redes neurais em tecnologia assistiva. Onde o objetivo é abordar a utilização da tecnologia, para desenvolver um algoritmo capaz de auxiliar deficientes visuais a realizarem a leitura da lousa em ambientes escolares sem necessidade de auxílio de um profissional, tornando possível a independência do usuário. O surgimento do tema ocorreu devido a falta de pesquisas para tecnologias assistivas, que auxiliem as minorias da sociedade a possuírem uma vida normal e com o mínimo de limitações possíveis. O trabalho é dividido em algumas partes, com o intuito de um detalhamento melhor, para que o mesmo seja reproduzido de uma maneira simples e rápido. Para realizar a automatização da tradução da grafia braille, será utilizada o sistema com redes neurais, que comumente é usado em diversas aplicações com inteligência artificial. Este tipo de processo permite que o sistema se adapte a novos padrões de escrita sem que o algoritmo necessite de alteração. O sistema basicamente utiliza uma biblioteca livre denominada mnist_loader, que possui um acervo de 60000 mil imagens de dígitos, que permite a construção e teste do algoritmo. Com todo o procedimento de elaboração do algoritmo seguido, o teste foi realizado e com isso encontrou-se uma eficiente de 95.28%. Todos os parâmetros foram usados de acordo com sua eficiência e determinando como padrão os que apresentaram melhores resultados. Avaliando todo o processo e resultados alcançados percebe-se que o sistema apresentou um bom desempenho e pode ser aplicado em outros bancos de dados. Além disso, o trabalho desenvolvido pode ser utilizado como base de outro algoritmo para realizar o reconhecimento de letras e palavras, tornando-o assim aplicado a um maior grupo de dados.

Palavras-chave: Tecnologia assistiva, redes neurais, escrita braille.

Sumário

1 INTRODUÇÃO....1
1.1 Objetivo principal....2
1.1.1 Objetivos específicos....2
1.2 Hipótese para solução....3
1.3 Resultados Esperados....3
1.4 Motivação....3
2 REFERENCIAL TEÓRICO....4
2.1 Breve História das redes neurais artificiais....4
2.2 Modelo de um neurônio biológico....5
2.2 Rede neural artificial....6
2.4 Tipos de arquiteturas de Redes Neurais Artificiais....7
2.4.1 Rede Feedforward de 1 camada....7
2.4.2 Rede Feedforward Multicamadas....8
2.4.3 Redes Neurais Recorrentes....9
2.4.4 Redes Neurais convolucionais....11
2.5 Algoritmo de propagação LeNet....12
2.5.1 Camada Convolucional....13
2.5.2 Camada de Pooling....16
2.5.3 Camada Totalmente Conectada....16
2.6 Tensorflow....17
2.7 O braile....18
2.7.2 Alfabeto em braille....18
2.7.3 Equipamentos de escrita braille....19
3 METODOLOGIA....21
3.1 Elaboração de um sistema de reconhecimento de dígitos....21
3.1.1 Descida em Gradiente....23
3.1.2 Construindo a Rede neural com Descida em Gradiente....23
3.1.3 Algoritmo de Backpropagation....26
3.1.4 Entropia Cruzada....28
3.1.5 Regulação (Overfitting)....28
3.1.6 Finalizando o algoritmo de reconhecimento de dígitos....29
4 Resultados e Discussão....32
4.1 Alterações de Épocas....32
4.2 Alterações de taxa de aprendizado....33
4.3 Alterações nos números de neurônios ocultos....35
4.4 Discussão dos resultados....36
5 Conclusão....38
6 Trabalhos Futuros....39
7 Referências Bibliográficas....40
Anexos....42
Anexo 1 - Algoritmo de reconhecimento de dígitos....42
Classe Network2....42
Classe Mnist_loader....45
Classe Overfitting....45
Classe Generate_gradient....51
Classe Test....54

Índice de figuras

Figura 1: Modelo de um neurônio biológico Fonte: Deep Learning Book.....5
Figura 2: Modelo Neurônio de Pitts e McCulloch Fonte: Furtado. 2019.....6
Figu*ra 3 - Rede FeedForward de uma camada*.....8
Figura 4 - Rede Feedforward Multicamadas Fonte: Furtado. 2019.....9
Figura 5 - Redes Neurais Recorrentes.....10
Figura 6 - Redes Neurais Recorrentes com neurônio oculto.....11
Figura 7: - Estrutura de uma Rede Neural Convolucional Fonte: Deep Learning Book.....13
Figura 8: - Arranjo 3D da camada de convolução.....14
Figura 9 - Arranjo de passos no sistema convolucional *Fonte: ARAÚJO et. al., 2017*.....15
Figura 10 - Camada Polling.....16
Figura 11: Unidade de processamento de um sistema totalmente conectado *Fonte: (ARAÚJO et. al., 2017).*.....17
Figura 12: Alfabeto em braile Fonte: FRAGA, 2017.....19
Figura 13 - Algoritmo Descida em Gradiente.....24
Figura 14 - Agorítmo do método de Feedforward Fonte: Elaborada pelo autor.....24
Figura 15 - Algoritmo Descida em Gradiente Fonte: Elaborada pelo autor.....25
Figura 16 - Algoritimo de MiniBatch Fonte: Elaborada pelo autor.....26
Figura 17 - Algoritmo de Backpropagation Fonte: Elaborada pelo autor.....27
Figura 18 - Atenuação de sinal de overfitting Fonte: Elaborada pelo autor.....29
Figura 19 - Itens que devem estar na mesma pasta Fonte: Elaborada pelo autor.....30
Figura 20: Imagem usada para teste Fonte: Biblioteca Mnist_loader.....30
Figura 21: - Representação de algarismo 2 em linguagem braille Fonte: Elaborada pelo autor.....31
Figura 22: Gráfico de alteração de Épocas em relação a precisão.....33
Figura 23: Gráfico de alteração da taxa de aprendizado em relação a precisão Fonte: Elaborada pelo autor.....34
Figura 24: Gráfico de alteração de neurônio ocultos em relação a precisão Fonte: Elaborada pelo autor.....36

Índice de Tabelas

Tabela 1: Alteração de número de Épocas.....32
Tabela 2: Alteração da taxa de aprendizado Fonte: Elaborada pelo autor.....34
Tabela 3: Alteração no número de neurônios ocultos Fonte: Elaborada pelo autor.....35

1 INTRODUÇÃO

O estudo de redes neurais, nos últimos anos vem ganhando destaque dentro das universidades e centro tecnológicos, devido a vasta aplicação em equipamentos e processos industriais. Este estudo proporciona o desenvolvimento de tecnologias que facilitam processos, possibilitando maior independência de equipamentos. Assim, equipamentos poderão realizar tarefas sem que a interferência humana seja necessária. Com isso é possível resolver problemas industriais bastante característicos, tais como: necessidade de grande quantidade de mão de obra, trabalho em ambiente de alta periculosidade ou insalubre e prejuízos financeiros provocados por erros humanos.

Em relação a redes neurais, Segundo D. Strilg 2010, uma das tarefas primordiais no desenvolvimento de aplicações com redes neurais é a etapa de treinamento na qual a rede é adaptada para um problema específico através de dados já disponíveis. Devido ao elevado custo computacional requerido nessa etapa, é frequente o uso de GPUs (Graphics Processing Unit) em virtude do massivo processamento paralelo possibilitado por tais unidades, o qual leva a uma aceleração significativa no processo de treinamento.

Um outro setor da produção tecnológica que tem ganhado destaque na última década é a produção de tecnologia assistiva. Tecnologia assistiva é:

> "(...) produtos, equipamentos, dispositivos, recursos, metodologias, estratégias, práticas e serviços que objetivem promover a funcionalidade, relacionada à atividade e à participação da pessoa com deficiência ou com mobilidade reduzida, visando à sua autonomia, independência, qualidade de vida e inclusão social. (BRASIL, 2015)"

Essa tecnologia tem o intuito de ajudar Pessoas com Deficiência (PcD), melhorando assim a inclusão dessas pessoas a sociedade. Atualmente PcD possuem leis que as permitem acesso a direitos considerados básicos, como acesso a empregos com concorrência igualitária, acesso a rede pública e privada de educação sem que haja discriminação ou a imposição de dificuldade ao acesso ou

ao aprendizado, além de leis de assistencialismo em caso de impossibilidade de trabalho.

1.1 Objetivo principal

Com isso, este trabalho tem o intuito de aumentar a inclusão na sociedade, facilitando ainda mais a adaptação de PcD a ambientes escolares, especificamente, os deficientes visuais. O objetivo da tecnologia assistiva é aumentar a independência de seu usuário e possibilitar a ele uma maior adequação aos locais sem perder qualquer tipo de informação.

Ao levar em consideração a pesquisa da Professora Andrea de Aguiar Kasper da Universidade Federal de Santa Catarina, publicada em 2008 na revista Educar, aproximadamente 15% da população brasileira possui algum tipo de deficiência. Além disso cerca de 45% deste percentual são pessoas que possuem algum tipo de deficiência visual. Com a finalidade de auxiliar os deficientes visuais no sistema educacional brasileiro, este trabalho de conclusão de curso abordará uma solução tecnológica para permitir uma maior independência dos mesmos.

Este trabalho propõe a criação de uma simulação computacional que permita fazer o reconhecimento de texto manuscrito e convertê-lo para linguagem braile, utilizando redes neurais.

1.1.1 Objetivos específicos

- Realizar o estudo de redes neurais convolucionais (CNN), redes neurais recorrentes (RNN) e redes neurais de classificação temporal conexional (CTC).

- Obter banco de dados de imagens para realização das análises computacionais.

- Realizar o estudo da implementação do sistema em um microcontrolador.

- Elaborar uma simulação computacional que represente o sistema de conversão de linguagem escrita para linguagem braile.

- Realizar o estudo da viabilidade de construção do protótipo.

1.2 Hipótese para solução

Para isso este trabalho propõe o estudo e a elaboração de uma simulação digital de um sistema que realize a conversão de texto manuscrito para linguagem em braile utilizando redes neurais, além disso também propõe o estudo e o levantamento dos materiais necessários para a elaboração de um protótipo do equipamento.

1.3 Resultados Esperados

Espera-se ao fim da pesquisa a elaboração de um algoritmo que permita a conversão de uma grafia manuscrita obtida através de uma lousa digital, limitando-se a números e letras do alfabeto. Espera-se que o tempo de resposta deste algoritmo seja pequeno, não permitindo perdas de letras.

1.4 Motivação

Ao analisar o cenário atual, é dever dos futuros engenheiros eletricistas atentarem a necessidade de todos os seres humanos. Com isso este trabalho teve como motivação tentar criar uma tecnologia que poderia beneficiar uma das camadas menos favorecidas da sociedade, os portadores de necessidades especiais com perda total ou perda parcial de visão.

2 REFERENCIAL TEÓRICO

2.1 Breve História das redes neurais artificiais

As primeiras informações sobre o uso de neuro computação ou redes neurais em computação datam de 1943, publicados em artigos do psiquiatra e neuroanatomista McCulloch e do matemático Pitts, que sugeriam a construção de uma máquina baseada no cérebro humano.

Em 1951 foi construído o primeiro neuro computador, criado por Mavin Minsky. Este equipamento não obteve êxito na sua função inicial, mas serviu de inspiração para as ideias de estruturas de redes neurais que é usado na atualidade. 5 anos depois, em 1956 surgiram dois grandes paradigmas da Inteligência Artificial: A Simbólica e Conexionista (FURTADO, 2019).

A Inteligência Artificial Simbólica tenta simular o comportamento inteligente humano desconsiderando os mecanismos responsáveis por tal, ou seja, não possui uma inspiração biológica. A Inteligência Artificial Conexionista acredita que construindo um sistema que simule a estrutura do cérebro, este sistema apresentará inteligência, será capaz de aprender, assimilar, errar e aprender com seus erros(FURTADO, 2019).

O primeiro neuro computador que obteve sucesso na sua função inicial, surgiu apenas no final de 1957 e início de 1958 por Frank Rosenblatt, Charles Wightman. Neste primeiro projeto de sucesso o interesse inicial era a criação de uma arquitetura de rede, que recebeu o nome de perceptron e era responsável por reconhecimento de padrões. Após a criação deste tipo de arquitetura por Frank Rosenblatt, Bernard Widrow, com a ajuda de alguns estudantes, desenvolveu um novo tipo de elemento de processamento de redes neurais chamado de Adaline, segundo Furtado 2019, Adaline é equipado com uma poderosa lei de aprendizado, que diferente do Perceptron permanece em uso.

2.2 Modelo de um neurônio biológico

Um neurônio é uma unidade de processamento de informação que é fundamental para a operação de uma rede neural. A figura 1 apresenta um modelo de um neurônio biológico.

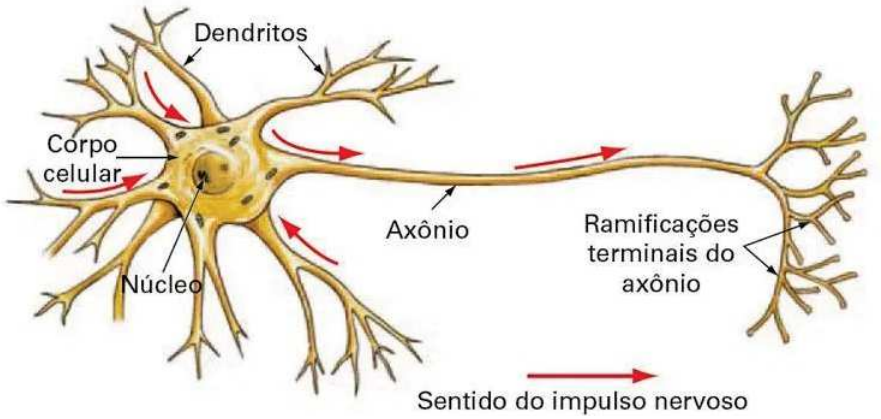


Figura 1: Modelo de um neurônio biológico
Fonte: Deep Learning Book

O neurônio é constituído por 3 partes principais: a soma ou corpo celular, do qual emanam algumas ramificações denominadas de dendritos, e por uma outra ramificação descendente da soma, porém mais extensa, chamada de axônio. Nas extremidades dos axônios estão os nervos terminais, pelos quais é realizada a transmissão das informações para outros neurônios. Esta transmissão é conhecida como sinapse (Haykin, 2001).

Cada neurônio possui um corpo central, diversos dendritos e um axônio. Os dendritos recebem sinais elétricos de outros neurônios através das sinapses, que constitui o processo de comunicação entre neurônios. O corpo celular processa a informação e envia para outro neurônio.

2.3 Modelo de um neurônio artificial

Um neurônio é uma unidade de processamento de informação que é fundamental para a operação de uma rede neural. A figura 2 apresenta um modelo de um

neurônio. Podemos identificar três elementos básicos do modelo:

1. Um conjunto de sinapses ou elos de conexão, onde cada sinapse possui um fator característico denominado peso. Com isso um sinal arbitrário na entrada de sinapse conectada a um neurônio é multiplicado pelo peso sináptico (FERNEDA,2006).

2. Um somador, que é responsável por somar os sinais de entrada, ponderados pelos pesos sinápticos.

3. Uma função de ativação, que restringe a amplitude de saída de um neurônio. Esta função de ativação pode ser referida como função restritiva. Tipicamente, o intervalo normalizado da amplitude da saída de um neurônio é escrito como o intervalo unitário fechado [0,1] ou alternativamente [-1, 1].

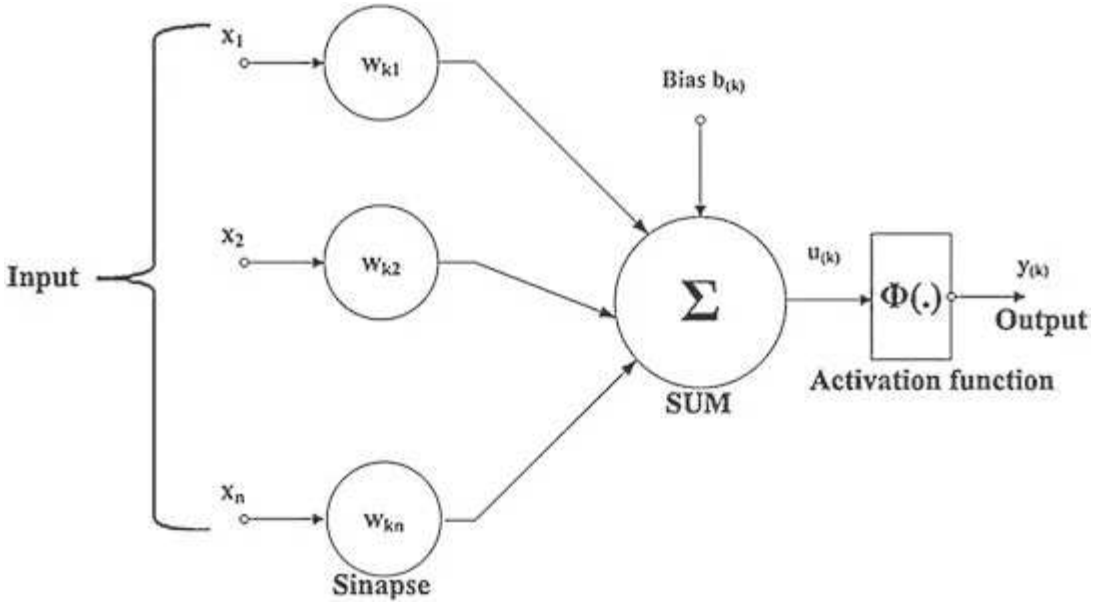


Figura 2: Modelo Neurônio de Pitts e McCulloch
Fonte: Furtado. 2019

2.2 Rede neural artificial

O neurônio artificial é uma estrutura lógico-matemática que procura simular a forma, o comportamento e as funções de um neurônio biológico. Pode-se associar o dendrito à entrada, o soma ao processamento e o axônio à saída; portanto, o

neurônio é considerado uma unidade fundamental processadora de informação.

Os dendritos são as entradas, cujas ligações com o corpo celular artificial são realizadas através de canais de comunicação que estão associados a um determinado peso (simulando as sinapses). Os estímulos captados pelas entradas são processados pela função do soma, e o limiar de disparo do neurônio biológico é substituído pela função de transferência. A figura 2 apresenta esquematicamente um neurônio de McCullock e Pitts (Furtado, 2019)

Uma rede neural artificial, na forma mais geral é um sistema que é projetado para modelar a maneira como o cérebro realiza uma tarefa particular. Para obter um bom desempenho, as redes neurais empregam uma interligação de células computacionais simples denominadas neurônios ou unidades de processamento.

> (...).Uma rede neural é um processador maciçamente paralelamente distribuído constituído de unidades de processamentos simples, que têm a propensão natural para armazenar conhecimento experimental e torna-lo disponível para o uso. Ela se assemelha ao cérebro em dois aspectos: O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem. Forças de conexão entre neurônios, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido. (Haykin 2001)

2.4 Tipos de arquiteturas de Redes Neurais Artificiais

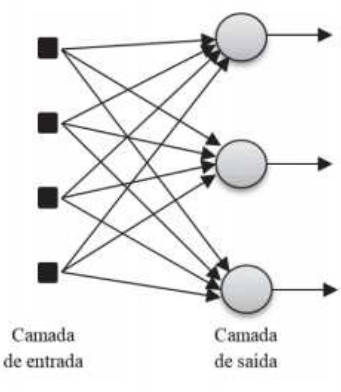
A maneira pela qual os neurônios de uma rede neural estão estruturados, está ligada com o algoritmo de aprendizagem usado para treinar a rede. Neste tópico, será explicado as redes neurais utilizadas para o desenvolvimento deste trabalho.

2.4.1 Rede Feedforward de 1 camada

Neste tipo de rede os neurônios são organizados em forma de camadas, como pode ser visto na figura 3. Os nós da camada de entrada se comunicam diretamente com a camada de saída (nós computacionais).

A arquitetura do tipo feedforward em camadas apresenta uma organização similar à do córtex humano, onde os neurônios se dispõem em camadas paralelas e consecutivas, e os axônios se estendem sempre no mesmo sentido, isto é, a informação propaga-se da entrada para a saída, não existindo portanto ligações entre os neurônios de uma mesma camada ou com camadas anteriores (*Furtado, 2019)*.

É chamada de rede de 1 camada em referência à camada de saída, pois os nós da entrada não processam nada, só apresentam os padrões à rede



Figur*a 3 - Rede FeedForward de uma camada*
Fonte: Furtado. 2019

2.4.2 Rede Feedforward Multicamadas

A segunda classe de redes feedforward distingue-se da anterior por apresentar uma ou mais camadas escondidas. A função dos neurônios escondidos é intervir entre a camada de entrada e a de saída da rede de alguma maneira útil. Pela adição de uma ou mais camadas, a rede passa a melhor mapear problemas mais complexos.

Por ser uma rede do tipo feedforward, as conexões se dão sempre no sentido

da camada de entrada para a de saída. Quando a rede possuir todos os nós de uma camada comunicando-se com todos os nós da camada posterior, ela é dita totalmente conectada. Caso alguma das conexões sinápticas não esteja ligada com a camada subsequente, a rede é dita parcialmente conectada (*Furtado, 2019)*. A figura 4 representa uma rede feedforward multicamada totalmente conectada.

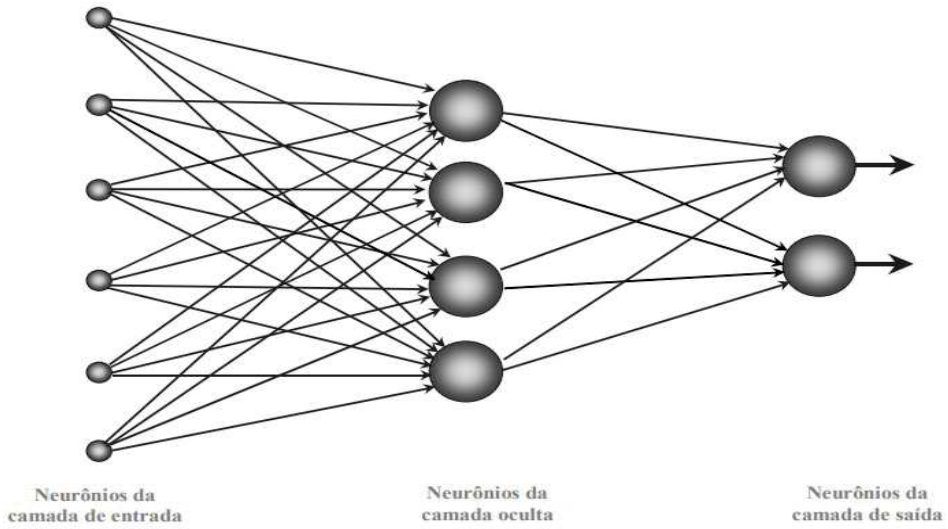


Figura 4 - Rede Feedforward Multicamadas
Fonte: Furtado. 2019

2.4.3 Redes Neurais Recorrentes

Uma rede neural recorrente se distingue de uma rede neural alimentada adiante por ter pelos menos um laço de realimentação. Uma rede recorrente pode consistir, de uma única camada de neurônios, com cada neurônio alimentando seu sinal de saída de volta para as entradas de todos os outros neurônios, como mostrado na figura 5. Podemos observar que na estrutura representada, não há laços de auto realimentação na rede; auto realimentação se refere a uma situação onde a saída de um neurônio é realimentada para a sua própria entrada (VIEIRA, 2003). A rede recorrente da figura 4 também não possui neurônio na cada oculta.

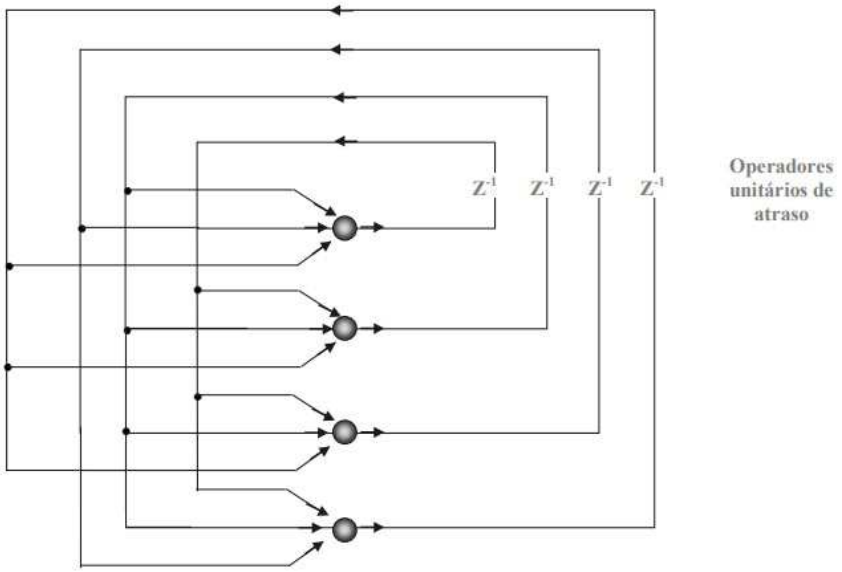


Figura 5 - Redes Neurais Recorrentes

Fonte: VIEIRA. 2003

Na figura 6 é apresentado uma outra classe de redes recorrentes com neurônios ocultos. As conexões de realimentação mostradas se originam dos neurônios ocultos bem como dos neurônios de saída.

A presença de laços de realimentação, quer seja na estrutura recorrente da figura 5 ou da estrutura da figura 6, tem um impacto profundo na capacidade de aprendizagem da rede e no seu desempenho. Além disso, os laços de realimentação envolvem o uso de ramos particulares compostos de elementos de atraso unitário, o que resulta em um comportamento dinâmico não linear, admitindo-se que a rede neural contenha unidades não lineares(Viera, *2013)* .

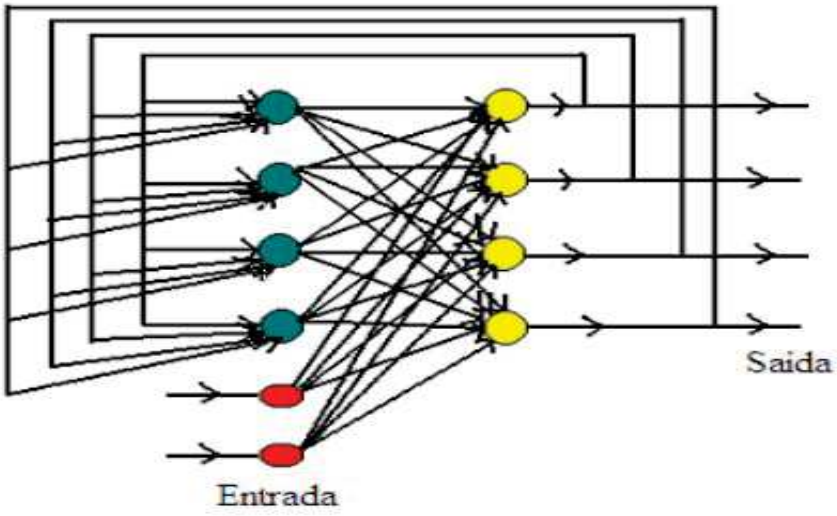


Figura 6 - Redes Neurais Recorrentes com neurônio oculto

Fonte: https://imasters.com.br/data/um-mergulho-profundo-nas-redes-neurais-recorrentes

2.4.4 Redes Neurais convolucionais

> (...). As redes neurais convolucionais (ConvNets ou CNNs) se tornaram o novo padrão em visão computacional e são fáceis de treinar quando existe grande quantidade de amostras rotuladas que representam as diferentes classes-alvo.
>
> Algumas das vantagens das redes convolucionais são: (a) capacidade de extrair características relevantes através de aprendizado de transformações (kernels) e (b) depender de menor número de parâmetros de ajustes do que redes totalmente conectadas com o mesmo número de camadas ocultas. Como cada unidade de uma camada não é conectada com todas as unidades da camada seguinte, há menos pesos para serem atualizados, facilitando assim o treinamento. (...). (Flavio H, 2017)

Para a utilização de redes convolucionais é necessário o uso de diversas bibliotecas. Estas bibliotecas são responsáveis por definir que tipo específico de

estrutura será usado.

> Atualmente, existem diversas bibliotecas que providenciam implementações das principais operações utilizadas pelas CNNs, tais como: Caffe [Jia et al. 2014], MatConvNet [Vedaldi e Lenc 2015], Theano [Al-Rfou et al. 2016], Torch [Collobert et al. 2011] e TensorFlow [Abadi et al. 2015]. No entanto, a familiarização com essas bibliotecas e o entendimento de como as CNNs trabalham com imagens não são tarefas triviais. (...) (Flavio H, 2017)

2.5 Algoritmo de propagação LeNet

Um dos primeiros projetos de redes neurais convolucionais, também chamados de CNNs foi proposto por Yann LeCun em 1988, onde foi usada para impulsionar o campo de Deep Learning. Este projeto foi utilizado para reconhecimento de caracteres. Novos tipos de arquiteturas foram propostos desde 1988, porém todas são baseadas no modelo original. A figura 6 descreve os conceitos fundamentais de uma CNN.

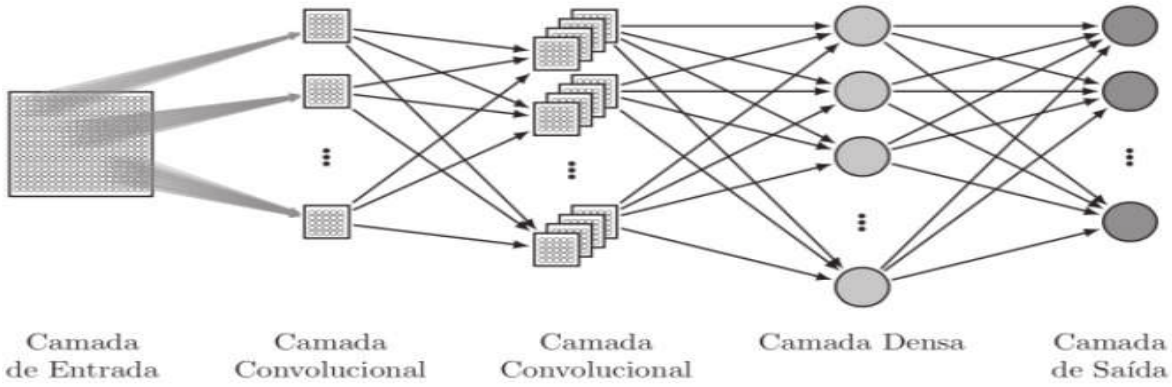


Figura 7: - Estrutura de uma Rede Neural Convolucional
Fonte: Deep Learning Book

As CNNs são formadas por sequências de camadas e cada uma destas possui uma função específica na propagação do sinal de entrada. A figura 7 ilustra a arquitetura de uma LeNet e suas três principais camadas: Convolucionais, de poooling (camada densa) e totalmente conectadas(camada de saída) (Deep Learning Book).

2.5.1 Camada Convolucional

As camadas convolucionais consistem de um conjunto de filtros que recebem como entrada um arranjo 3D, também chamado de volume. Cada filtro possui dimensões reduzidas, porém ele se estende por toda a profundidade do volume de entrada. Automaticamente, durante o processo de treinamento da rede, esses filtros são ajustados para que sejam ativados em presença de características relevantes identificadas no volume de entrada, como orientação de bordas ou manchas de cores.

Existem três parâmetros que controlam o tamanho do volume resultante da camada Convolucional: profundidade (depth), passo (stride) e zero-passing. (ARAÚJO et. al., 2017):

- Profundidade: depende do número de filtros utilizados.
- Passo: tamanho do salto na operação de convolução.
- Zero-Padding: preenche a borda do volume de entrada.

Cada um desses filtros será responsável por extrair as características diferentes dos dados de entrada. Quanto o maior número de filtros, maior será o número de características extraídas dos dados de entrada, porém a complexidade computacional, relativa ao tempo e ao uso de memória será maior.

A largura e a altura do volume resultante dependem do passo e do zero-

podding. O parâmetro passo especifica o tamanho do salto na operação de convolução, como é ilustrado na figura 8 (ARAÚJO et. al., 2017).

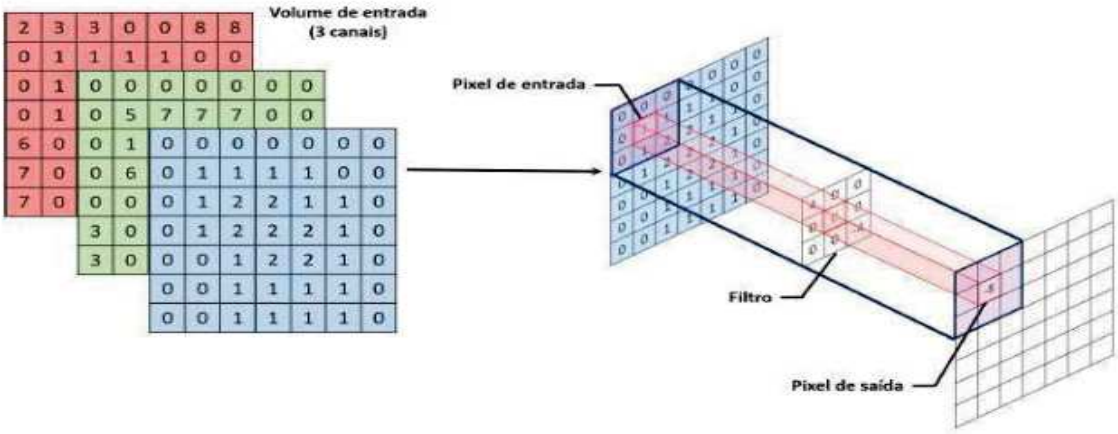


Figura 8: - Arranjo 3D da camada de convolução
Fonte: (ARAÚJO et. al., 2017)

Observando a figura 9, quando o passo é igual a 1 o filtro salta somente uma posição por vez, e quando o passo é igual a dois o filtro salta duas posições por vez. Quando se salta várias posições de uma vez, é possível que algumas informações sejam perdidas, este é um dos motivos que o filtro de dois passos não é utilizado com frequência.

A operação de zero-padding consiste em preencher com zeros a borda do volume de entrada. A grande vantagem dessa operação é poder controlar a altura e largura do volume resultante.

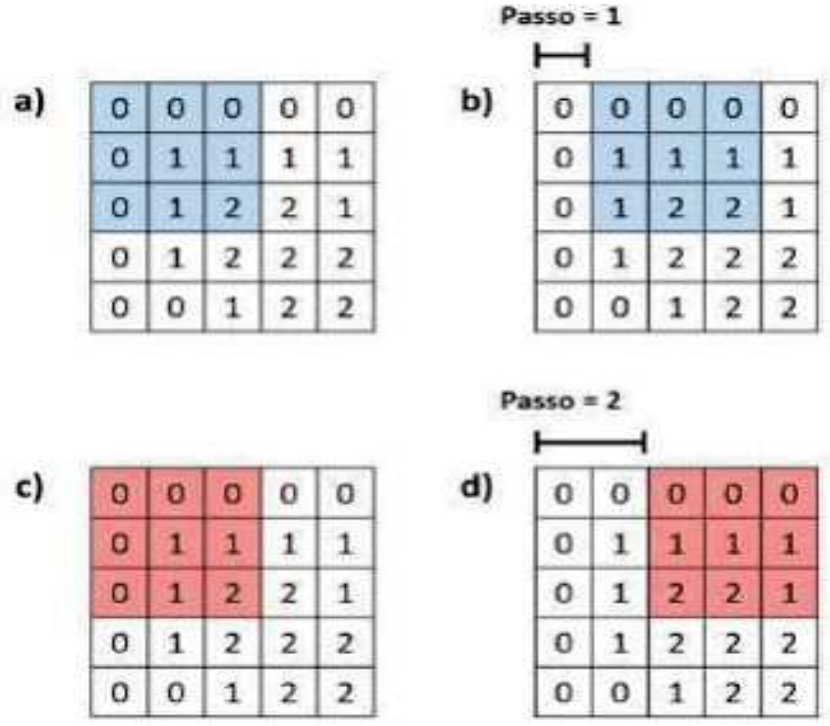


Figura 9 - Arranjo de passos no sistema convolucional
Fonte: ARAÚJO et. al., 2017

2.5.2 Camada de Pooling

Após uma camada Convolucional, geralmente existe uma camada de pooling. A função desta camada é reduzir progressivamente a dimensão espacial do volume de entrada e consequentemente a redução do custo computacional da rede e evitando overfitting.

> Na operação de pooling, os valores pertencentes a uma determinada região do mapa de atributos, gerados pelas camadas convolucionais, são substituídos por alguma métrica dessa região. A forma mais simples de pooling consiste em substituir os valores de uma região pelo valor máximo. (ARAÚJO et. al., 2017).

A forma mais utilizada de pooling consiste em substituir valores de uma região pelo seu valor máximo, esta operação é conhecida como max pooling e é útil para eliminar valores desprezíveis reduzindo a dimensão da representação dos dados e acelerando a computação necessária para as próximas camadas. Este tipo de ação faz com que o sistema crie uma invariância a pequenas mudanças e distorções locais. A figura 10 apresenta uma operação de max pooling (ARAÚJO et. al., 2017).

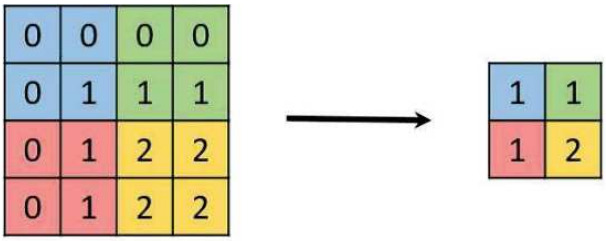


Figura 10 - Camada Polling
Fonte: (ARAÚJO et. al., 2017).

2.5.3 Camada Totalmente Conectada

Com os resultados das camadas convolucionais e de Pooling que trarão as características da imagem de entrada, a camada totalmente conectada realiza a classificação da imagem utilizando uma classe pré-determinada para avaliar essas características. (ARAÚJO et. al., 2017).

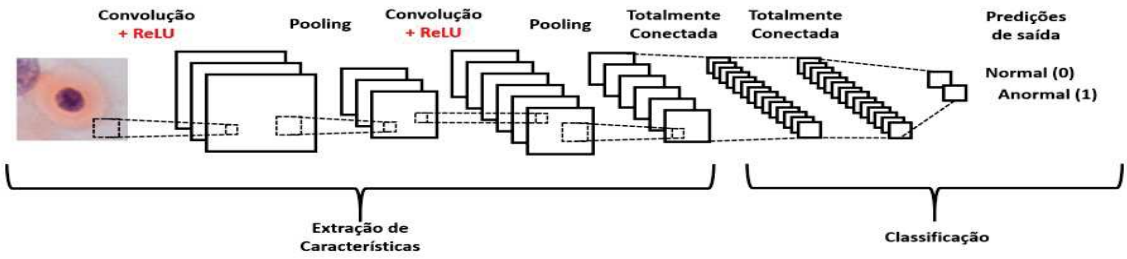


Figura 11: Unidade de processamento de um sistema totalmente conectado
Fonte: (ARAÚJO et. al., 2017).

Essas camadas são formadas por unidades de processamento conhecidas como neurônio, e o termo totalmente conectado significa que todos os neurônios da camada anteriores estão conectados a todos os neurônios da camada seguinte.

> (...). Uma técnica conhecida como dropout também é bastante utilizada entre as camadas totalmente conectadas para reduzir o tempo de treinamento e evitar overfitting. Esta técnica consiste em remover, aleatoriamente a cada iteração de treinamento, uma determinada porcentagem dos neurônios de uma camada, readicionando-os na iteração seguinte. Essa técnica também confere à rede a habilidade de aprender atributos mais robustos, uma vez que um neurônio não pode depender da presença especifica de outros neurônios. (Flavio H, 2017)

2.6 Tensorflow

O TensorFlow é uma biblioteca de código aberto para computação numérica rápida. Foi criada pelo Google e é mantida por ela até hoje. Nominalmente foi lançada para linguagem Python, embora exista acesso a C++ subjacente. Ao contrário de outras bibliotecas numéricas destinadas ao uso em Deep Learning como o Theano, o TensorFlow foi projetado para uso em pesquisa e desenvolvimento e em sistemas de produção, além do RankBrainna pesquisa do Google e no divertido projeto DeepDream. Ele pode ser executado em sistemas de CPU única, GPUs, bem como dispositivos móveis e sistemas distribuídos em larga escala de centenas de máquinas. (*ARAÚJO et. al., 2017*)

2.7 O braile

2.7.1 Uma breve história

O braille foi inventado no século XIX por um homem chamado Louis Braille, que era completamente cego.

> Em 1825, Braille mal tinha dezesseis anos, mas achava ter encontrado algo ao mesmo tempo funcional e superior ao sistema existente (de letras em relevo). Seu código original consistia em seis pontos dispostos

> em duas fileiras paralelas; cada conjunto de linhas representava uma letra. Isso era mais simples do que o sistema de Barbier, e ainda versátil o suficiente para permitir até 64 variações, o suficiente para todas as letras do alfabeto e pontuação.(https://gizmodo.uol.com.br/invencao-braille/)

2.7.2 Alfabeto em braille

A figura 12 exibe o alfabeto em braille. O conjunto numérico é representado por dois conjuntos de pontos em relevo, já as letras são dispostas em apenas um conjunto de pontos em relevo. Além da representação da figura 12, existem escritas específicas para pontuações, acentos e caracteres especiais.

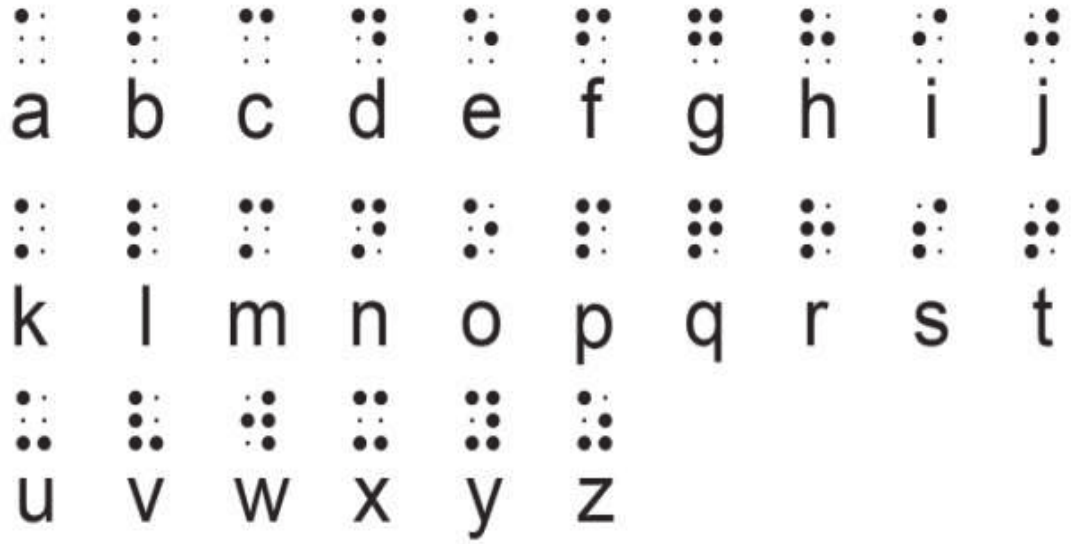


Figura 12: Alfabeto em braile
Fonte: FRAGA, 2017

2.7.3 Equipamentos de escrita braille

Uns dos primeiros equipamentos desenvolvidos para as pessoas com deficiência visual é o reglete. “A escrita foi adaptada do próprio criador deste alfabeto usado para que pessoas cegas possam ler e escrever, Louis Braille. Ele

usava uma prancha com uma régua que continha as celas do alfabeto para que qualquer letra pudesse ser escrita” (FRAGA, 2017).

Mesmo sendo o antigo, é um dos instrumentos mais utilizados atualmente, mesmo depois da invenção de equipamentos mais modernos. Esta utilização em grande parte é por causa da sua praticidade no transporte.

Um outro equipamento utilizado é a impressora Braille, que faz a impressão em um papel específico que permite que seu usuário escreva e leia todos seus textos digitados. Mesmo com tantos equipamentos no cenário atual, existem problemas significativos no fornecimento e divulgação de materiais textuais em Braille, devido a vários fatores. Dentre os quais, o mais significativo trata do elevado custo das impressoras Braille e do volume físico ocupado pelo material impresso. Outro elemento a ser considerado é a deterioração dos caracteres depois de três leituras táteis, devido ao desgaste físico provocado pelo contato da pele humana com o material impresso(FRAGA, 2017).

Vários sistemas têm sido desenvolvidos para armazenar eletronicamente dados representativos de caracteres Braille e os reproduzir para um leitor com deficiência visual, transformando arquivos de texto em arquivos de áudio. Esta prática viabiliza o uso de computadores por deficientes visuais, permitindo um alto nível de independência no estudo e no trabalho. Este sistema é o mais moderno que existe, porém, ainda está passando por processo de modelagem para que atinja todas as classes sociais.

3 METODOLOGIA

A caracterização do trabalho proposto é dividida nas seguintes etapas: elaboração de um sistema de reconhecimento de dígitos; testes do sistema de redes neurais desenvolvidos, buscando diminuir o erro do sistema; elaboração do sistema responsável por realizar a conversão para linguagem braile.

3.1 Elaboração de um sistema de reconhecimento de dígitos

Após o levantamento bibliográfico, com a finalidade de obter uma aproximação entre o aluno e as redes neurais, foi realizado o estudo e simulação de uma rede neural capaz de reconhecimento de dígitos manuscrito. Para isso foi utilizado uma rede neural de três camadas. A camada de entrada da rede contém neurônios que codificam os valores dos pixels de entrada. Os dados de treinamento para a rede consistirão em muitas imagens de 28 por 28 pixels de dígitos manuscritos digitalizados e, portanto, a camada de entrada contém 28 × 28 = 784 neurônios. Os pixels de entrada são de escala de cinza, com um valor de 0.0 representando branco e um valor de 1.0 representando preto. Valores intermediários representam tonalidades gradualmente escurecidas de cinza.

A segunda camada da rede é uma camada oculta. Será representado o número de neurônios nesta camada oculta por *n*, e vamos experimentar diferentes valores para n.

A camada de saída da rede contém 10 neurônios. Se o primeiro neurônio “disparar” (for ativado), ou seja, tiver uma saída ≈ 1, então isso indicará que a rede acha que o dígito é 0. Se o segundo neurônio “disparar”, isso indicará que a rede pensa que o dígito é um 1. E assim por diante. Em resumo, vamos numerar os neurônios de saída de 0 a 9 e descobrimos qual neurônio possui o maior valor de ativação. Se esse neurônio é, digamos, neurônio número 6, então nossa rede adivinhará que o dígito de entrada era um 6. E assim por diante para os outros neurônios de saída.

Com isso é definido arquitetura de rede neural. Na etapa seguinte é necessário definir como será o processo de aprendizagem do algoritmo, antes de começar a codificar a rede em linguagem Python. Para isso é definido que será utilizado o treinamento com Gradiente Descendente. Com a definição do processo de aprendizado, é necessário obter um conjunto de dados para o treinamento da rede. Para isso será utilizado um conjunto de dados coletados pelo Instituto Nacional de Padrões e Tecnologia dos Estados Unidos, denominado conjunto de dados MNIST.

O MNIST tem duas partes. A primeira parte contém 60.000 imagens para serem usadas como dados de treinamento. Essas imagens são amostras de manuscritos escaneados de 250 pessoas. As imagens estão em escala de cinza e 28 por 28 pixels de tamanho. A segunda parte do conjunto de dados MNIST tem 10.000 imagens a serem usadas como dados de teste, também 28 por 28 pixels em escala de cinza.

A segunda parte do conjunto de dados será utilizado para validação da rede neural criada, permitindo assim definir a taxa de erro que a mesma possui. Basicamente o que é preterido é um algoritmo que nos permita encontrar pesos e bias para que a saída da rede se aproxime de y(x) para todas as entradas de treinamento x. Para quantificar o quão bem a arquitetura está de alcançar o objetivo, definindo assim uma função de custo:

$$C(w,b)=\frac{1}{2n}\sum_{x}\|y(x)-a\|^2 \qquad (1)$$

Na fórmula apresentada, w indica a coleta de todos os pesos na rede, b os bias (viés), n é o número total de entradas de treinamento, a é o vetor de saídas da rede (quando x é entrada) e a soma é sobre todas as entradas de treinamento x. C é a função de custo quadrático, que também é conhecido como o erro quadrático médio ou apenas o MSE (Mean Squared Error).

Analisando a função de custo definida, pode-se concluir que C não será negativo para nenhuma hipótese. Portanto, o algoritmo de treinamento definido pode encontrar pesos e bias para que C (w, b) ≈ 0. Isso significa basicamente que o modelo proposto deverá fazer as previsões corretas, quando C tender a zero, porém apresentará uma quantidade de erros significativos quando C não tender a zero, por isso a justificativa do método de treinando conhecido como Descida de Gradiente, pois o mesmo diminuirá os erros apresentados quando C não tender a zero.

3.1.1 Descida em Gradiente

A Descida do Gradiente é uma ferramenta padrão para otimizar funções complexas iterativamente dentro de um programa de computador. Seu objetivo é: dada alguma função arbitrária, encontrar um mínimo. Para alguns pequenos subconjuntos de funções há apenas um único *minumum* que também acontece de ser global. Para as funções mais realistas, pode haver muitos mínimos, então a maioria dos mínimos são locais.

Descida do Gradiente é um algoritmo de otimização usado para encontrar os valores de parâmetros (coeficientes ou se preferir w e b – weight e bias) de uma função que minimizam uma função de custo. A Descida do Gradiente é melhor usada quando os parâmetros não podem ser calculados analiticamente (por exemplo, usando álgebra linear) e devem ser pesquisados por um algoritmo de otimização.

O procedimento começa com valores iniciais para o coeficiente ou coeficientes da função. Estes poderiam ser 0.0 ou um pequeno valor aleatório (a inicialização dos coeficientes é parte crítica do processo e diversas técnicas podem ser usadas, ficando a escolha a cargo do problema a ser resolvido com o modelo).

3.1.2 Construindo a Rede neural com Descida em Gradiente

O programa desenvolvido neste item estará disponível no anexo 1. O tópico será utilizado para explicação de como foi o desenvolvimento e os resultados obtidos.

```
class Network(object):

    def __init__(self, sizes):


        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
```

Figura 13 - Algoritmo Descida em Gradiente
Fonte: Elaborada pelo autor

Em relação a figura 13, a classe principal do programa será chamada por Network, nela possui todo o programa responsável pelo reconhecimento dos dígitos. O responsável pelo número de neurônios será o comando sizes.

Os bias e pesos no objeto rede1 são todos inicializados aleatoriamente, usando a função Numpy np.random.randon para gerar distribuições gaussianas com 0 de média e desvio padrão 1. Esta inicialização aleatória dá ao nosso algoritmo de descida do gradiente estocástico um local para começar. No algoritmo é possível notar que os bias e pesos são armazenados como lista de matrizes Numpy. Assim, por exemplo, rede1.weights é uma matriz Numpy armazenando os pesos conectando a segunda e terceira camadas de neurônios.

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

Figura 14 - Agorítmo do método de Feedforward
Fonte: Elaborada pelo autor

Após a inicialização do sistema, um método feedforward é adicionado na classe principal, aplicando a equação 2 ao sistema:

$$a' = \sigma(wa + b) \qquad (2)$$

Ao aplicar a equação 2, é dado início o processo de aprendizado do sistema, para isso é aplicado o método de Descida em Gradiente, denominado no programa como SGD, apresentada pela figura 15.

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):

    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)]

        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)

        if test_data:
            print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test));
        else:
            print("Epoch {} finalizada".format(j))
```

Figura 15 - Algoritmo Descida em Gradiente
Fonte: Elaborada pelo autor

No segmento apresentado na figura 15 vale ressaltar que é a função de aprendizado, porém a parte mais importante de treinando é realizado em uma outra função, denominada update_mini_batch.

```
def update_mini_batch(self, mini_batch, eta):


    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

    self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]
```

Figura 16 - Algoritimo de MiniBatch
Fonte: Elaborada pelo autor

No segmento apresentado na figura 16 a maior parte do trabalho é feita pela linha delta_nabla_b, delta_nabla_w = self.backprop (x, y). Isso invoca algoritmo chamado de backpropagation, que é uma maneira rápida de calcular o gradiente da função de custo.

3.1.3 Algoritmo de Backpropagation

Como já apresentado anteriormente, a maior parte do programa se passa pela atuação do algoritmo backpropagation, que é utilizado para calcular as derivadas parciais ∂Cx / ∂blj e ∂Cx / ∂wljk. Portanto, update_mini_batch funciona simplesmente calculando esses gradientes para cada exemplo de treinamento no mini_batch e, em seguida, atualizando self.weights e self.biases adequadamente. O código para backpropagation apresentado na figura 17, acompanhado de algumas funções auxiliares, que são usadas para calcular a função σ, a derivada σ′ e a derivada da função de custo.

```
def backprop(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    activation = x
    activations = [x]
    zs = []
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

Figura 17 - Algoritmo de Backpropagation
Fonte: Elaborada pelo autor

Nesse segmento do algoritmo apresentado na figura 17, são inicializadas as matrizes de pesos (nabla_w) e bias (nabla_b) com zeros. Essas matrizes serão alimentadas com valores durante o processo de treinamento. Depois de inicializar alguns objetos, existe um loop for para cada valor de b e w. Neste loop, é utilizado a função np.dot do Numpy para a multiplicação entre matrizes e adição do bias, adicionando o resultado na lista z e fazendo assim uma chamada à função de ativação Sigmoide. Ao final deste loop, é obtido a lista com todas as ativações.

Na Backward Pass (passada para trás) é calculado as derivadas e realizadas as multiplicações de matrizes mais uma vez. Nota-se que é chamado o método Transpose() para gerar a transposta da matriz e assim ajustar as dimensões antes de efetuar os cálculos. Por fim, é retornado bias e pesos.

Com isso finaliza-se o programa principal. Porém para que tenhamos um sistema que seja capaz de reconhecer padrões numéricos, é necessário realizar alguns ajustes e acrescentar programas secundários capazes de auxiliarem no processo.

3.1.4 Entropia Cruzada

A Cross-Entropy (entropia cruzada) é fácil de implementar como parte de um programa que aprende usando gradiente descendente e backpropagation. Para isso será utilizado uma rede com 30 neurônios ocultos, e um tamanho de mini-lote de 10. Definimos a taxa de aprendizado para $\eta = 0,5$ e nós treinamos por 30 épocas. O comando net.large_weight_initializer() é usado para inicializar os pesos e vieses.

Este comando é executado, pois posteriormente o peso padrão para inicialização das redes serão alterados. O resultado da execução da sequência de comandos acima é uma rede com 93,53% de precisão.

A entropia cruzada é comumente usada para quantificar a diferença entre duas distribuições de probabilidade. Geralmente, a distribuição “verdadeira” (dos dados usados para treinamento) é expressa em termos de uma distribuição One-Hot.

3.1.5 Regulação (Overfitting)

Overfitting é um grande problema em redes neurais. Isso é especialmente verdadeiro em redes modernas, que geralmente têm um grande número de pesos e vieses. Para treinar de forma eficaz, é necessário desenvolver uma maneira de detectar quando o overfitting está acontecendo.

A maneira de detectar overfitting é usar a abordagem continua, mantendo o controle da precisão nos dados de teste conforme os treinos da rede. Este algoritmo depois de uma quantidade de iterações já não apresenta melhoria em sua precisão, neste momento é que se deve parar o treinamento. É claro que, estritamente falando, isso não é necessariamente um sinal de overfitting. Pode ser que a precisão

nos dados de teste e os dados de treinamento parem de melhorar ao mesmo tempo. Ainda assim, a adoção dessa estratégia impedirá o overfitting.

Portanto, será utilizado uma variação dessa estratégia. Ressalta-se que, quando os dados MNIST foram carregados, os mesmos possuíam três conjuntos de dados:

```
import mnist_loader
import network2
import pickle
import gzip
import numpy as np
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
```

Figura 18 - Atenuação de sinal de overfitting
Fonte: Elaborada pelo autor

Com isso, além de usar o training_data e test_data iremos também utilizar o validation_data. O validation_data contém 10.000 imagens de dígitos, imagens que são diferentes das 50.000 imagens no conjunto de treinamento MNIST e das 10.000 imagens no conjunto de teste MNIST. Em vez de usar o test_data para evitar overfitting, será usado o validation_data. Para fazer isso, praticamente a mesma estratégia descrita anteriormente será utilizada para o test_data. Ou seja, a precisão da classificação nos dados de validação no final de cada época será calculada. Quando a precisão da classificação nos dados de validação estiver saturada, o treinamento será interrompido. Essa estratégia é chamada de parada antecipada (Early-Stopping). Para o sistema apresentado, foi utilizado um programa de Overfitting padrão extraído do livro Neural Networks and Deep Learning de Michael Nielsen

3.1.6 Finalizando o algoritmo de reconhecimento de dígitos

Com todos os programas auxiliares descritos e o programa principal finalizado, é realizado a comunicação entre todos, assim fazendo um programa completo, onde todos os resultados de interações e alterações de itens serão discutidos nos resultados desta monografia.

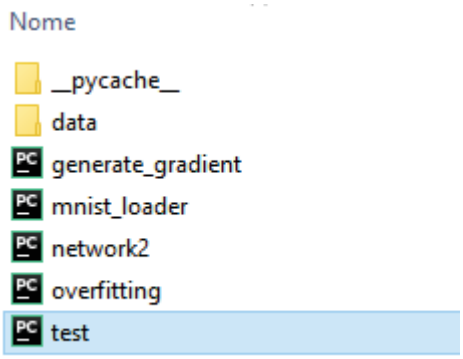


Figura 19 - Itens que devem estar na mesma pasta
Fonte: Elaborada pelo autor

Como apresentado na figura 19, é de extrema importância que todos os arquivos estejam na mesma pasta.

3.2 Conversão para a gráfia braille.

Como já relatado na construção do algoritmo, a camada de saída possui 10 neurônios, onde os mesmos são responsáveis pela identificação dos dígitos. Como exemplo usemos a figura 20 a seguir. Esta sequência de dígitos foi aplicada no algoritmo, para que o mesmo pudesse vir a atuar e encontrar seus correspondentes em grafia Braille.

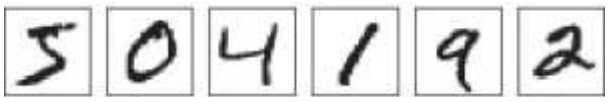


Figura 20: Imagem usada para teste
Fonte: Biblioteca Mnist_loader

Para realizar tal aplicação, todos os dígitos foram separados e identificados individualmente. Quando o último número desta sequência foi identificado, o neurônio 2 da camada de saída foi ativado, fazendo com que o mesmo agora pudesse ser convertido em linguagem braille. O corresponde deste número 2 em braille está representado na figura 21

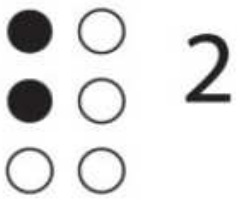


Figura 21: - Representação de algarismo 2 em linguagem braille
Fonte: Elaborada pelo autor

Todos os outros números corresponderão a neurônios de ativação, sendo o número 5 ao neurônio 5, o número 0 ao neurônio 10, o número 4 ao neurônio 4, o número 1 ao neurônio 1 e o número 9 ao neurônio 9.

.

4 Resultados e Discussão

Neste tópico será discutido todos os resultados extraídos do algoritmo construído, além de realização de mudança em números de épocas, taxa de aprendizado e número de neurônios ocultos. Estas alterações têm a finalidade de verificar o comportamento do sistema para situações novas. Para conclusão da precisão apresentada pelo algoritmo, o programa será executado por um total de 10 vezes, e assim será realizado o cálculo de média simples. O algoritmo utilizado neste tópico será disponibilizado em formado de Anexo I.

4.1 Alterações de Épocas

Para realização dos primeiros testes, foram fixados os valores de taxa de aprendizado, neurônios ocultos e tamanho do lote. Os valores definidos foram 3.0, 30 e 10 respectivamente. Com isso o processo teve início com alterações dos valores de épocas, como descritos na tabela 1 e gráfico apresentado na figura 22. Fonte:

Tabela 1: Alteração de número de Épocas

Número de Épocas	Tamanho do Lote	Taxa de Aprendizado	Neurônios Ocultos	Precisão Alcançada
30	10	3.0	30	95.01%
50	10	3.0	30	94.51%
100	10	3.0	30	95.28%
200	10	3.0	30	95.24%

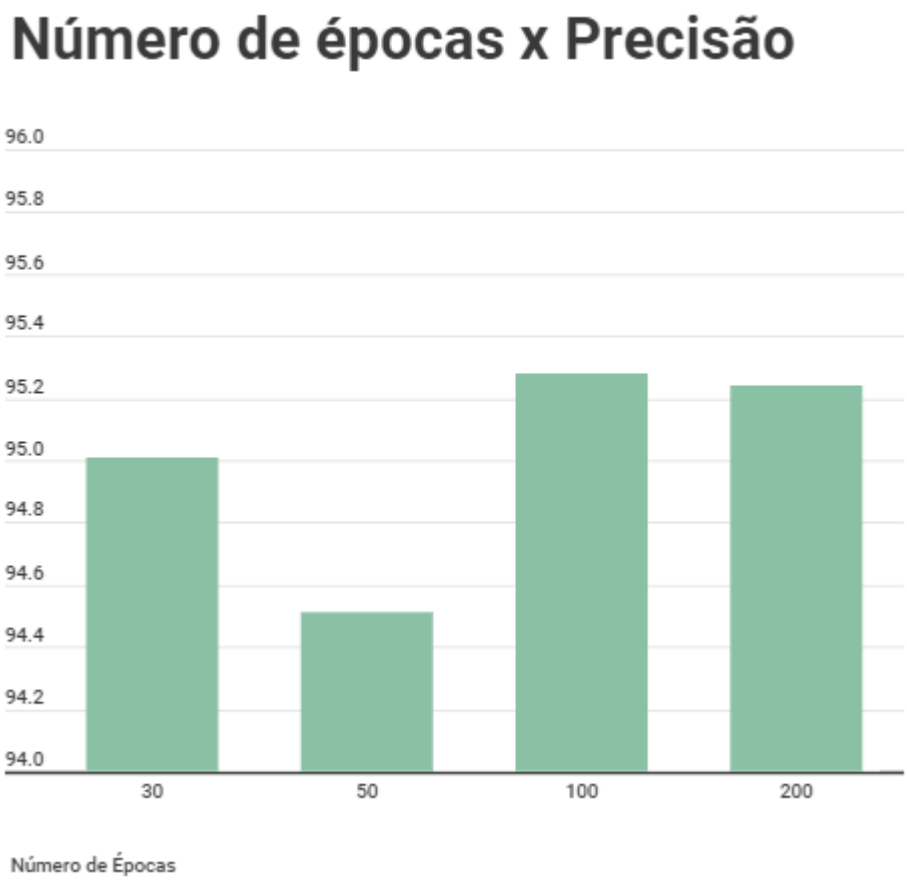


Figura 22: Gráfico de alteração de Épocas em relação a precisão

Fonte: Elaborada pelo autor

4.2 Alterações de taxa de aprendizado.

Após alteração do número de épocas, será alterada a taxa de aprendizado do sistema, para verificar se ocorrerá o aumento da taxa de precisão do algoritmo. Para isso será utilizado o número de épocas que obteve a melhor performasse no item 4.1. Em relação ao número de neurônios e tamanho lote os valores serão 30 e 10 respectivamente. A tabela 2 e o gráfico na figura 23 representam a variação da precisão de acordo com os valores incrementados na taxa de aprendizado.

Tabela 2: Alteração da taxa de aprendizado
Fonte: Elaborada pelo autor

Número de Épocas	Tamanho do Lote	Taxa de Aprendizado	Neurônios Ocultos	Precisão Alcançada
100	10	0.5	30	94.05%
100	10	1.0	30	94.32%
100	10	1.5	30	94.67%
100	10	2.0	30	94.87%
100	10	2.5	30	95.01%
100	10	3.0	30	95.28%
100	10	3.5	30	94.68%

Figura 23: Gráfico de alteração da taxa de aprendizado em relação a precisão
Fonte: Elaborada pelo autor

4.3 Alterações nos números de neurônios ocultos

Após alteração do número de épocas e da taxa de aprendizado, será alterado o número de neurônios na camada oculta, para verificar se ocorrerá o aumento da taxa de precisão do algoritmo. Para isso foi utilizado o número de épocas que obteve a melhor performasse no item 4.1 e a taxa de aprendizado que obteve melhor performasse no item 4.2. Em relação ao tamanho lote o valor utilizado será 10. A tabela 3 e o gráfico da figura 24 representam a variação da precisão de acordo com os valores incrementados no número de neurônios ocultos.

Tabela 3: Alteração no número de neurônios ocultos
Fonte: Elaborada pelo autor

Número de Épocas	Tamanho do Lote	Taxa de Aprendizado	Neurônios Ocultos	Precisão Alcançada
100	10	3.0	10	90.05%
100	10	3.0	15	92.83%
100	10	3.0	20	94.12%
100	10	3.0	25	95.01%
100	10	3.0	30	95.28%
100	10	3.0	35	94.78%
100	10	3.0	40	94.68%

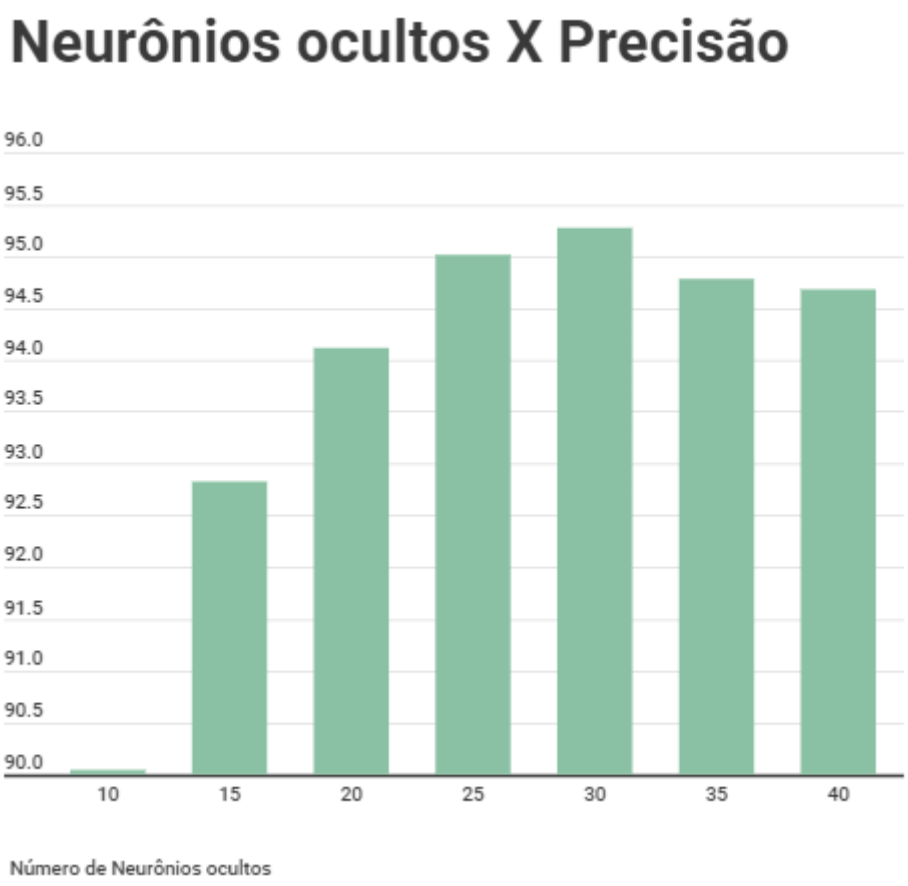


Figura 24: Gráfico de alteração de neurônio ocultos em relação a precisão
Fonte: Elaborada pelo autor

4.4 Discussão dos resultados

Observa-se que o algoritmo foi criado com a finalidade de conseguir uma performasse satisfatória para captação de dígitos. Para isso foi realizada a análise dos dados obtidos nos testes.

Primeiramente, a definição da taxa e de número de neurônios foi realizado de forma empírica, pois este foi um valor inicial utilizado na definição do problema e que apresentou uma taxa de erro menor que 5%. Com isso ao analisar a variação do número de épocas, observa-se que chegando-se a 100 épocas é o suficiente para encontrar a estabilidade. Na figura 22 podemos observar que com 200 épocas praticamente a taxa de sucesso não há mudanças significativas. Para fins de comparação foi realizada apenas uma única vez a simulação com 500 épocas, onde

constatou-se uma taxa de sucesso menor que a apresentada para 100 e 200 épocas. Ao definirmos a época padrão do algoritmo, podemos agora sim verifica qual a taxa de aprendizado será mais adequada para algoritmo. Nota-se na figura 23 e tabela 2, quanto maior o valor de taxa, maior a taxa de sucesso. O valor que essa taxa de sucesso se satura é o valor de 3.0, sendo assim, a partir deste valor a taxa de sucesso volta a cair. Com isso a taxa de aprendizado padrão do nosso algoritmo será 3.0.

Por fim, é verificado se o número de neurônios na camada oculta influenciará na taxa de sucesso do algoritmo. Percebe-se um comportamento semelhante ao que ocorre com a taxa de aprendizado. Com valores mais baixos a taxa de sucesso é menor, porém ao aumentar o número de neurônios, maior será a taxa de sucesso. Vale ressaltar que há um valor de saturação, que neste caso seria 30 neurônios ocultos, pois a partir deste valor, a taxa de sucesso volta a ter redução em seu valor.

Com todos estes testes, é possível encontrar valores padrões que farão com que o algoritmo tenha uma taxa de sucesso satisfatória. A taxa de sucesso para um total de 10 testes é de 95.28%, sendo possível encontrar os dígitos quase sempre corretos.

Com a taxa de erro menor que 5%, o algoritmo torna-se viável para a aplicação aqui pesquisada, fazendo com que o mesmo seja base de um algoritmo maior que será proposto como trabalho futuro.

5 Conclusão

O desenvolvimento do presente estudo possibilitou uma análise de como uma tecnologia assistiva pode ser criada, levando em consideração a parte responsável por fazer todo o reconhecimento de padrões e assim tornar o sistema independente. Além disso, também permitiu um maior aprofundamento em sistemas de inteligência artificial, que em si é a parte que mais demanda tempo do processo.

Ao fazer o teste do algoritmo, verificou-se que as partes mais complexas e desgastantes do processo são o registro de dados. Contudo é a parte mais importante, pois é neste momento que são feitos os ajustes e permitindo assim, que os objetivos propostos fossem realmente alcançados.

Nesse sentido, a utilização de recursos digitais permite aos deficientes visuais possam realizar seu estudo de forma mais rápida e eficiente.

6 Trabalhos Futuros

Para trabalho futuro, coloca-se a extensão deste algoritmo, pois o que foi proposto por esta pesquisa, busca realizar a identificação dos dígitos usando redes neurais e os convertendo para grafia braille, para isso a sequência dele seria dividido em outros dois tópicos. O primeiro seria a adequação para que em um mesmo algoritmo possa ser feito a identificação, classificação de letras do alfabeto convencional. E o segundo seria a adequação para que o algoritmo também pudesse realizar a identificação de pontuações na língua portuguesa.

Vale ressaltar que, o sistema de identificação de letras, foi iniciado, porém apresentou uma taxa de sucesso inferior a 30%, tornando-o incapaz de obter sucesso na sua aplicação final.

7 Referências Bibliográficas

ARAÚJO, Flávio HD et al. Redes neurais convolucionais com tensorflow: Teoria e prática. SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. III Escola Regional de Informática do Piauí. Livro Anais-Artigos e Minicursos, v. 1, p. 382-406, 2017.

BRASIL. Lei nº 13.146, de 6 de julho de 2015.

COSTA, Marly Guimarães Fernandes. Redes Neurais Convolucionais na Saúde. Journal of Health Informatics, v. 9, n. 4, 2017.

Deep Learning , livro de Ian Goodfellow, Yoshua Bengio e Aaron Courville 2019< disponível em :http://neuralnetworksanddeeplearning.com/>

FLÁVIO H.. D. Araújo, Allan C. Carneiro, Romuere R. V. Silva, Fátima N. S. Medeiros e Daniela M.

FRAGA, Vladimir Figueiredo. Tecnologias de interface cérebro-computador para tradução de braille e libras: possibilidades e alternativas. ScientiaTec, v. 4, n. 1, p. 3-19, 2017.

FURTADO, Maria Inês Vasconcellos. Redes Neurais Artificiais: Uma Abordagem Para Sala de Aula. 2019. Editora Atena.

Lei nº 13.146, de 6 de julho de 2015.

PRESSMAN, Roger; MAXIM, Bruce. Engenharia de Software-8ª Edição. McGraw Hill Brasil, 2016.

SANTOS, Alan et al. Uma Abordagem de Classificação de Imagens Dermatoscópicas Utilizando Aprendizado Profundo com Redes Neurais Convolucionais. In: Anais do XVII Workshop de Informática Médica. SBC, 2017.

STRIGL, Daniel; KOFLER, Klaus; PODLIPNIG, Stefan. Performance and scalability of GPU-based convolutional neural networks. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. IEEE, 2010. p. 317-324.BRASIL.

VARGAS, Ana Caroline Gomes; PAES, Aline; VASCONCELOS, Cristina Nader. Um estudo sobre redes neurais convolucionais e sua aplicação em detecção de pedestres. In: Proceedings of the XXIX Conference on Graphics, Patterns and Images. 2016. p. 1-4

VIEIRA, Flávio Henrique Teles; LEMOS, Rodrigo Pinto; LEE, Luan Ling. Alocação Dinâmica de Taxa de Transmissão em Redes de Pacotes Utilizando Redes Neurais Recorrentes Treinadas com Algoritmos em Tempo Real. IEEE Latin America, v. 1, n. 1, 2003.

Ushizima. Redes Neurais Convolucionais com Tensorflow: Teoria e Prática.

Anexos

Anexo 1 - Algoritmo de reconhecimento de dígitos

Classe Network2

```
import random
import numpy as np
# Classe Network
class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in
zip(sizes[:-1], sizes[1:])]

   def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

   def SGD(self, training_data, epochs, mini_batch_size, eta,
test_data=None):
        training_data = list(training_data)
        n = len(training_data)
        if test_data:
            test_data = list(test_data)
            n_test = len(test_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k:k+mini_batch_size] for k
```

```
in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print("Epoch {} : {} /
{}".format(j,self.evaluate(test_data),n_test));
            else:
                print("Epoch {} finalizada".format(j))

    def update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
delta_nabla_w)]
       self.weights = [w-(eta/len(mini_batch))*nw for w, nw in
zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb for b, nb in
zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        activation = x
        activations = [x]
       zs = []
       for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
```

```
            activation = sigmoid(z)
            activations.append(activation)
        delta = self.cost_derivative(activations[-1], y) *
sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) *
sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-
1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y) for (x,
y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        return (output_activations-y)

def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

Classe Mnist_loader

```
     import pickle
import gzip
import numpy as np

def load_data():
    f = gzip.open('data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f,
encoding="latin1")
    f.close()
    return (training_data, validation_data, test_data)
def load_data_wrapper():
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)

def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

Classe Overfitting

```
sys.path.append('../src/')
```

```
import mnist_loader
import network2


def main(filename, num_epochs,
         training_cost_xmin=200,
         test_accuracy_xmin=200,
         test_cost_xmin=0,
         training_accuracy_xmin=0,
         training_set_size=1000,
         lmbda=0.0):

    run_network(filename, num_epochs, training_set_size, lmbda)
    make_plots(filename, num_epochs,
               training_cost_xmin,
               test_accuracy_xmin,
               test_cost_xmin,
               training_accuracy_xmin,
               training_set_size)

def run_network(filename, num_epochs, training_set_size=1000,
lmbda=0.0):
    random.seed(12345678)
    np.random.seed(12345678)
    training_data, validation_data, test_data =
mnist_loader.load_data_wrapper()
    net = network2.Network([784, 30, 10],
cost=network2.CrossEntropyCost())
    net.large_weight_initializer()
    test_cost, test_accuracy, training_cost, training_accuracy \
        = net.SGD(training_data[:training_set_size], num_epochs, 10,
0.5,
```

```
                    evaluation_data=test_data, lmbda = lmbda,
                    monitor_evaluation_cost=True,
                   monitor_evaluation_accuracy=True,
                   monitor_training_cost=True,
                   monitor_training_accuracy=True)
    f = open(filename, "w")
    json.dump([test_cost, test_accuracy, training_cost,
training_accuracy], f)
    f.close()

def make_plots(filename, num_epochs,
               training_cost_xmin=200,
               test_accuracy_xmin=200,
               test_cost_xmin=0,
               training_accuracy_xmin=0,
                training_set_size=1000):
    f = open(filename, "r")
    test_cost, test_accuracy, training_cost, training_accuracy =
json.load(f)
    f.close()
    plot_training_cost(training_cost, num_epochs,
training_cost_xmin)
    plot_test_accuracy(test_accuracy, num_epochs,
test_accuracy_xmin)
    plot_test_cost(test_cost, num_epochs, test_cost_xmin)
    plot_training_accuracy(training_accuracy, num_epochs,
                           training_accuracy_xmin, training_set_size)
    plot_overlay(test_accuracy, training_accuracy, num_epochs,
                 min(test_accuracy_xmin, training_accuracy_xmin),
                 training_set_size)

def plot_training_cost(training_cost, num_epochs,
```

```
training_cost_xmin):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(training_cost_xmin, num_epochs),
           training_cost[training_cost_xmin:num_epochs],
            color='#2A6EA6')
    ax.set_xlim([training_cost_xmin, num_epochs])
    ax.grid(True)
    ax.set_xlabel('Epoch')
    ax.set_title('Cost on the training data')
    plt.show()

def plot_test_accuracy(test_accuracy, num_epochs,
test_accuracy_xmin):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(test_accuracy_xmin, num_epochs),
           [accuracy/100.0
            for accuracy in
test_accuracy[test_accuracy_xmin:num_epochs]],
            color='#2A6EA6')
    ax.set_xlim([test_accuracy_xmin, num_epochs])
    ax.grid(True)
    ax.set_xlabel('Epoch')
    ax.set_title('Accuracy (%) on the test data')
    plt.show()

def plot_test_cost(test_cost, num_epochs, test_cost_xmin):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(test_cost_xmin, num_epochs),
           test_cost[test_cost_xmin:num_epochs],
```

```
            color='#2A6EA6')
    ax.set_xlim([test_cost_xmin, num_epochs])
    ax.grid(True)
    ax.set_xlabel('Epoch')
    ax.set_title('Cost on the test data')
    plt.show()

def plot_training_accuracy(training_accuracy, num_epochs,
                           training_accuracy_xmin,
training_set_size):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(training_accuracy_xmin, num_epochs),
           [accuracy*100.0/training_set_size
            for accuracy in
training_accuracy[training_accuracy_xmin:num_epochs]],
            color='#2A6EA6')
    ax.set_xlim([training_accuracy_xmin, num_epochs])
    ax.grid(True)
    ax.set_xlabel('Epoch')
    ax.set_title('Accuracy (%) on the training data')
    plt.show()

def plot_overlay(test_accuracy, training_accuracy, num_epochs, xmin,
                 training_set_size):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(xmin, num_epochs),
           [accuracy/100.0 for accuracy in test_accuracy],
           color='#2A6EA6',
            label="Accuracy on the test data")
    ax.plot(np.arange(xmin, num_epochs),
```

```
            [accuracy*100.0/training_set_size
             for accuracy in training_accuracy],
            color='#FFA933',
             label="Accuracy on the training data")
    ax.grid(True)
    ax.set_xlim([xmin, num_epochs])
    ax.set_xlabel('Epoch')
    ax.set_ylim([90, 100])
    plt.legend(loc="lower right")
    plt.show()

if __name__ == "__main__":
    filename = raw_input("Enter a file name: ")
    num_epochs = int(raw_input(
        "Enter the number of epochs to run for: "))
    training_cost_xmin = int(raw_input(
        "training_cost_xmin (suggest 200): "))
    test_accuracy_xmin = int(raw_input(
        "test_accuracy_xmin (suggest 200): "))
    test_cost_xmin = int(raw_input(
        "test_cost_xmin (suggest 0): "))
    training_accuracy_xmin = int(raw_input(
        "training_accuracy_xmin (suggest 0): "))
    training_set_size = int(raw_input(
        "Training set size (suggest 1000): "))
    lmbda = float(raw_input(
        "Enter the regularization parameter, lambda (suggest: 5.0):
"))
    main(filename, num_epochs, training_cost_xmin,
        test_accuracy_xmin, test_cost_xmin, training_accuracy_xmin,
         training_set_size, lmbda)
```

Classe Generate_gradient

```
import json
import math
import random
import shutil
import sys
import matplotlib.pyplot as plt
import numpy as np
import network2
sys.path.append("../src/")
import mnist_loader


def main():
    # Carregando os dados
    full_td, _, _ = mnist_loader.load_data_wrapper()
    td = full_td[:1000]
   epochs = 500
   print ("\nDuas Camadas Ocultas:")
    net = network2.Network([784, 30, 30, 10])
    initial_norms(td, net)
    abbreviated_gradient = [
        ag[:6] for ag in get_average_gradient(net, td)[:-1]]
   f = open("initial_gradient.json", "w")
    json.dump(abbreviated_gradient, f)
    f.close()
    shutil.copy("initial_gradient.json",
"../../js/initial_gradient.json")
    training(td, net, epochs, "norms_during_training_2_layers.json")
    plot_training(
```

```
        epochs, "norms_during_training_2_layers.json", 2)
    print ("\nTrês Camadas Ocultas:")
    net = network2.Network([784, 30, 30, 30, 10])
    initial_norms(td, net)
    training(td, net, epochs, "norms_during_training_3_layers.json")
    plot_training(
        epochs, "norms_during_training_3_layers.json", 3)
    print ("\nQuatro Camadas Ocultas:")
    net = network2.Network([784, 30, 30, 30, 30, 10])
    initial_norms(td, net)
    training(td, net, epochs, "norms_during_training_4_layers.json")
    plot_training(epochs, "norms_during_training_4_layers.json", 4)

def initial_norms(training_data, net):
    average_gradient = get_average_gradient(net, training_data)
    norms = [list_norm(avg) for avg in average_gradient[:-1]]
    print ("Average gradient for the hidden layers: "+str(norms))

def training(training_data, net, epochs, filename):
    norms = []
    for j in range(epochs):
        average_gradient = get_average_gradient(net, training_data)
        norms.append([list_norm(avg) for avg in average_gradient[:-
1]])
        print ("Epoch: %s" % j)
        net.SGD(training_data, 1, 1000, 0.1, lmbda=5.0)
    f = open(filename, "w")
    json.dump(norms, f)
    f.close()

def plot_training(epochs, filename, num_layers):
    f = open(filename, "r")
```

```
    norms = json.load(f)
    f.close()
    fig = plt.figure()
    ax = fig.add_subplot(111)
    colors = ["#2A6EA6", "#FFA933", "#FF5555", "#55FF55", "#5555FF"]
    for j in range(num_layers):
        ax.plot(np.arange(epochs),
                [n[j] for n in norms],
                color=colors[j],
                 label="Hidden layer %s" % (j+1,))
    ax.set_xlim([0, epochs])
    ax.grid(True)
    ax.set_xlabel('Number of epochs of training')
    ax.set_title('Speed of learning: %s hidden layers' % num_layers)
    ax.set_yscale('log')
    plt.legend(loc="upper right")
    fig_filename = "training_speed_%s_layers.png" % num_layers
    plt.savefig(fig_filename)
    shutil.copy(fig_filename, "../../images/"+fig_filename)
    plt.show()

def get_average_gradient(net, training_data):
    nabla_b_results = [net.backprop(x, y)[0] for x, y in
training_data]
    gradient = list_sum(nabla_b_results)
    return [(np.reshape(g, len(g))/len(training_data)).tolist()
            for g in gradient]

def zip_sum(a, b):
   return [x+y for (x, y) in zip(a, b)]

def list_sum(l):
```

```
    return reduce(zip_sum, l)

def list_norm(l):
    return math.sqrt(sum([x*x for x in l]))

if __name__ == "__main__":
    main()
```

Classe Test

```
import mnist_loader
import network2
import pickle
import gzip
import numpy as np
training_data, validation_data, test_data =
mnist_loader.load_data_wrapper()
training_data = list(training_data)
net = network2.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 1.5, test_data=test_data)
```

RELATÓRIO FINAL DE CURSO Nº 16/2025 - CEENP (11.51.21)

(Nº do Protocolo: NÃO PROTOCOLADO)

(Assinado digitalmente em 09/04/2025 18:20)
MARCIO WLADIMIR SANTANA
PROFESSOR ENS BASICO TECN TECNOLOGICO
CTETTNP (11.50.36)
Matrícula: ###520#9

Visualize o documento original em https://sig.cefetmg.br/documentos/ informando seu número: ***16****, ano:* ***2025****, tipo:* ***RELATÓRIO FINAL DE CURSO****, data de emissão:* ***09/04/2025*** *e o código de verificação:* ***3f271ece0f***