VITOR ARINS PINTO

REDES NEURAIS CONVOLUCIONAIS DE PROFUNDIDADE PARA RECONHECIMENTO DE TEXTOS EM IMAGENS DE CAPTCHA.

Florianópolis

22 de outubro de 2016

VITOR ARINS PINTO

REDES NEURAIS CONVOLUCIONAIS DE PROFUNDIDADE PARA RECONHECIMENTO DE TEXTOS EM IMAGENS DE CAPTCHA.

Trabalho de Conclusão de Curso submetido ao Programa de graduação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Sistemas de Informação.

Orientadora: Luciana de Oliveira Rech

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Florianópolis

22 de outubro de 2016

AGRADECIMENTOS

RESUMO

Atualmente muitas aplicações na Internet seguem a política de manter alguns dados acessíveis ao público. Para isso é necessário desenvolver um portal que seja robusto o suficiente para garantir que todas as pessoas possam acessá-lo. Porém as requisições feitas para recuperar dados públicos nem sempre vêm de um ser humano. Empresas especializadas em Big data possuem um grande interesse em fontes de dados públicos para poder fazer análises e previsões a partir de dados atuais. Com esse interesse, Web Crawlers são implementados. Eles são responsáveis por consultar fontes de dados milhares de vezes ao dia, fazendo diversas requisições a um site. Tal site pode não estar preparado para um volume de consultas em um período tão curto de tempo. Com o intuito de impedir que sejam feitas consultas por programas de computador, as instituições que mantêm dados públicos investem em ferramentas chamadas CAPTCHA (teste de Turing público completamente automatizado, para diferenciação entre computadores e humanos). Essas ferramentas geralmente se tratam de imagens contendo um texto qualquer e o usuário deve digitar o que vê na imagem. O objetivo do trabalho proposto é realizar o reconhecimento de texto em imagens de CAPTCHA através da aplicação de redes neurais convolucionais.

ABSTRACT

Currently many applications on the Internet follow the policy of keeping some data accessible to the public. For this it is necessary to develop a portal that is robust enough to ensure that all people can access this data. But the requests made to recover Public Data not always come from a human. companies specializing in Big data have a great interest in data from public sources in order to make analyzes and forecasts from current data. With this interest, Web Crawlers are implemented. They are responsible for querying data sources thousands of times a day, making several requests to a site. This site may not be prepared for a volume of inquiries in a short period of time. In order to prevent queries to be made by computer programs, institutions that keep public data invest in tools called CAPTCHA (*Completely Automated Public Turing test to tell Computers and Humans Apart*). These tools usually deal with images containing text and the user must enter what he or she sees in the image. The objective of the proposed work is to perform the text recognition in CAPTCHA images through the application of convolutional neural networks.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CAPTCHA ..... *Completely Automated Public Turing test to tell Computers and Humans Apart*
AWS ..... *Amazon Web Services*
IA ..... *Inteligência Artificial*
DCNN ..... *Deep Convolutional Neural Networks*
ReLU ..... *Rectified Linear Unit*
GPU ..... *Graphical Processing Unit*
GPGPU ..... *General Purpose Graphical Processing Unit*
MNIST ..... *Mixed National Institute of Standards and Technology*
RAM ..... *Random Access Memory*
ASCII ..... *American Standard Code for Information Interchange*
RGB ..... *Red Green Blue*

SUMÁRIO

1 INTRODUÇÃO

Redes neurais artificiais clássicas existem desde os anos 60, como fórmulas matemáticas e algorítimos. Atualmente os programas de aprendizado de máquina contam com diferentes tipos de redes neurais. Um tipo de rede neural muito utilizado para processamento de imagens é a rede neural convolucional de profundidade. O trabalho em questão tratará da utilização e configuração de uma rede neural convolucional de profundidade para reconhecimento de textos em imagens específicas de CAPTCHAs.

1.1 PROBLEMA

Com o aumento constante na quantidade de informações geradas e computadas atualmente, percebe-se o surgimento de uma necessidade de tornar alguns tipos de dados acessíveis a um público maior. A fim de gerar conhecimento, muitas instituições desenvolvem portais de acesso para consulta de dados relevantes a cada pessoa. Esses portais, em forma de aplicações na Internet, precisam estar preparados para receber diversas requisições e em diferentes volumes ao longo do tempo.

Devido a popularização de ferramentas e aplicações especializadas em Big data, empresas de tecnologia demonstram interesse em recuperar grandes volumes de dados de diferentes fontes públicas. Para a captura de tais dados, Web crawlers são geralmente implementados para a realização de várias consultas em aplicações que disponibilizam dados públicos.

Para tentar manter a integridade da aplicação, as organizações que possuem estas informações requisitadas investem em ferramentas chamadas CAPTCHA (teste de Turing público completamente automatizado para diferenciação entre computadores e humanos). Essas ferramentas frequentemente se tratam de imagens contendo um texto qualquer e o usuário precisa digitar o que vê na imagem.

O trabalho de conclusão de curso proposto tem a intenção de retratar a ineficiência de algumas ferramentas de CAPTCHA, mostrando como redes neurais convolucionais podem ser

aplicadas em imagens a fim de reconhecer o texto contido nestas imagens.

1.2 OBJETIVOS

1.2.1 *OBJETIVO GERAL*

Analisar o treinamento e aplicação de redes neurais convolucionais de profundidade para o reconhecimento de texto em imagens de CAPTCHA.

1.2.2 *OBJETIVOS ESPECÍFICOS*

- Estudar trabalhos correlatos e analisar o estado da arte;
- Entender como funciona cada aspecto na configuração de uma rede neural convolucional;
- Realizar o treinamento e aplicação de uma rede neural artificial para reconhecimento de CAPTCHAs.

1.3 ESCOPO DO TRABALHO

O escopo deste trabalho inclui o estudo e análise de uma rede neural convolucional de profundidade para reconhecimento de texto em imagens de um CAPTCHA específico.

Não está no escopo do trabalho:

- Analisar outras formas de inteligência no reconhecimento de texto.
- O estudo, análise ou implementação da aplicação de redes neurais convolucionais para outros tipos de problemas.
- O estudo, análise ou implementação de softwares do tipo “crawler” ou qualquer programa automatizado para recuperar quaisquer informações de websites públicos.
- A análise e comparação de diferentes técnicas ou parâmetros para otimização de redes neurais.

1.4 METODOLOGIA

Para realizar o proposto, foram feitas pesquisas em base de dados tais como IEE Xplorer e ACM Portal. Adquirindo assim maior conhecimento sobre o tema, estudando trabalhos relacionados.

Com base no estudo do estado da arte, foram feitas pesquisas e estudos para indicar caminhos possíveis para desenvolvimento da proposta de trabalho.

1.5 ESTRUTURA DO TRABALHO

Para uma melhor compreensão e separação dos conteúdos, este trabalho está organizado em 6 capítulos. Sendo este o capítulo 1 cobrindo a introdução ao tema, citando os objetivos e explicando a proposta.

O capítulo 2 apresenta a fundamentação teórica, com as definições das abordagens de desenvolvimento de aprendizado de máquina e redes neurais. Os conceitos de tipos de redes neurais.

No capítulo 3 está a proposta de experimento a ser realizado. Assim como uma breve ideia dos resultados esperados e a forma de avaliação dos mesmos.

O capítulo 4 contém as informações do desenvolvimento do sistema de reconhecimento de imagens de CAPTCHA. Também a apresentação dos dados obtidos através das metodologias escolhidas na seção anterior.

No capítulo 5 são apresentados os resultados da aplicação do sistema de reconhecimento de imagens de CAPTCHA.

Por fim, no capítulo 6 estão as conclusões obtidas através dos resultados deste trabalho, as ameaças que podem comprometer o acesso à dados públicos disponbilizados e as sugestões para trabalhos futuros relacionados.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 APRENDIZADO DE MÁQUINA

Aprendizado de máquina, ou *Machine Learning*, é uma área da computação que emergiu de estudos relacionados ao reconhecimento de padrões e inteligência artificial. Nesta área é contemplado o estudo e implementação de algoritmos que conseguem aprender e fazer previsões baseadas em dados. Esses algoritmos funcionam através da construção de um modelo preditivo que tem como entrada um conjunto de treinamento com dados de observações quaisquer. Desse modo as previsões são feitas orientadas aos dados e não a partir de instruções estáticas de um programa.

2.2 REDES NEURAIS

Diante das ferramentas disponíveis que tratam de aprendizado de máquina, uma delas é a rede neural artificial.

Redes neurais artificiais são conjuntos de modelos inspirados por redes neurais biológicas, usados para aproximar funções que dependem de um número muito grande de entradas. De acordo com Mackay[1], Redes neurais geralmente são especificadas utilizando 3 coisas:

- **Arquitetura:** Especifica quais variáveis estão envolvidas na rede e quais as relações topológicas. Por exemplo, as variáveis envolvidas em uma rede neural podem ser os pesos das conexões entre os neurônios.
- **Regra de atividade:** A maioria dos modelos de rede neural tem uma dinâmica de atividade com escala de tempo curta. São regras locais que definem como as "atividades" de neurônios mudam em resposta aos outros. Geralmente a regra de atividade depende dos parâmetros da rede.
- **Regra de aprendizado:** Especifica o modo com que os pesos da rede neural muda conforme o tempo. O aprendizado normalmente toma uma escala de tempo maior do que a

escala referente a dinâmica de atividade. Normalmente a regra de aprendizado dependerá das "atividades" dos neurônios. Também pode depender dos valores que são objetivos definidos pelo usuário e valores iniciais dos pesos.

Tomando imagens como exemplo, uma rede neural para reconhecimento de texto pode ter como entrada o conjunto de pixels[1] da imagem. Depois de serem atribuídos os pesos para cada item da entrada, os próximos neurônios serão ativados mediante a função de atividade pré-definida. Os pesos são recalculados através da regra de aprendizado e todo processo é repetido até uma condição determinada pelo usuário.

2.3 REGRESSÃO LOGÍSTICA MULTINOMIAL

Regressão logística multinomial é um método de classificação que consiste em um modelo que é usado para prever probabilidades de variáveis associadas a uma determinada classe, baseado em um conjunto de variáveis independentes. Para construir este modelo, esta seção descreve as tarefas e cálculos principais.

2.3.1 *CLASSIFICAÇÃO SUPERVISIONADA*

Classificação é uma tarefa central para o aprendizado de máquina, e consiste em receber uma entrada, como a imagem da letra "A" por exemplo, e dar um rótulo que diz que essa imagem é da classe "A". Geralmente temos muitos exemplos da entidade que queremos classificar. Esses exemplos já mapeados com seu respectivo rótulo, são chamados de conjunto de treinamento. Após o treinamento o objetivo é receber um exemplo completamente novo e descobrir em qual classe esse exemplo se encaixa.

Dizemos que esse aprendizado é supervisionado pois cada exemplo recebeu um rótulo durante o treinamento. Ao contrário deste, o aprendizado não supervisionado não conhece os rótulos de cada exemplo, mas tenta agrupar os exemplos que possuem semelhança baseado em propriedades úteis encontradas ao longo do treinamento.

2.3.2 *CLASSIFICADOR LOGÍSTICO*

Um classificador logístico, ou linear, recebe como entrada os pixels de uma imagem por exemplo, e aplica uma função linear a eles para gerar suas predições. Uma função linear é

[1] pixel é o menor ponto que forma uma imagem digital, sendo que o conjunto de milhares de pixels formam a imagem inteira. Cada Pixel é composto por um conjunto de 3 pontos: verde, vermelho e azul.

apenas uma grande multiplicação de matriz. Recebe todas as entradas como um grande vetor que será chamado de “X”, e multiplica os valores desse vetor com uma matriz para gerar as predições, cada predição é como uma pontuação, que possui o valor que indica o quanto as entradas se encaixam em uma classe de saída.

$$WX + b = Y \tag{2.1}$$

“X” é como chamaremos nosso vetor das entradas, “W” serão pesos e o termo tendencioso (*bias*) será representado por “b”. “Y” corresponde ao vetor de pontuação para cada classe. Os pesos da matriz e o *bias* é onde age o aprendizado de máquina, ou seja, é necessário tentar encontrar valores para os pesos e para o *bias* que terão uma boa performance em fazer predições para as entradas.

2.3.3 *INICIALIZAÇÃO DE PESOS XAVIER*

Uma tarefa crucial para o sucesso na construção de redes neurais é a inicialização da matriz de pesos. Geralmente os pesos são inicializados de maneira aleatória. No caso do trabalho proposto, os valores serão inicializados de forma aleatória seguindo uma regra de distribuição, utilizando a inicialização de Xavier[2].

Se os pesos forem inicializados com um valor muito baixo, é capaz que as ativações da rede neural diminuam ao passar por cada camada. Já com uma inicialização de valores muito altos para o peso, as ativações podem acabar crescendo demais ao longo das camadas. A inicialização de Xavier garante que os pesos serão inicializados na “medida certa”, mantendo as ativações em uma variação razoável de valores mediante várias camadas da rede neural. A distribuição segue a seguinte fórmula:

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{2.2}$$

Onde W é a distribuição da inicialização para o peso em questão, n_{in} é o número de neurônios de entrada, e n_{out} é o número de neurônios de saída.

2.3.4 *FUNÇÃO SOFTMAX*

Como cada imagem pode ter um e somente um rótulo possível, é necessário transformar essas pontuações em probabilidades. Queremos que a probabilidade de ser a classe correta seja muito perto de **1.0** e a probabilidade para todas as outras classes fique perto de **0.0**. Para tornar

essas pontuações em probabilidades utilizamos uma função chamada ***Softmax***. Denotada na equação por "S".

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{2.3}$$

O mais importante dessa fórmula é que pode receber qualquer tipo de pontuação gerado por predições e transformá-la em probabilidades adequadas. Os valores dessas probabilidades serão altos quando a pontuação da classe for alta e baixos quando a pontuação da classe for baixa. A soma das probabilidades fica igual a 1.

Ao final do processo de aplicação da função linear e da função Softmax temos um vetor de tamanho igual ao número de classes possíveis e em cada posição do vetor temos a probabilidade para a classe referente a essa específica posição do vetor.

2.3.5 *ONE-HOT ENCODING*

Para facilitar o treinamento é preciso representar de forma matemática os rótulos de cada exemplo que iremos alimentar a rede neural. Cada rótulo será representado por um vetor de tamanho igual ao número de classes possíveis, assim como o vetor de probabilidades. No caso dos rótulos, será atribuído o valor de **1.0** para a posição referente a classe correta daquele exemplo e **0.0** para todas as outras posições. Essa tarefa é bem simples e geralmente chamada de *One-Hot Encoding*. Com isso é possível medir a eficiência do treinamento apenas comparando 2 vetores.

2.3.6 *CROSS ENTROPY*

O jeito mais comum em redes neurais de profundidade para medir a distância entre o vetor de probabilidades e o vetor correspondente ao rótulo se chama ***cross entropy***.

$$D(S,L) = -\sum_i L_i \log(S_i) \tag{2.4}$$

Na equação o *cross entropy* é representado por "D" que é a distância. "S" é o vetor de probabilidades vindo da função *Softmax* e "L" é o vetor referente ao rótulo do exemplo em questão.

2.3.7 *TREINAMENTO*

Com todas as tarefas e cálculos disponíveis, resta descobrir os valores dos pesos e *biases* mais adequados ao nosso modelo de regressão.

PERDA

Para cada valor aleatório de peso e *bias*, podemos medir a distância média para todas as entradas de todo o conjunto de treinamento e todos rótulos que estão disponíveis. Esse valor é chamado de **perda** do treinamento. Esta perda, que é a média de *cross entropy* de todo treinamento, é uma função grande e custosa.

$$L = \frac{1}{N}\sum_{i} D(S(WX_i + b), L_i) \tag{2.5}$$

Cada exemplo no conjunto de treinamento é multiplicado por uma grande matriz "W". Depois é tudo adicionado em um grande somatório.

O objetivo é que as distâncias sejam minimizadas, o que significa que a classificação está indo bem para todos os exemplos dos dados de treinamento. Portanto queremos que nossa perda seja pequena. A perda nada mais é que uma função em relação aos pesos e *biases*. Assim é necessário tentar minimizar essa função, tornando um problema de aprendizado de máquina em um problema de otimização numérica.

2.3.8 *OVERFITTING*

Segundo Goodfellow[3],no aprendizado de máquina há dois fatores que são desafios centrais para os pesquisadores: *underfitting* e *overfitting*. *Underfitting* acontece quando o modelo não está apto para obter um valor de perda suficientemente baixo com o conjunto de dados de treinamento. Isso varia de acordo com o problema que se está querendo resolver.

Já o *overfitting* ocorre quando a diferença é muito grande entre o valor de perda para o conjunto de treinamento e o valor de perda para o conjunto de teste. Podemos controlar se um modelo fica mais propenso ao *overfit* ou ao *underfit* alterando sua **capacidade**. Informalmente, a capacidade de um modelo é sua habilidade de se encaixar em uma grande variedade de funções. Modelos com baixa capacidade terão mais trabalho para se encaixar em um conjunto de dados. Enquanto modelos com alta capacidade podem se encaixar muito bem e acabar memorizando propriedades do conjunto de treinamento que não servem para o conjunto de teste.

2.3.9 *MÉTODO DO GRADIENTE*

O jeito mais simples de otimização numérica é utilizando método do gradiente (ou *Gradient Descent* em inglês).

$$w \leftarrow w - \alpha \Delta_w L \tag{2.6}$$

$$b \leftarrow b - \alpha \Delta_b L \tag{2.7}$$

Este método calcula a derivada da função de perda em relação a cada peso(w) e cada *bias*(b), assim computando um novo valor para essas variáveis e indo na direção oposta à derivada.

Para o treinamento funcionar esse processo será executado dezenas ou centenas de vezes até encontrar os valores ideais de pesos e *biases*.

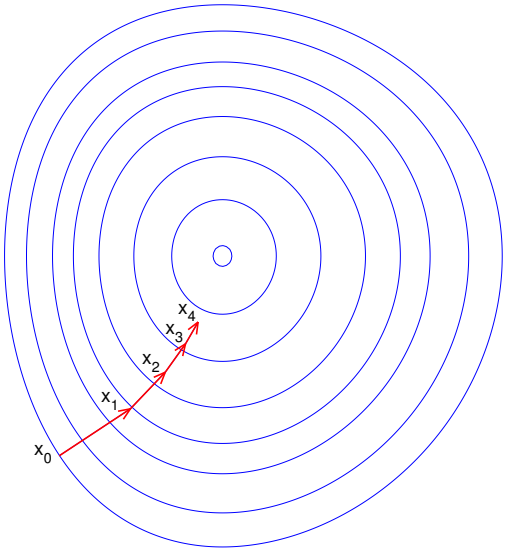

Figura 1: Os círculos na imagem representam a função de perda quando há apenas 2 parâmetros de peso como exemplo, a função será maior em algumas áreas e menor em outras. Tentaremos encontrar os pesos que fazem com que a perda seja reduzida. Portanto o método do Gradiente irá calcular a derivada da perda em relação aos parâmetros de peso e dar um passo na direção oposta $(x_0, x_1, ..., x_n)$, que significa calcular novos pesos para minimizar a perda.

2.4 APRENDIZADO EM PROFUNDIDADE

O aprendizado em profundidade permite que modelos computacionais compostos por múltiplas camadas de processamento possam aprender representações de dados com múltiplos níveis de abstração[4]. Essa técnica de aprendizado começa a ficar mais famosa depois de 2 adventos específicos da computação: a geração de enormes volumes de dados e a utilização de GPUs

para propósitos gerais (GPGPU).

A solução de *Deep learning* permite que computadores aprendam a partir de experiências e entendam o mundo em termos de uma hierarquia de conceitos, com cada conceito definido em termos da sua relação com conceitos mais simples. Juntando conhecimento de experiência, essa abordagem evita a necessidade de ter operadores humanos especificando formalmente todo o conhecimento que o computador precisa. A hierarquia de conceitos permite que o computador aprenda conceitos complexos construindo-os à partir de conceitos mais simples. Desenhando um gráfico que mostra como esses conceitos são construídos em cima de outros, o gráfico fica profundo, com muitas camadas. Por esta razão, essa abordagem para IA é chamada de Aprendizado em profundidade[3].

2.4.1 *OTIMIZAÇÃO COM SGD*

O algoritmo SGD (do inglês, *Stochastic Gradient Descent*) é uma peça chave de *Deep learning*. Praticamente todo o aprendizado em profundidade é alimentado por esse algoritmo muito importante. O problema do método do Gradiente visto anteriormente, é que o mesmo se torna muito difícil de escalar. Para cada vez que é calculada a perda do modelo, um computador pode levar em torno de 3 vezes esse tempo para calcular o gradiente.

Como foi dito anteriormente, um ponto crucial do aprendizado em profundidade é a utilização de uma grande quantidade de dados. Visto o tempo e a ineficiência do método do Gradiente, no algoritmo SGD é feita uma adaptação para realizar o treinamento sobre um conjunto de dados maior. Ao invés de calcular a perda, será calculada uma estimativa dessa perda. Esta estimativa será feita baseada na perda calculada para uma pequena parte do conjunto de dados do treinamento. Essa pequena fração terá entre 1 e 1000 exemplos dos dados e precisa ser escolhida aleatoriamente do conjunto de treinamento. Utilizando este método, a perda pode aumentar em alguns momentos, mas isto será compensado pois será possível executar esse processo muito mais vezes do que com o método do Gradiente comum. Ao longo do tempo, executar esses procedimentos por milhares ou milhões de vezes é muito mais eficiente do que utilizar somente o método do Gradiente.

2.4.2 *MOMENTUM*

Em cada iteração do processo de treinamento, será tomado um passo bem pequeno em uma direção aleatória que seria a mais indicada para diminuir a perda. Mas ao agregar todos esses passos chegamos na função com perda mínima. É possível tomar vantagem do conhecimento

acumulado de passos anteriores para saber qual direção tomar. Um jeito barato de fazer isto é manter uma média móvel[2] de todos os gradientes, e usar essa média móvel ao invés da direção do atual conjunto de dados. Essa técnica é chamada de *momentum* e geralmente leva a uma convergência melhor.

2.4.3 *DECLÍNIO DA TAXA DE APRENDIZADO*

Como foi dito anteriormente, em cada etapa do processo de treinamento é tomado um pequeno passo em direção a minimização da perda. A **taxa de aprendizado** é o parâmetro que diz o quão pequeno é esse passo. Existe uma área inteira de pesquisa sobre essa taxa, e os melhores resultados indicam que é mais apropriado decair a taxa de aprendizado ao longo do treinamento. Neste trabalho iremos aplicar um declínio exponencial à taxa de aprendizado.[5]

2.4.4 *RELU*

Modelos lineares são simples e estáveis numéricamente mas podem se tornar ineficientes ao longo do tempo. Portanto para adicionar mais camadas em nosso modelo, será necessário introduzir alguns cálculos não lineares entre camadas. Em arquiteturas de profundidade, as funções de ativação dos neurônios se chamam ReLUs, e são capazes de introduzir cálculos não lineares aos modelos que possuem mais de uma camada. Essas são as funções não lineares mais simples que existem, elas são lineares ($y = x$) se $\mathbf{x}$ é maior que $\mathbf{0}$, senão ficam iguais a $\mathbf{0}$ ($y = 0$). Isso simplifica o uso de *backpropagation* e evita problemas de saturação, fazendo o aprendizado ficar muito mais rápido.

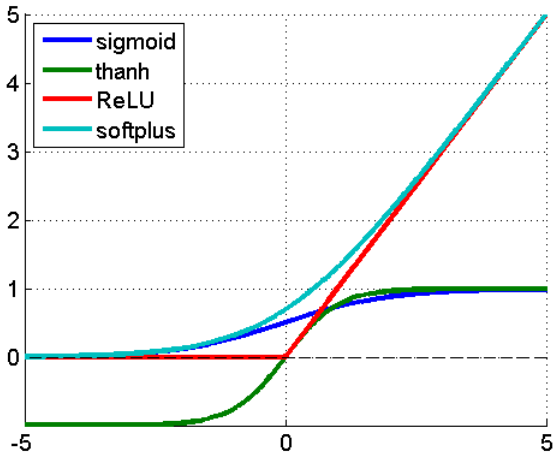

Figura 2: Comparação de funções de ativação.

[2]Média móvel é um cálculo que analisa pontos de dados criando séries de médias de diferentes subconjuntos de um conjunto completo de dados

CAMADA OCULTA

Como as unidades ReLU não precisam de parâmetros e não são observáveis fora da rede, a introdução dessas unidades entre camadas do modelo é chamada de camada oculta e pode possuir quantas unidades for necessário para uma melhor performance.

2.4.5 *BACKPROPAGATION*

Backpropagation é um método que faz o cálculo de derivadas de funções complexas eficientemente, contanto que estas funções sejam feitas de funções menores que possuem derivadas simples.

REGRA DA CADEIA

Um motivo de construir uma rede juntando operações simples é que torna a matemática muito mais simples. Com a regra da cadeia podemos concluir que para calcular a derivada de funções compostas, precisamos apenas calcular o produto das derivadas dos componentes.

Utilizando o método da cadeia, a maioria dos frameworks de aprendizado de máquina implementa o conceito de *backpropagation* automaticamente para o usuário. Assim é possível reutilizar dados pré-calculados e potencializar a eficiência do processo de treinamento.

2.4.6 *REGULARIZAÇÃO*

Regularizar significa aplicar restrições artificiais em sua rede que fazem com que o número de parâmetros livres reduza e isso não aumente a dificulade para otimizar. Essa é uma das formas de previnir o *overfitting* em nosso modelo pois adicionamos um fator externo que torna a rede mais flexível.

REGULARIZAÇÃO COM L_2

A ideia é adicionar um termo a mais à perda, o que dá uma penalidade em pesos maiores. Essa regularização é atingida adicionando a norma L_2 dos pesos a perda, multiplicada por uma constante (β) de valor baixo. Esta constante será mais um parâmetro que será necessário fornecer ao modelo para o treinamento.

$$L' = L + \beta \frac{1}{2} \|W\|_2^2 \tag{2.8}$$

2.4.7 *DROPOUT*

Outra forma de regularização que previne o *overfitting* é o *dropout*. Supondo que temos uma camada conectada à outra em nossa rede neural, os valores que vão de uma camada para a próxima podem se chamar de **ativações**. No *dropout*, são coletadas todas as ativações e aleatoriamente, para cada exemplo treinado, atribuímos valor 0 para metade desses valores. Basicamente metade dos dados que estão fluindo pela rede neural é destruído aleatoriamente.

Isso faz com que sua rede nunca dependa de nenhuma ativação estar presente pois ela pode ser destruída a qualquer momento. Por fim a rede neural é obrigada a aprender uma representação redundante de tudo para ter certeza que pelo menos alguma informação permaneça. Então algumas ativações serão removidas, mas sempre haverá uma ou mais ativações que fazem o mesmo trabalho e não serão removidas.

2.5 REDES NEURAIS CONVOLUCIONAIS DE PROFUNDIDADE

CNNs são a primeira abordagem verdadeiramente bem sucedida em aprendizado em profundidade onde muitas camadas de uma hierarquia são treinadas com sucesso de uma maneira robusta. Uma CNN é uma escolha de topologia ou arquitetura que se aproveita de relações espaciais para reduzir o número de parâmetros que devem ser aprendidos, e assim melhora o treinamento diante de uma rede com *feed-forward backpropagation*[6].

A grande vantagem na abordagem de redes neurais convolucionais de profundidade (CNN) para reconhecimento é que não é necessário um extrator de características desenvolvido por um ser humano. Nas soluções de [7] e [8] é possível perceber que foram usadas diversas camadas para o aprendizado das características.

Redes neurais convolucionais são muito similares a redes neurais comuns. De acordo com Karpathy[9]:

> “Arquiteturas de redes convolucionais assumem explicitamente que as entradas são imagens, o que nos permite cifrar algumas propriedades dentro da arquitetura. Essas então fazem a função de ativação mais eficiente de implementar e reduz drasticamente a quantidade de parâmetros na rede.” (KARPATHY, 2015, tradução nossa).

Portanto para o caso de reconhecimento de texto em imagens, as redes neurais convolucionais fazem muito sentido. Ao combinar o aprendizado em profundidade com redes convolucio-

nais, conseguimos tratar problemas muito mais complexos de classificação em imagens. Assim problemas mais simples, como o reconhecimento de textos, podem ser resolvidos cada vez mais rápido e facilmente.

2.5.1 *CAMADA CONVOLUCIONAL*

A camada de uma rede neural convolucional é uma rede que compartilha os seus parâmetros por toda camada. No caso de imagens, cada exemplo possui uma largura, uma altura e uma profundidade que é representada pelos canais de cor (Vermelho, Verde e Azul). Uma convolução consiste em pegar um trecho da imagem de exemplo e aplicar uma pequena rede neural que teria uma quantidade qualquer de saídas (K). Isso é feito deslizando essa pequena rede neural pela imagem sem alterar os pesos e montando as saídas verticalmente em uma coluna de profundidade K. No final será montada uma nova imagem de largura, altura e profundidade diferente. Essa imagem na verdade é um conjunto de **mapas de características** da imagem original. Como exemplo, estamos transformando 3 mapas de carcterísticas (canais de cores) para uma quantidade K de mapas de características.

Ao invés de apenas Vermelho, Verde e Azul, agora temos uma saída que possui vários canais de cor. O trecho de imagem é chamado de *Kernel*, e se for do tamanho da imagem inteira essa seria igual uma camada comum de uma rede neural. Mas como estamos trabalhando com este pequeno fragmento, temos bem menos pesos e eles são todos compartilhados pelo espaço da imagem.

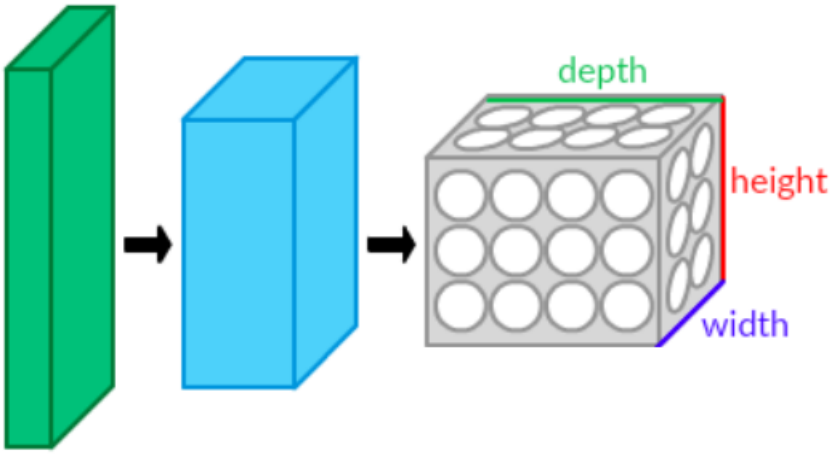

Figura 3: Para cada convolução, criamos uma nova imagem que possui uma nova largura (*width* em inglês), altura (*height* em inglês) e profundidade (*depth* em inglês)

Uma rede convolucional será basicamente uma rede neural de profundidade. Ao invés de

empilharmos camadas de multiplicação de matrizes, estamos empilhando convoluções. Portanto no começo teremos uma imagem grande que possui apenas os valores de pixel como informação. Assim aplicamos convoluções que irão "espremer" as dimensões espaciais e aumentar a profundidade. No final podemos conectar nosso classificador e podemos lidar apenas com parâmetros que mapeiam o conteúdo da imagem.[10]

STRIDE

Quando estamos realizando uma convolução, deslizamos uma janela com o tamanho do *Kernel* pela imagem, o *stride* é o parâmetro que diz quantos pixels de espaçamento teremos entre um fragmento da imagem e outro. Por exemplo um *stride* de 1 significa que a imagem de saída pode ter a mesma largura e altura que a imagem de entrada. Um valor de 2 significa que a imagem de saída pode ter metade do tamanho.

PADDING

O parâmetro de *padding* é o que se faz nas bordas das imagens de saída. Uma possibilidade é não deslizar o *Kernel* até as bordas da imagem, isso é chamado de ***valid padding***. Outra possibilidade é deslizar o seu *Kernel* até as bordas da imagem e completar com 0, essa técnica é chamada de ***same padding***.

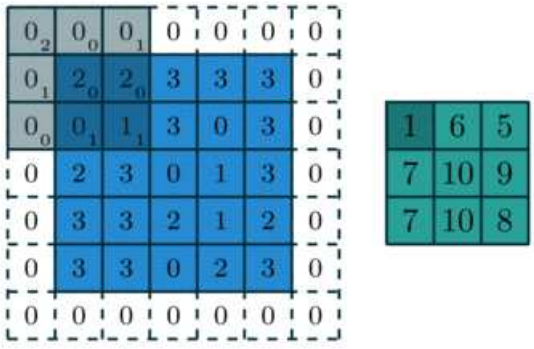

Figura 4: No exemplo temos uma imagem representada por uma matriz 5x5 e está sendo aplicado um *kernel* de tamanho 3x3, com o *stride* igual a 2 e um *same padding*, completando as bordas com 0. Isso gera uma nova imagem 3x3 por consequência dos parâmetros escolhidos.

2.5.2 *POOLING*

Reduzir as dimensões espaciais da nossa rede neural é primordial para uma arquitetura eficaz do modelo. No entanto utilizar uma convolução com *stride* igual a 2 para essa tarefa é

uma forma agressiva e arriscada para isso, pois podemos perder bastante informação no processo. Ao invés disso, iremos realizar convoluções com *stride* igual a 1, sem perder nenhuma informação da imagem original. Após a camada convolucional iremos adicionar uma camada de *pooling* que irá receber todas as convoluções e combiná-las de alguma forma[10].

MAX POOLING

Para cada ponto nos nossos mapas de características essa operação olha para uma pequena vizinhança ao redor deste ponto. Com esses valores em mãos é possível calcular o valor máximo dessa vizinhança.

Esta técnica geralmente leva a modelos mais eficazes. Porém a computação das convoluções com *stride* menor pode se tornar mais lenta. Além disso, agora será necessário trabalhar com mais parâmetros para nossa rede neural, o tamanho de região de *pooling* e o parâmetro de *stride* para o *pooling*.

2.5.3 *CAMADA COMPLETAMENTE CONECTADA*

De acordo com Krizhevsky[7], uma camada completamente conectada tem conexões com todas as ativações das camadas anteriores, assim como em redes neurais comuns. Suas ativações podem ser calculadas através de uma multiplicação de matrizes seguida da adição do fator *bias*.

3 *PROPOSTA DE EXPERIMENTO*

Para realizar o experimento será necessário treinar um modelo de rede neural que seja capaz, ou esteja próximo, de decifrar um CAPTCHA. Para isso serão efetuadas três etapas básicas e comuns quando se trabalha com redes neurais. Primeiro será coletado o maior número possível de imagens de CAPTCHA. Em seguida será gerado um dataset com as características dessas imagens junto com a classe em que pertence. A partir daí podemos realizar a configuração e treinamento da rede neural. E por fim será testada a acurácia do modelo mediante imagens de teste.

3.1 COLETA DE IMAGENS

Como o escopo do trabalho não contempla a automatização da recuperação de informações de websites públicos, foi disponibilizado um repositório com as imagens necessárias. Esse repositório possui 188.564 imagens e foi disponibilizado pela empresa Neoway. As imagens se tratam de um CAPTCHA publicado pelo site do SINTEGRA de Santa catarina (`http://sistemas3.sef.sc.gov.br/sintegra/consulta_empresa_pesquisa.aspx`).

Figura 5: Um exemplo do CAPTCHA utilizado pelo sistema de consulta do SINTEGRA de Santa Catarina.

3.1.1 *FONTE PÚBLICA*

Para demonstrar a ineficiência de certas imagens de CAPTCHA foi escolhido um software Web. Este software do SINTEGRA, fornece dados públicos de contribuintes mediante consulta via website. O SINTEGRA é o Sistema Integrado de Informações sobre Operações Interestaduais com Mercadorias e Serviçoes. Esta fonte pública possui dados fornecidos pelos próprios

contribuintes na hora do cadastro. Os comerciantes ou profissionais autônomos fazem seu cadastro para facilitar o comércio de produtos e prestação de serviços. O cadastro contempla inscrição da pessoa física ou jurídica, endereço e informações complementares referentes ao fisco estadual.

3.2 GERAÇÃO DO CONJUNTO DE DADOS

O conjunto de dados (ou "dataset") que alimenta a rede neural é gerado em tempo de execução do treinamento. Cada imagem é lida de seu diretório em disco e carregada na memória como uma matriz de valores de pixel. Ao final deste processo há um vetor em memória com todas imagens existentes já pré-processadas. Isso é feito para o dataset de treinamento e de teste. O dataset de treinamento terá a maioria das imagens, portanto **180 mil**.

3.2.1 *PRÉ-PROCESSAMENTO*

A fase de pré-processamento das imagens é mínima e é feita junto com a geração do conjunto de dados.

- **Escala de cinza**

  Ao gerar um array representativo da imagem, apenas é considerado um valor de escala de cinza da imagem, assim padronizando os valores de intensidade de pixels entre 0 e 1.

- **Redimensionamento**

  Ao gerar o array que representa a imagem, é feito um cálculo para diminuir a imagem com base em uma escala. Essa escala será configurada à partir de um valor padrão para a largura e altura das imagens.

3.2.2 *CONJUNTO DE DADOS DE TESTE*

Para o treinamento será necessário um conjunto separado para teste que não possui nenhuma imagem presente no conjunto de treinamento. O dataset de testes terá uma amostra bem menor que o conjunto de treinamento. Portanto terá **8 mil** imagens.

3.3 TREINAMENTO

Após gerado o conjunto de dados, é possível trabalhar no treinamento do modelo da rede neural. Para isso será usado o Framework **TensorFlow**[11] destinado à *Deep Learning* e um script em Python que fará uso das funções disponibilizadas pela biblioteca do TensorFlow. Assim realizando o treinamento até atingir um valor aceitável de acerto no conjunto de teste. O resultado do treinamento será um arquivo binário representando o modelo que será utilizado para avaliação posteriormente.

3.3.1 *INFRAESTRUTURA*

Com o intuito de acelerar o processo, foi utilizada uma máquina com **GPU** para o treinamento. A máquina foi adquirida em uma *Cloud* privada da AWS. A GPU utilizada se trata de uma *NVIDIA K80* com 2.496 cores e 12GB de memória de vídeo. Como processador a máquina possui um *Intel Xeon E5-2686v4 (Broadwell)* com 4 cores, e ainda possui 61GB de memória RAM [12].

3.3.2 *BIBLIOTECAS UTILIZADAS*

Todo o código foi implementado utilizando a linguagem de programação ***Python***, e as seguintes bibliotecas foram utilizadas:

- **TensorFlow**[11]: Um Framework implementado em *Python* destinado à *Deep Learning*. Proporciona a criação da arquitetura e automatização do processo de treinamento de redes neurais com *backpropagation*.
- **NumPy**[13]: Uma biblioteca em *Python* criada para computação científica. Possui um objeto de *array* com várias dimensões e várias funções sofisticadas para cálculos com algebra linear.
- **OpenCV**[14]: Uma biblioteca, implementada em C/C++, destinada à computação visual. Utilizada para ler imagens em disco e realizar o pré-processamento nas mesmas.

3.4 AVALIAÇÃO DE ACURÁCIA

Para a avaliação, uma nova amostra de imagens será coletada do mesmo modo que foram coletadas as imagens para treinamento. Essa amostra terá uma quantidade maior de imagens do

que o conjunto de teste.

Com essas amostra de imagens, será feita a execução do teste do modelo contra cada uma das imagens, assim armazenando uma informação de erro ou acerto do modelo. Ao final da execução será contabilizado o número de acertos e comparado com o número total da amostra de imagens para avaliação. Resultando assim em uma porcentagem que representa a acurácia do modelo gerado.

4 DESENVOLVIMENTO

Este capítulo descreve o desenvolvimento do projeto proposto. O projeto é composto da implementação do processador do conjunto de dados, da implementação do script que monta e treina a rede neural e da configuração da rede neural para otimização dos resultados. No início foi utilizado como base um código já existente destinado ao reconhecimento de dígitos em imagens. A partir daí foi construído o reconhecedor textos do trabalho.

4.1 CÓDIGO FONTE UTILIZADO COMO BASE

Como base da implementação deste trabalho, foram utilizados exemplos de código aberto disponíveis no repositório de códigos do *TensorFlow*[15]. No repositório há diversos tutoriais e exemplos que incentivam o auto aprendizado dos usuários. Um dos exemplos mais conhecido entre a comunidade é o reconhecedor de dígitos da base dados MNIST[16].

O reconhecedor utilizado como base funciona apenas para dígitos isolados em imagens separadas. Para o caso do trabalho em questão foi necessário adaptá-lo para reconhecer conjuntos com 5 dígitos ou letras em uma mesma imagem sem passar por um processo de segmentação antes do treinamento.

4.2 LEITURA E PRÉ-PROCESSAMENTO DO CONJUNTO DE DADOS

Para a leitura das imagens e pré-processamento do conjunto de dados, foi implementada uma classe chamada *OCR_data*. Esta classe utiliza a memória RAM para armazenar o conjunto de dados enquanto é processado pelo treinamento. Para Inicialização da classe, são necessários alguns parâmetros:

- Número de imagens que deve ser lido do disco.
- Diretório onde as imagens estão disponíveis.

- Número de classes que um caractere pode ter. Para o caso do trabalho esse número é igual a 36 pois cada caractere do CAPTCHA utilizado como exemplo pode ser somente uma letra minúscula sem acentos de “a” à “z” (26 letras) ou um número de “0” à “9” (10 dígitos).
- Tamanho da fração dos dado para cada iteração com treinamento.
- Tamanho da palavra contida no CAPTCHA. 5 para nosso caso.
- Altura da imagem. Número fixo em 60 para as imagens disponíveis.
- Largura da imagem. Número fixo em 180 para as imagens disponíveis.
- Altura definida para redimensionamento da imagem.
- Largura definida para redimensionamento da imagem.
- A quantidade de canais de cor.

4.2.1 *LEITURA DAS IMAGENS*

As imagens são carregadas utilizando *OpenCV*[14] com o método *imread*. Após a leitura precisamos fixar o seu rótulo para a utilização no treinamento. Como as imagens já estão nomeadas com o respectivo conteúdo da sua imagem, só o que precisamos fazer é um vetor utilizável desse texto.

Primeiro transformamos cada caractere em um número de 0 à 35. Fazemos isso recuperando o código ASCII de cada caractere e normalizando a sequência. Portanto, para os dígitos (0 à 9) que possuem códigos indo de 48 à 57, subtraímos 48. E para as letras (a à z) que possuem códigos indo de 97 à 122, subtraímos 87 e ficamos com números de 10 à 35.

Depois de traduzido o caractere para um número, precisamos criar o vetor do rótulo através do nosso algoritmo de *One-hot encoding*. Para isso criamos 5 vetores, um para cada caractere da imagem, e cada vetor possui 36 posições. Completamos todas as posições com 0 e em seguida é atribuído o número 1 para a posição referente ao carctér. A posição do caractere foi determinada pelo passo anterior, sendo igual o número correspondente ao caractere.

FRAÇAO DOS DADOS PARA TREINAMENTO

Como em cada iteração do treinamento será recuperado apenas uma fração dos dados, foi criado um método *next_batch* na classe *OCR_data*. Outra motivação para este método é a ne-

cessidade de recuperar uma amostra aleatória dos dados em cada iteração.

Portanto temos uma variável global na classe *OCR_data* que mantém o estado da posição que estamos no conjunto de dados. Após passar por todo o conjunto de dados, nosso método começa a fazer feita uma permutação aleatória para garantir que as posições recuperadas do conjunto de dados sejam completamente escolhidas ao acaso.

4.2.2 *PRÉ-PROCESSAMENTO DAS IMAGENS*

Como foi dito anteriormente, a fase de pré-processamento é mínima e requer apenas alguns parâmetros. Essa etapa é necessária para garantir uma velocidade maior no treinamento e também garantir uma eficiência maior como veremos a seguir.

QUANTIDADE DE CANAIS DE COR

No contexto do trabalho, não nos importamos com a cor de um caractere da imagem. Uma letra “A” pode ser vermelha, azul ou verde mas ainda terá que ser reconhecido como letra “A”. Com isso em mente podemos reduzir a quantidade de informações que nosso modelo precisa aprender. É reduzido também a complexidade dos cálculos feitos pelo modelo. Quando fazemos a leitura da imagem com a biblioteca OpenCV, indicamos um parâmetro que diz que a imagem deve ser lida em escala de cinza (*IMREAD_GRAYSCALE*). A escala de cinza de uma imagem representa para cada valor de pixel uma média dos valores de cor RGB da imagem. Para cada pixel é somado o valor de vermelho com os valores de verde e azul e dividido por 3. Com isso podemos normalizar nossos dados de entrada para um valor entre 0 e 1, onde 0 seria um ponto completamente preto e 1 seria branco.

TAMANHO DA IMAGEM

Outro modo de reduzir informações desnecessárias é redimensionando a imagem. Com a biblioteca *OpenCV* fazemos isso invocando a função *resize*. Nessa função passamos como parâmetro a largura e altura alvos, assim como o algoritmo que deve ser usado para a interpolação[1]. Foi escolhido tamanho de 88 de largura por 24 de altura pois esses valores correspondem a mais ou menos metade da imagem. Na seção de arquitetura da rede, também veremos que esses valores se encaixarão mais naturalmente no nosso modelo. Como algoritmo de interpolação foi escolhido a reamostragem utilzando a relação da área de pixel (opção *INTER_AREA* do

[1]Interpolação se trata do algoritmo utilizado para redimensionar a imagem. Esse algoritmo irá interpolar cada valor de pixel da imagem para obter uma nova imagem redimensionada.

OpenCV). Este algoritmo é o indicado pela própria biblioteca para reduzir imagens. Agora com menos dados a ser processados o treinamento terá uma velocidade maior.

Ao final da geração de conjunto de dados criamos dois arrays multidimensionais com a biblioteca *NumPy*. Um array é das imagens e terá a forma $quantidade\_de\_imagens\ x\ 88\ x\ 24\ x\ 1$, sendo a quantidade fornecida como parâmetro, 88 x 24 a largura e altura da imagem, e 1 é nossa quantidade de canais de cor (ou profundidade). O outro array é para os rótulos e terá a forma $quantidade\_de\_imagens\ x\ 180$, sendo a quantidade fornecida como parâmetro e 180 o tamanho do vetor do rótulo pois se trata de 36 classes possíveis multiplicado por 5 caracteres.

4.3 ARQUITETURA DA REDE NEURAL

Para a construção e treinamento da rede neural foi implementado um script em *Python* que possui toda a arquitetura da rede descrita de forma procedural. O framework *TensorFlow* chama a arquitetura dos modelos de *Graph* (ou grafo, em português).

A arquitetura implementada começa com uma camada de entrada, possui 4 camadas convolucionais, 1 camada completamente conectada e mais uma camada completamente conectada de saída com 5 saídas, uma para cada caractere da imagem. Entre uma e outra camada convolucional há uma camada de ativação (ReLu) e uma camada de *pooling*. Ao final da última camada convolucional e antes da camada de saída há uma camada de *dropout*, resultando em um total de 14 camadas sendo 11 visíveis e 3 ocultas.

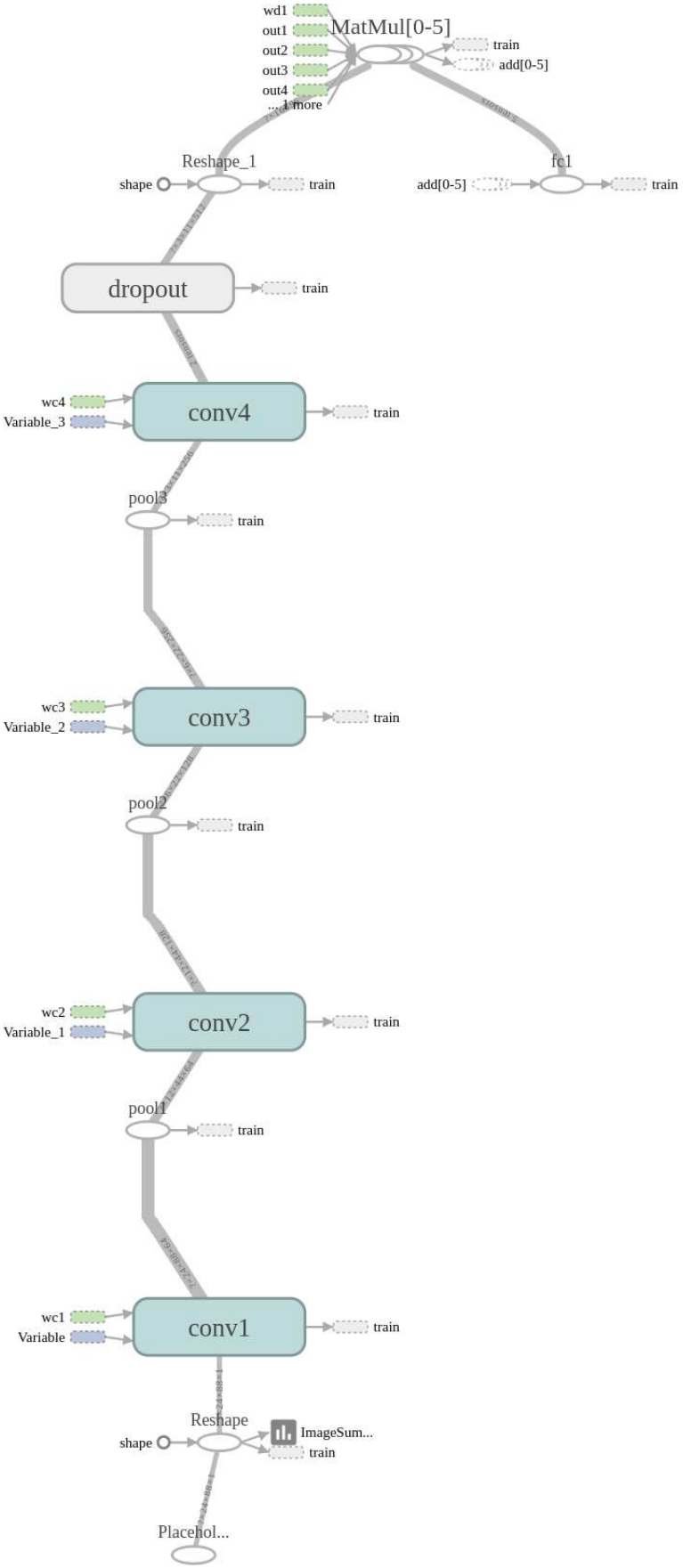

Figura 6: Arquitetura geral do modelo de rede neural treinado.

4.3.1 *ENTRADAS*

Nosso grafo começa com dois parâmetros de entrada, as imagens de entrada e os rótulos correspondentes. Para esses parâmetros criamos *placeholders* disponibilizados pelo framework. Esses *placeholders* inicialmente precisam saber qual tipo dos dados serão inseridos e o formato final. O tipo dos dados são os valores normalizados dos pixels das imagens portanto serão pontos flutuantes. O formato é o que foi definido em nossa classe do conjunto dos dados (*quantidade_de_imagens x* 88 *x* 24 *x* 1 para as imagens e *quantidade_de_imagens x* 180 para os rótulos).

4.3.2 *CAMADAS*

Para melhor visualização e compreensão da arquitetura será descrita cada camada utilizada por ordem de sequência da entrada até a saída.

1. Camada de entrada: um array multidimensional de formato **88x24x1** que será alimentado com os valores da imagem.

2. Camada convolucional (1): tem como entrada a imagem carregada na camada de entrada com **1** de profundidade. Executa convoluções aplicadas à imagem com um *kernel* de formato 5*x*5 e **64** valores de profundidade. Seu valor de *stride* é igual a **1** e utiliza *same padding*. Por fim é adicionado um *bias* de **64** valores à convolução. O formato do array multidimensional desta camada é **88x24x64**.

3. Camada oculta (1): utiliza **ReLU** como função de ativação e não recebe nenhum parâmetro. Tem como entrada a camada convolucional anterior.

4. Camada de *pooling* (1): executa a operação de *max pooling* com um *kernel* de formato 2*x*2 e *stride* igual a **2**. Essa operação tem como entrada a imagem gerada pelas convoluções após passar pela função de ativação. Isso irá reduzir o tamanho desta imagem pela metade. O formato do array multidimensional desta camada é **44x12x64**.

5. Camada convolucional (2): tem como entrada a imagem gerada nas camadas anteriores com **64** de profundidade. Executa convoluções aplicadas à imagem com um *kernel* de formato 5*x*5 e **128** valores de profundidade. Seu valor de *stride* é igual a **1** e utiliza *same padding*. Por fim é adicionado um *bias* de **128** valores à convolução. O formato do array multidimensional desta camada é **44x12x128**.

6. Camada oculta (2): utiliza **ReLU** como função de ativação e não recebe nenhum parâmetro. Tem como entrada a camada convolucional anterior.

7. Camada de *pooling* (2): executa a operação de *max pooling* com um *kernel* de formato 2*x*2 e *stride* igual a **2**. Essa operação tem como entrada a imagem gerada pelas convoluções após passar pela função de ativação. Isso irá reduzir o tamanho desta imagem pela metade. O formato do array multidimensional desta camada é **22x6x128**.

8. Camada convolucional (3): tem como entrada a imagem gerada nas camadas anteriores com **128** de profundidade. Executa convoluções aplicadas à imagem com um *kernel* de formato 5*x*5 e **256** valores de profundidade. Seu valor de *stride* é igual a **1** e utiliza *same padding*. Por fim é adicionado um *bias* de **256** valores à convolução. O formato do array multidimensional desta camada é **22x6x256**.

9. Camada oculta (3): utiliza **ReLU** como função de ativação e não recebe nenhum parâmetro. Tem como entrada a camada convolucional anterior.

10. Camada de *pooling* (2): executa a operação de *max pooling* com um *kernel* de formato 2*x*2 e *stride* igual a **2**. Essa operação tem como entrada a imagem gerada pelas convoluções após passar pela função de ativação. Isso irá reduzir o tamanho desta imagem pela metade. O formato do array multidimensional desta camada é **11x3x256**.

11. Camada convolucional (4): tem como entrada a imagem gerada nas camadas anteriores com **256** de profundidade. Executa convoluções aplicadas à imagem com um *kernel* de formato 3*x*3 e **512** valores de profundidade. Seu valor de *stride* é igual a **1** e utiliza *same padding*. Por fim é adicionado um *bias* de **512** valores à convolução. O formato do array multidimensional desta camada é **11x3x512**.

12. Camada de *dropout*: tem como entrada a camada convolucional anterior e possui um formato **11x3x512**. Recebe o valor de **0.75** como parâmetro de probabilidade de manter cada peso da rede neural.

13. Camada completamente conectada: tem como entrada todas as ativações das camadas anteriores. Para habilitar nossa camada completamente conectada precisamos realizar uma reformatação na matriz de entrada. Como a última camada possui um formato de **11x3x512**, multiplica-se esses valores para que ao invés de ter uma matriz, tenha-se um vetor de tamanho **16896** como entrada. Assim nossa camada completamente conectada terá **16896** ativações de entrada e **4096** ativações de saída.

14. Camadas completamente conectadas de saída: cada camada terá como entrada as **4096** ativações da camada anterior. E cada saída será um vetor de **36** posições que corresponde às probabilidades de classe para cada caractere. No total serão 5 camadas paralelas agregadas em uma.

4.4 CONFIGURAÇÃO DA REDE NEURAL

Parâmetros fornecidos para a configuração do treinamento da rede neural são chamados de **hiperparâmetros**. Dependendo da arquitetura utilizada, uma rede neural pode ter uma quantidade diferente de hiperparâmetros. A maioria dos hiperparâmetros utilizados no trabalho foram indicados artigos e trabalhos publicados sobre redes neurais convolucionais. Outro fator levado em consideração é a experiência do autor em relação ao tema em outras ocasiões.

4.4.1 *QUANTIDADE DE ATIVAÇÕES*

Os números de ativações 64, 128, 256, 512 e 4096 nas saídas das camadas foram utilizados com base em estudos anteriores feitos sobre redes convolucionais[7].

4.4.2 *TAMANHO DO KERNEL*

Baseado nos estudos de [8], foi escolhido um formato de 5x5 para o tamanho do *kernel* para a maioria das camadas. Para a última camada convolucional foi escolhido o tamanho de 3x3 pois o *kernel* não pode ter uma dimensão maior que a imagem de entrada. Como na última camada recebemos uma imagem no formato 11x3, não é possivel aplicar convoluções de tamanho 5x5.

4.4.3 *PARÂMETROS DO DECLÍNIO EXPONENCIAL DA TAXA DE APRENDIZADO*

Ao utilizar uma taxa de aprendizado decadente no otimizador, são fornecidos alguns parâmetros relativos ao processo de decadência da taxa. Os valores fornecidos tem como base um dos treinamentos de rede neural disponível em [15].

- **taxa de aprendizado inicial (*initial_learning_rate*):** é fornecido um valor de **0,01** para a taxa de aprendizado no início do treinamento.
- **passos para decair (*decay_steps*):** valor que indica a cada quantos passos a taxa de aprendizado deve diminuir. Esse valor é de **1.000** passos para o caso do trabalho.

- **taxa de decadência (*decay_rate*):** valor referente ao quanto a taxa de aprendizado deve decair. Foi escolhido **0,9** para o caso do trabalho, portanto a taxa de aprendizado vai cair 10% a cada 1.000 passos do treinamento.

4.4.4 *MOMENTUM*

A estratégia de *momentum* de nosso treinamento precisa de uma variável que será o fator determinante para o cálculo do gradiente. O valor dessa variável recomendado pela maioria dos estudos e exemplos é igual a **0,9** e é o valor utilizado no treinamento proposto.

4.4.5 *REGULARIZAÇÃO COM* L_2

Como foi dito no capítulo de fundamentação teórica, um parâmetro de regularização pode ser adicionado a perda do treinamento. Além da norma L_2 calculada baseada nos pesos, esse valor é multiplicado por uma variável β que tem valor igual a **0,0003** para o treinamento feito neste trabalho.

4.4.6 *PROBABILIDADE DO DROPOUT*

Cada valor das ativações terá uma probabilidade de ser mantido ou não. Como já foi contemplado na explicação do *dropout*, cada ativação pode ser removida entre uma camada e outra. Para os treinamentos realizados, foram utilizados dois valores como tentativa. O primeiro valor foi de **0,75** e o segundo foi **0,5**, isso dá 75% e 50% das ativações mantidas respectivamente. O segundo valor foi empregado na tentativa de minimizar o problema de *overfitting*.

4.4.7 *TAMANHO DA CARGA EM CADA PASSO*

Na otimização com SGD é fornecido um pedaço do conjunto de dados total em cada passo que calcula-se o método do gradiente. Este pedaço dos dados será chamado de “carga” (ou *batch* em inglês) para o presente contexto. Baseado em exemplos anteriores, foi escolhido um valor de **64** imagens para o tamanho de carga.

4.4.8 *NÚMERO DE ITERAÇÕES*

O número de iterações consiste basicamente na quantidade de exemplos que será calculado o gradiente. Este número leva em consideração o tamanho da carga e dá o resultado do número de passos que serão executados no treinamento. Para os treinamentos realizados neste

trabalho foram escolhidos dois valores, um com **200 mil** iterações outro com **500 mil** iterações. Portanto para um treinamento haverá **3.125** (200.000/64) passos e para os outros haverá **7.812** (500.000/64) passos.

5 TESTES

Neste capítulo são apresentados os testes realizados no sistema proposto com a arquitetura de rede neural definida no capítulo anterior. Todos os treinamentos foram realizados na infraestrutura citada no capítulo de proposta do trabalho. Os resultados foram satisfatórios para o conexto do trabalho. Para produzir os gráficos foi utilizada um ferramenta chamada ***TensorBoard***, que vem junto com a instalação do *TensorFlow*.

5.1 TREINAMENTO COM 200 MIL ITERAÇÕES

Inicialmente realizamos um treinamento com 200 mil iterações, portanto 3.125 passos com uma carga de 64 imagens. A fase de treinamento completa levou **1 hora 23 minutos e 54 segundos** para completar. Deve-se salientar que para o primeiro treinamento há uma espera maior devido ao *caching* dos dados. Isso é feito pelo sistema operacional para otimizar a memória da GPU e do sistema em geral quando os dados são carregados para a memória volátil.

Como é possível observar nos gráficos o valor da perda para este treinamento oscila entre 17,02 e 17,93 até o passo número 2.050 (iteração 131.200) onde a perda começa a decair. O mesmo acontece com a acurácia, ficando em torno de 5% até o passo 2.050 quando começa a subir. Ao final do treinamento foi alcançada uma acurácia de **80,94%** no conjunto de treinamento, **79,6%** no conjunto de teste e uma perda de **2,87** para o conjunto de treinamento, **2,91** para o conjunto de teste.

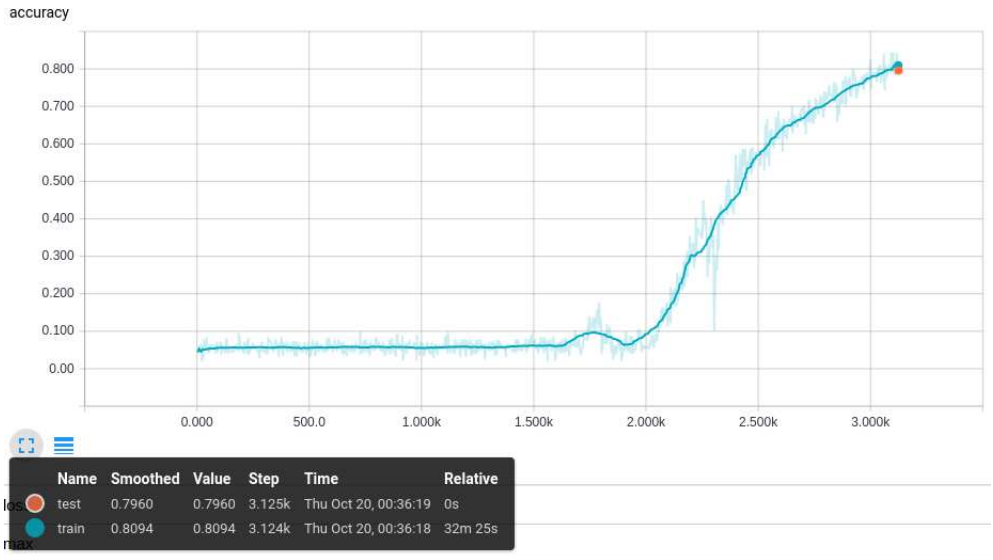

Figura 7: Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 200 mil iterações.

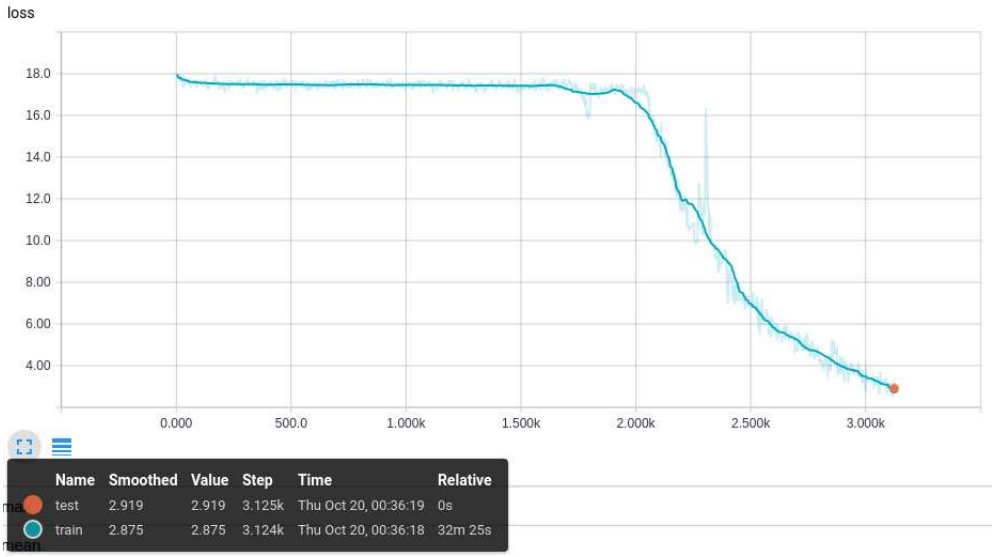

Figura 8: Gráfico da perda em relação ao número de passos para o treinamento da rede com 200 mil iterações.

Mesmo com o bom resultado nos testes, foi notado uma falta de estabilidade nos gráficos gerados. Analisando os gráficos de desvio padrão dos valores de pesos e *biases* das últimas camadas convolucionais, percebe-se que alguns valores poderiam continuar alterando se o treinamento continuasse por mais iterações.

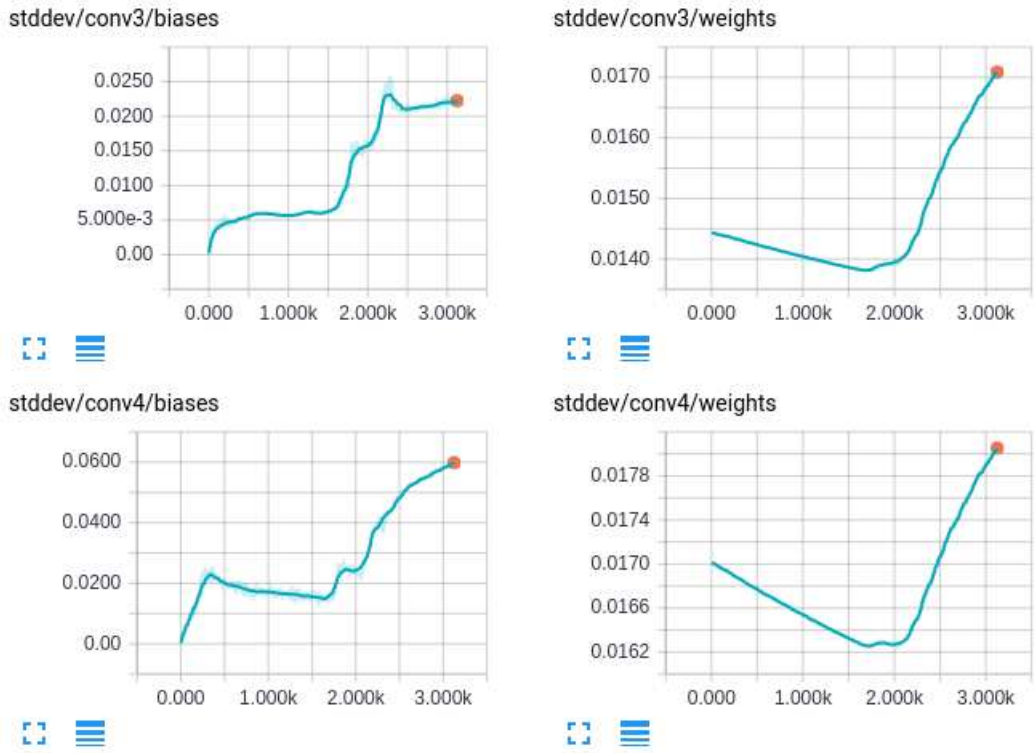

Figura 9: Desvio padrão dos pesos e *biases* em relação ao número de passos para o treinamento da rede com 200 mil iterações.

5.2 TREINAMENTO COM 500 MIL ITERAÇÕES

Visto a instabilidade nos valores de gráficos no treinamento anterior, a tentativa seguinte foi aumentar o número de iterações para 500 mil, portanto 7.812 passos. O tempo total de treinamento foi de **1 hora 18 minutos e 23 segundos**.

Analisando os gráficos gerados com este treinamento, novamente o valor da perda para oscila entre 16,87 e 17,51 até um certo ponto. Dessa vez é no passo número 2.922 (iteração 187.008) onde a perda começa a decair. O mesmo acontece com a acurácia, ficando em torno de 6% até o passo 2.977 (iteração 190.528) quando começa a subir. Ao final do treinamento foi alcançada uma acurácia de **97,81%** no conjunto de treinamento, **81,37%** no conjunto de teste e uma perda de **0,43** para o conjunto de treinamento, **13,52** para o conjunto de teste.

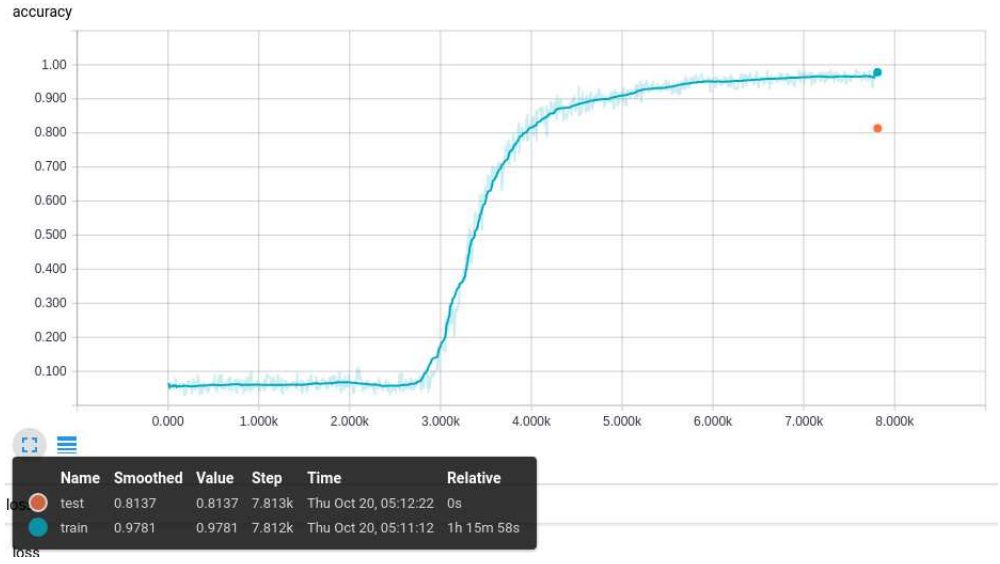

Figura 10: Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 500 mil iterações.

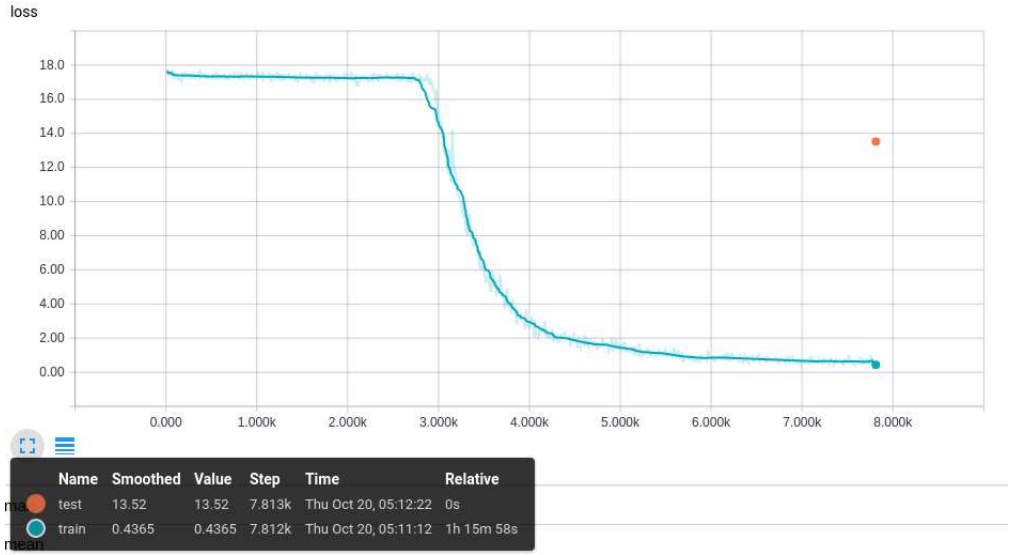

Figura 11: Gráfico da perda em relação ao número de passos para o treinamento da rede com 500 mil iterações.

Analisando os resultados, é possível observar que o valor da perda para o conjunto de treinamento é muito diferente do valor da perda para o conjunto de teste. Também nota-se que a acurácia no conjunto de treinamento chegou bem perto de 100%. De acordo com a fundamentação teórica, esses dois fatores podem ter sido causados pelo *overfitting* do modelo ao conjunto de dados do treinamento.

5.3 TREINAMENTO COM 500 MIL ITERAÇÕES E DROPOUT DE 50%

Na tentativa de minimizar os problemas encontrados anteriormente, foi realizado um terceiro treinamento. Foi visto que uma das técnicas de regularização para minimizar o *overfitting* é adicionando uma camada de *dropout* ao modelo. Nossa arquitetura já previa uma camada de *dropout*, no entanto o parâmetro de probabilidade de mantimento das ativações estava configurado para 75% (0,75). Para o terceiro treinamento foi configurada a probabilidade do *dropout* para 50% (0,5) e assim analisados os resultados. O tempo total de treinamento foi de **1 hora 18 minutos e 53 segundos**.

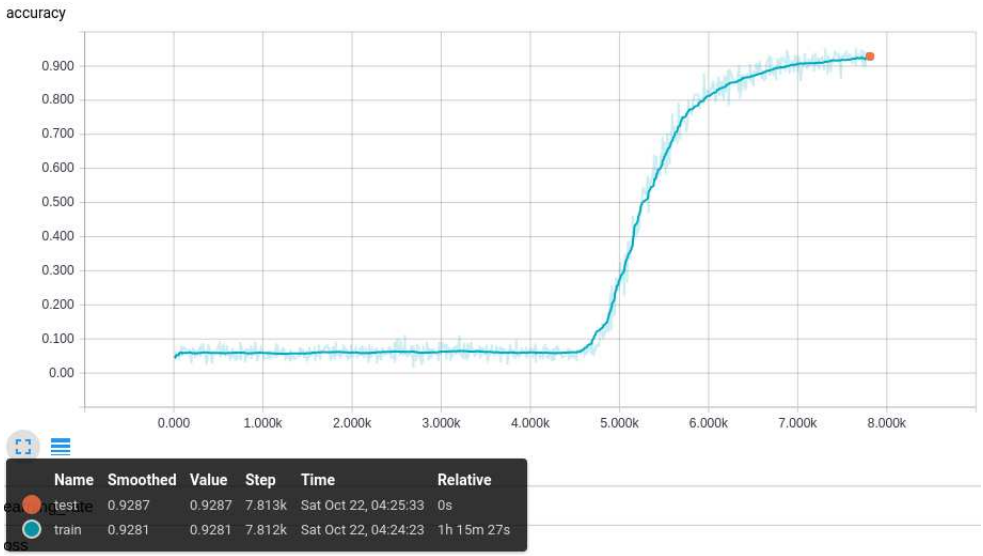

Figura 12: Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 500 mil iterações e probabilidade de *dropout* igual a 50%.

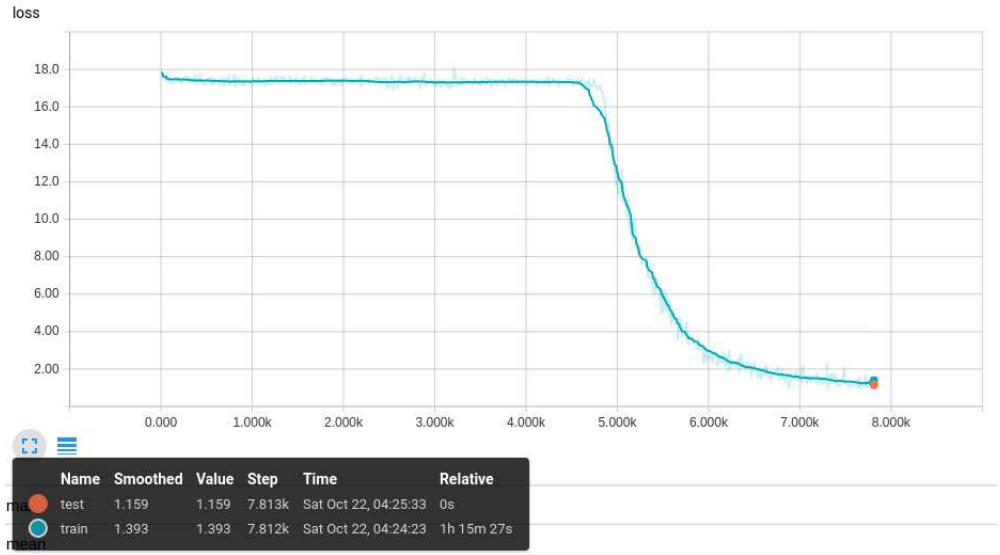

Figura 13: Gráfico da perda em relação ao número de passos para o treinamento da rede com 500 mil iterações e probabilidade de *dropout* igual a 50%.

Ao final do treinamento foi alcançada uma acurácia de **92,81%** no conjunto de treinamento, **92,87%** no conjunto de teste e uma perda de **1,39** para o conjunto de treinamento, **1,15** para o conjunto de teste. Vale lembrar que durante a fase de treinamento o algoritmo de otimização passa por uma carga de imagens completamente diferente em cada passo executado. Já na fase de teste o algoritmo passa por todas as 8 mil imagens de teste de uma só vez e fornece os valores registrados. Portanto no conjunto de treinamento os gráficos mostram que a acurácia chegou a **95,31%** e a perda chegou a **0,93** mas os valores registrados para estudo são os últimos valores de saída do treinamento.

5.4 RESULTADOS

De acordo com os testes realizados a configuração de alguns parâmetros no treinamento foram essenciais para eficácia do sistema proposto.

	200k it.	**500k it.**	**500k it. e 50% *dropout***
Acurácia (teste)	**79,6%**	**81,37%**	**92,87%**
Acurácia (treinamento)	80,94%	97,81%	92,81%
Perda (treinamento)	2,87	0,43	1,39
Perda teste	2,91	13,52	1,15

Tabela 1: Desempenho geral do sistema.

6 CONCLUSÕES

REFERÊNCIAS

[1] MACKAY, D. J. C. *Information Theory , Inference And Learning Algorithms*. Cambridge University Press, 2005. ISBN 9780521670517. Disponível em: <http://www.inference.phy.cam.ac.uk/mackay/itila/>.

[2] GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. Acessado: 21/10/2016. Disponível em: <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>.

[3] BENGIO, I. G. Y.; COURVILLE, A. Deep learning. Book in preparation for MIT Press. 2016. Disponível em: <http://www.deeplearningbook.org>.

[4] LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. Disponível em: <https://www.cs.toronto.edu/ hinton/absps/NatureDeepReview.pdf>.

[5] ZEILER, M. D. ADADELTA: AN ADAPTIVE LEARNING RATE METHOD.

[6] AREL, I.; ROSE, D. C.; KARNOWSKI, T. P. Deep Machine Learning—A New Frontier in Artificial Intelligence Research. 2010. Disponível em: <http://web.eecs.utk.edu/ itamar/Papers/DML_Arel_2010.pdf>.

[7] KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. Acessado: 21/10/2016. Disponível em: <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.

[8] GOODFELLOW, I. J. et al. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. Disponível em: <http://arxiv.org/pdf/1312.6082v4.pdf>.

[9] KARPATHY, A. *CS231n Convolutional Neural Networks for Visual Recognition*. Acessado: 06/08/2016. Disponível em: <http://cs231n.github.io/convolutional-networks/>.

[10] DUMOULIN, V.; VISIN, F.; BOX, G. E. P. A guide to convolution arithmetic for deep learning. 2016.

[11] TENSORFLOW. *TensorFlow — an Open Source Software Library for Machine Intelligence*. Acessado: 03/08/2016. Disponível em: <https://www.tensorflow.org/>.

[12] AWS. *EC2 Instance Types – Amazon Web Services (AWS)*. Acessado: 02/08/2016. Disponível em: <https://aws.amazon.com/ec2/instance-types/#gpu>.

[13] NUMPY. *NumPy is the fundamental package for scientific computing with Python.* Acessado: 20/10/2016. Disponível em: <http://www.numpy.org/>.

[14] OPENCV. *OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library.* Acessado: 20/10/2016. Disponível em: <http://opencv.org/>.

[15] TENSORFLOW; COMMUNITY. *tensorflow*. Acessado: 20/10/2016. Disponível em: <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/tutorials/>.

[16] LECUN, Y.; CORTES, C.; BURGES, C. J. *MNIST database*. Acessado: 20/10/2016. Disponível em: <http://yann.lecun.com/exdb/mnist/>.