VITOR ARINS PINTO

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

Florianópolis

6 de dezembro 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

6 de dezembro de 2016

Vitor Arins Pinto

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

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Sistemas de Informação", e aprovado em sua forma final pelo Curso de Bacharelado em Sistemas de Informação.

Prof. Renato Cislaghi, Dr.
Coordenador

Banca Examinadora:

Prof.ª Luciana de Oliveira Rech, Dr.ª
Orientadora

Prof. Mário Antônio Ribeiro Dantas, Dr.

Prof.ª Jerusa Marchi, Dr.ª

Florianópolis
2016

AGRADECIMENTOS

Primeiramente gostaria de agradecer à minha amada namorada e melhor amiga Letícia, que além de me mostrar o real significado do amor e me apresentar a felicidade de uma maneira que eu não conhecia, cooperou no desenvolvimento do trabalho desde o seu início, suportando amorosamente a minha ausência, se sobrecarregando com trabalho (inclusive o de revisar esse texto) permitindo que eu me dedicasse integralmente ao desenvolvimento do mesmo.

Aos meus pais agradeço por todo o esforço e sacrifício que fizeram para que eu pudesse chegar aqui, sem eles eu certamente não teria oportunidade de realizar este trabalho nem de viver todas as experiências maravilhosas que tive até hoje.

Ao meu irmão Marcel por tudo que me ensinou e todos os conselhos que me deu até hoje.

Agradeço à Marilene Bittencourt que sempre me incentivou a ser melhor e há muito tempo dá ensinamentos que me levam para um caminho de sucesso.

Ao meu mentor Eduardo Bellani por me guiar nos primeiros momentos profissionais de minha carreira.

Agradeço a Neoway e todos os seus funcionários pelas oportunidades que me foram dadas e por fornecer dados indispensáveis no desenvolvimento deste trabalho.

Por fim agradeço a todos os professores da UFSC que tive contato, em especial a minha orientadora Prof.ª Luciana de Oliveira Rech, Dr.ª e aos membros da banca Prof. Mário Antônio Ribeiro Dantas, Dr. e Prof.ª Jerusa Marchi, Dr.ª.

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 *website*. Tal *website* pode não estar preparado para um volume de consultas tão grande 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. In order to do this, it's 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 may not always come from a human. Companies specializing in Big data have a great interest in data from public sources in order to make analysis 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 website. This website may not be prepared for such a great 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 algoritmos. 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. Também são apresentados alguns 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 treinamento da rede neural para reconhecimento de imagens de CAPTCHA.

No capítulo 5 são apresentados os testes do treinamento da rede neural para reconhecimento de imagens de CAPTCHA. Este capítulo também contém a apresentação dos dados obtidos através das metodologias escolhidas no capítulo 4.

Por fim, no capítulo 6 estão as conclusões obtidas através dos resultados deste trabalho, as vulnerabilidades 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. Este modelo tem como entrada um conjunto de treinamento que possui dados de observações em geral. Estas observações podem estar relacionadas a qualquer evento ou objeto, físico ou virtual. Desse modo as previsões são feitas com orientação 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[2], redes neurais geralmente são especificadas utilizando 3 artefatos:

- **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. Este modelo é usado para prever probabilidades de variáveis associadas a uma determinada classe. As previsões são baseadas em um conjunto de variáveis independentes. Para construir este modelo, esta seção descreve as tarefas e cálculos principais que são utilizadas nesse método.

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 rotulá-la como "classe A". Geralmente há muitos exemplos da entidade que se deseja classificar. Esses exemplos, já mapeados com seu respectivo rótulo, são chamados de conjunto de treinamento. Após o treinamento, o objetivo é descobrir em qual classe um exemplo completamente novo se encaixa.

É dito que esse aprendizado é supervisionado[3], pois cada exemplo recebeu um rótulo durante o treinamento. Já 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.

[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 valores: quantidade de verde, quantidade de vermelho e quantidade de azul.

2.3.2 *CLASSIFICADOR LOGÍSTICO*

Um classificador logístico (geralmente chamado de regressão logística[3]) recebe como entrada uma informação, como por exemplo os pixels de uma imagem, e aplica uma função linear a eles para gerar suas predições. Uma função linear é 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.

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

Na equação 2.1, "X" é como chamaremos o 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[4].

Se os pesos forem inicializados com um valor muito baixo, existe a possibilidade das ativações da rede neural diminuírem 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:

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

No qual 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 as pontuações geradas pelo classificador logístico em probabilidades. É essencial 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 transformar essas pontuações em probabilidades utiliza-se uma função chamada ***Softmax***[3]. Denotada na equação 2.3 por "S".

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

O mais importante dessa fórmula é receber qualquer tipo de pontuação gerada 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. 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 à 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 simples e geralmente chamada de *One-Hot Encoding*. Com isso é possível medir a eficiência do treinamento apenas comparando dois 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*** [3].

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

Na equação 2.4, 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 modelo de regressão.

PERDA

Para cada valor aleatório de peso e *bias*, é possível medir a distância média para todas as entradas do conjunto completo de treinamento e todos rótulos que estão disponíveis. Esse valor é chamado de **perda**[3] do treinamento. Esta perda, que é a média de *cross entropy* de todo treinamento, é uma função grande e custosa.

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

Na equação 2.5, cada exemplo no conjunto de treinamento é multiplicado por uma grande matriz “W”, e em seguida todos os valores são somados.

O objetivo é que as distâncias sejam minimizadas, o que significa que a classificação está funcionando para todos os exemplos dos dados de treinamento. 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 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. É possível 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 alcançado utilizando o método do gradiente (ou *Gradient Descent* em inglês) [3].

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

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

De acordo com as equações 2.6 e 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*.

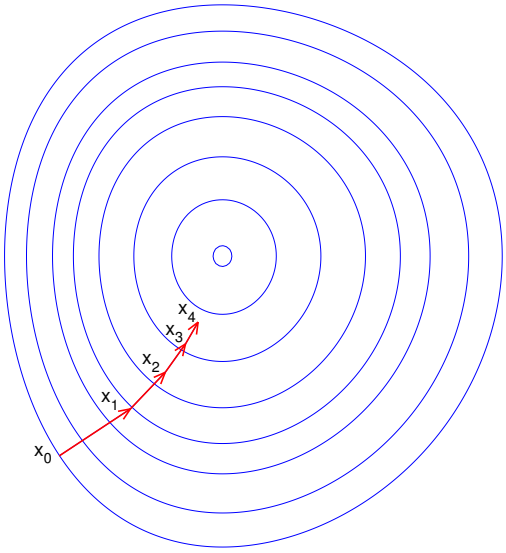

Figura 1: Utilização do método do gradiente mediante a função de perda.

Na figura 1, os círculos da imagem representam a função de perda quando há apenas dois 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[5]. Essa técnica de aprendizado começou a ficar mais famosa depois de dois adventos específicos da computação: a geração de enormes volumes de dados e a utilização de *Graphic Processing Units* (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 por 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 a partir de conceitos mais simples. Ao desenhar 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*)[3] é uma peça chave de *Deep learning*. Praticamente todo o aprendizado em profundidade é alimentado por esse algoritmo. O problema do método do Gradiente visto anteriormente está na dificuldade de escalabilidade do mesmo. 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. 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* [3] 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 será aplicado um declínio exponencial à taxa de aprendizado[6].

2.4.4 *RELU*

Modelos lineares são simples e estáveis numericamente, mas podem se tornar ineficientes ao longo do tempo. Portanto, para adicionar mais camadas ao modelo será necessário introduzir alguns cálculos não lineares entre camadas. Em arquiteturas de profundidade, as funções de ativação dos neurônios se chamam *Rectified Linear Units* (ReLUs)[3], e são capazes de introduzir os cálculos necessários 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.

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

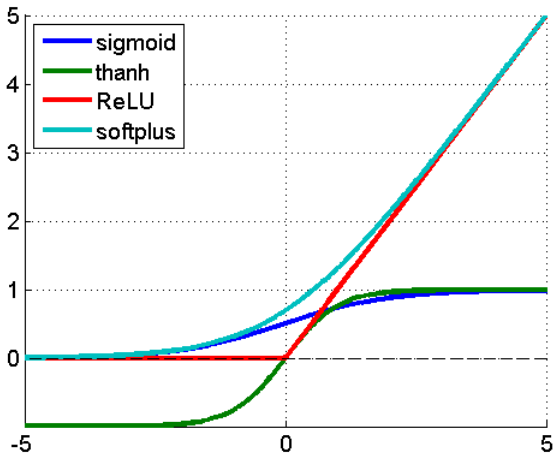

Figura 2: Comparação de funções de ativação.

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 forem necessárias para uma melhor performance.

2.4.5 *BACKPROPAGATION*

Backpropagation[3] 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 é possível 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* no modelo, pois é adicionado 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 gera uma penalidade em pesos maiores. Essa regularização é atingida adicionando a norma L_2 [3] 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*[7]. Supondo que temos uma camada conectada à outra em uma 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, é atribuído o valor 0 para metade desses valores. Basicamente metade dos dados que estão fluindo pela rede neural é destruída aleatoriamente.

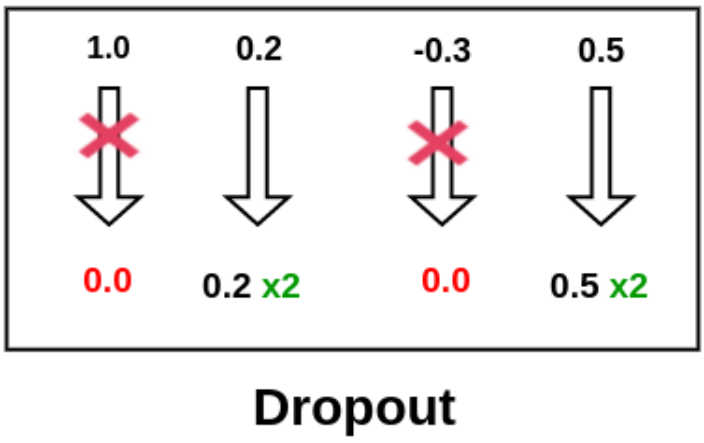

Figura 3: Exemplo da tarefa realizada por uma camada de *dropout*.

A figura 3 tenta dar um exemplo em que metade dos pesos são excluídos. A técnica de

dropout faz com que a 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. 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

Redes neurais convolucionais de profundidade (CNNs) são a primeira abordagem verdadeiramente bem sucedida de 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*[8].

A grande vantagem na abordagem de CNNs para reconhecimento é que não é necessário um extrator de características desenvolvido por um ser humano. Nas soluções de Krizhevsky[7] e Goodfellow[9] 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[10]:

> "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 se encaixam perfeitamente. Ao combinar o aprendizado em profundidade com redes convolucionais 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 pro-

fundidade que é representada pelos canais de cor (vermelho, verde e azul). Uma convolução consiste em coletar 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 é um conjunto de **mapas de características** da imagem original. Como exemplo, transforma-se 3 mapas de carcterísticas (canais de cores) para uma quantidade K de mapas de características.

Ao invés de apenas vermelho, verde e azul, agora foi gerada 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 só é levado em consideração este pequeno fragmento, há bem menos pesos e eles são todos compartilhados pelo espaço da imagem.

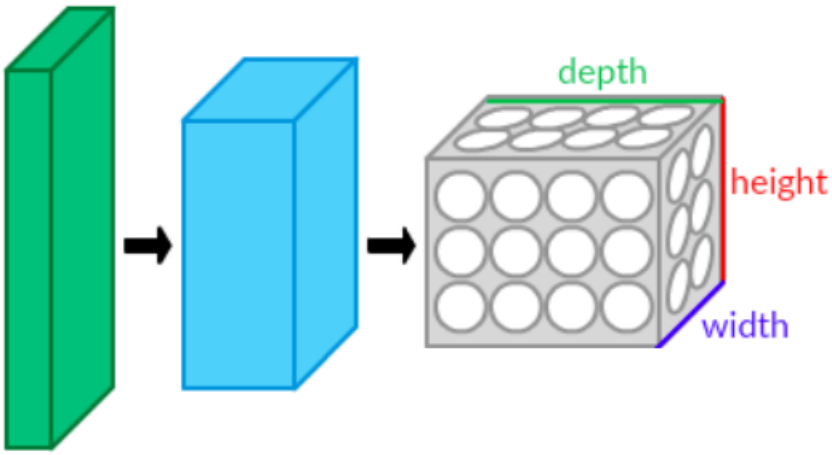

Figura 4: Exemplo de camadas convolucionais.

Para cada convolução da figura 4, é criada uma nova imagem que possui uma nova largura (*width* em inglês), altura (*height* em inglês) e profundidade (*depth* em inglês).

Uma rede convolucional[1] será basicamente uma rede neural de profundidade. Ao invés de empilhar camadas de multiplicação de matrizes, empilha-se convoluções. No começo haverá uma imagem grande que possui apenas os valores de pixel como informação. Em seguida são aplicadas convoluções que irão "espremer" as dimensões espaciais e aumentar a profundidade. No final é possivel conectar o classificador e ainda lidar apenas com parâmetros que mapeiam o conteúdo da imagem.

STRIDE

Quando é realizada uma convolução desliza-se uma janela com o tamanho do *Kernel* pela imagem. Esta janela possui um parâmetro chamado *stride*[1], que indica quantos pixels de espaçamento haverá entre um fragmento da imagem e outro. Por exemplo, um *stride* igual a "um" significa que a imagem de saída pode ter a mesma largura e altura que a imagem de entrada. Um valor igual a "dois" significa que a imagem de saída pode ter metade do tamanho.

PADDING

O parâmetro de *padding*[1] define 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***.

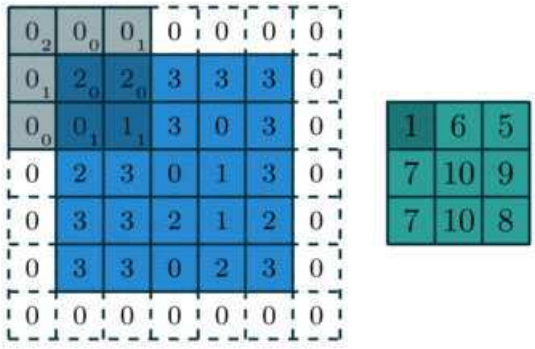

Figura 5: Aplicação de uma convolução sobre uma imagem.[1]

No exemplo da figura 5 há 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 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 é possível perder bastante informação no processo. Ao invés disso, são realizadas convoluções com *stride* igual a 1, sem perder nenhuma informação da imagem original.

MAX POOLING

Após a camada convolucional adiciona-se uma camada de *pooling* que irá receber todas as convoluções e combiná-las da seguinte forma[1]. Para cada ponto nos mapas de características a execução desta camada 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 a 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*.

Devido a quantidade de componentes presentes na estrutura de redes neurais é aparente a complexidade quanto ao entendimento do funcionamento geral. A figura 6 tenta explicar como esses componentes se conectam e em qual sequência. Os valores são completamente fictícios e não condizem com um cálculo real.

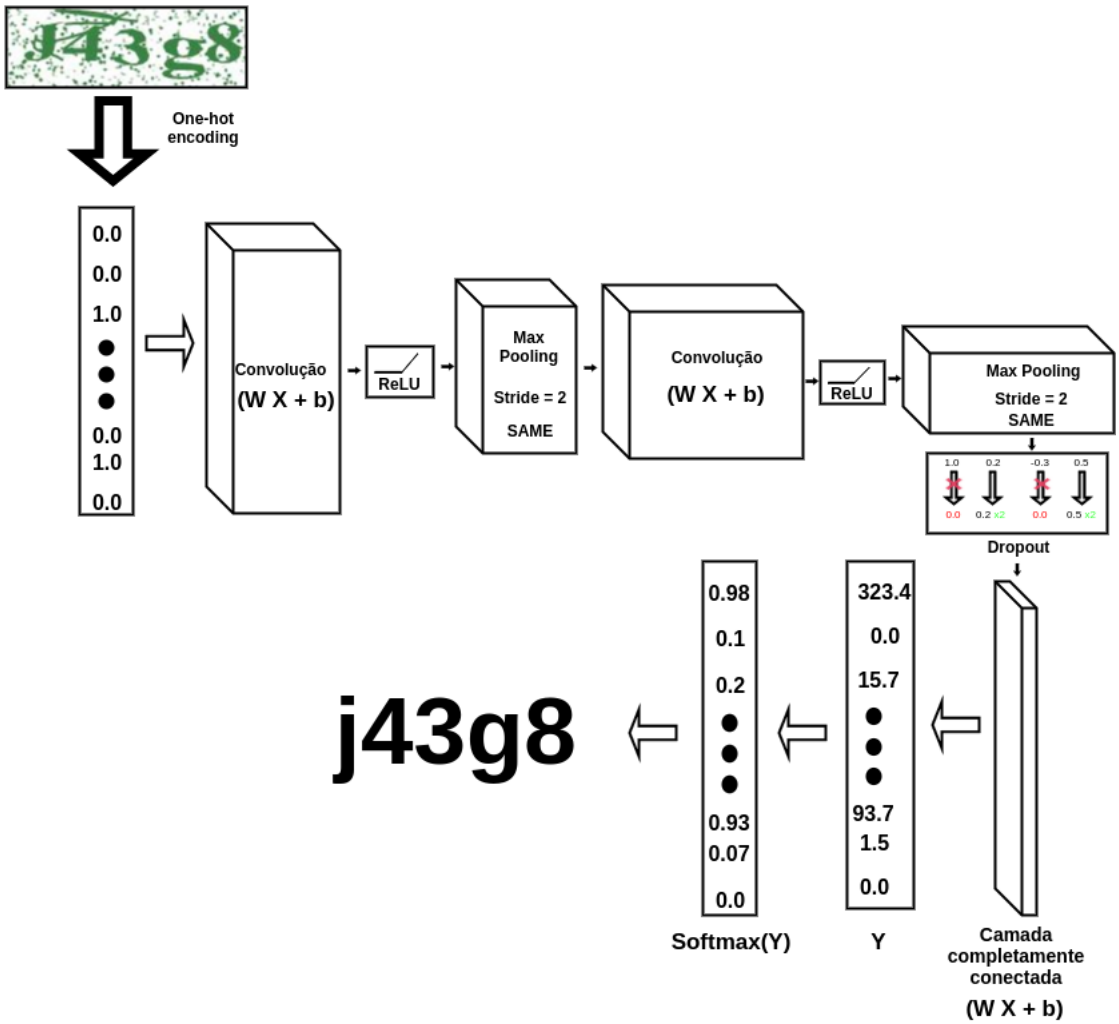

Figura 6: Exemplo da composição de todos os componentes presentes em redes neurais convolucionais de profundidade.

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 é possível realizar a configuração e treinamento da rede neural. E por fim será calculada a acurácia, mediante imagens de teste, do modelo que teve a melhor performance no treinamento.

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 206.564 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 7: 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 Serviços. 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, que significa **180 mil** imagens para o contexto do trabalho.

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*. Também será desenvolvido 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 virtual da AWS[12]. 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.

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 essa 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. 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) e o treinamento da rede neural é feito em uma *Session*.

O projeto é composto por 5 tarefas de implementação:

- Desenvolvimento do leitor e processador do conjunto de dados.
- Desenvolvimento da função que monta a rede neural.
- Configuração da rede neural para otimização dos resultados.
- Desenvolvimento da etapa de treinamento da rede neural.
- Desenvolvimento da etapa de teste e acurácia do modelo da rede neural.

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 de 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” a “z” (26 letras) ou um dígito de “0” a “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 é preciso 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 é preciso fazer é gerar um vetor utilizável desse texto.

Primeiro é transformado cada caractere em um número de 0 a 35. Isso é feito recuperando o código ASCII de cada caractere e normalizando a sequência. Portanto, para os dígitos (0 a 9) que possuem códigos indo de 48 a 57, é subtraído 48. E para as letras (a a z) que possuem códigos indo de 97 a 122, é subtraído 87 e assim resultando em números de 10 a 35.

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

FRAÇÃO 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 necessidade de recuperar uma amostra aleatória dos dados em cada iteração.

Portanto há uma variável global na classe *OCR_data* que mantém o estado da posição que se encontra o conjunto de dados. Após passar por todo o conjunto de dados, o método começa a fazer 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 será visto a seguir.

QUANTIDADE DE CANAIS DE COR

No contexto do trabalho, a cor de um caractere da imagem não importa. Uma letra “A” pode ser vermelha, azul ou verde mas ainda terá que ser reconhecido como letra “A”. Com isso em mente é possível reduzir a quantidade de informações que o modelo precisa aprender. É reduzida também a complexidade dos cálculos feitos pelo modelo. Quando é feita a leitura da imagem com a biblioteca OpenCV, indica-se 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 é possível normalizar os 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* isso é feito invocando a função *resize*. Nessa função é passado como parâmetro a largura e altura alvos, assim como o algoritmo que deve ser usado para a interpolação[1]. Foi escolhido o 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 será visto que esses valores se encaixarão mais naturalmente no modelo. Como algoritmo de interpolação foi escolhido a reamostragem utilzando a relação da área de pixel (opção *INTER_AREA* do *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 são criados 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 a 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

O desenvolvimento da arquitetura da rede neural foi realizado criando a função *ocr_net*, que é responsável por especificar o grafo da rede neural. Essa função irá receber as imagens de entrada, a quantidade de pesos em cada camada e a quantidade de *biases* para cada camada.

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.

[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.

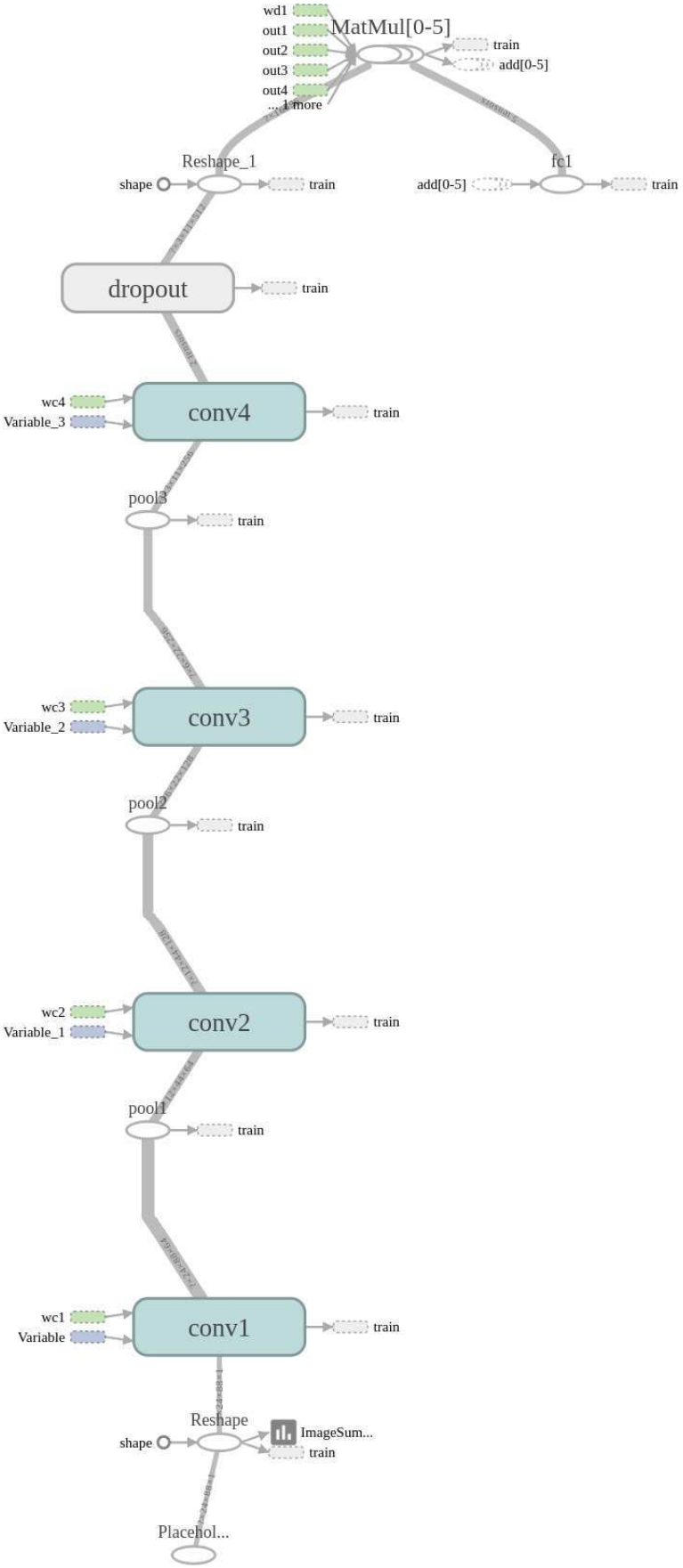

Figura 8: Arquitetura geral do modelo de rede neural treinado.

4.3.1 *ENTRADAS*

O grafo da arquitetura começa com dois parâmetros de entrada, as imagens de entrada e os rótulos correspondentes. Para esses parâmetros são criados *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 na 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. 1ª Camada convolucional: 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. 1ª Camada oculta: utiliza **ReLU** como função de ativação e não recebe nenhum parâmetro. Tem como entrada a camada convolucional anterior.

4. 1ª Camada de *pooling*: 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. 2ª Camada convolucional: 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. 2ª Camada oculta: utiliza **ReLU** como função de ativação e não recebe nenhum parâmetro. Tem como entrada a camada convolucional anterior.

7. 2ª Camada de *pooling*: 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. 3ª Camada convolucional: 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. 3ª Camada oculta: utiliza **ReLU** como função de ativação e não recebe nenhum parâmetro. Tem como entrada a camada convolucional anterior.

10. 3ª Camada de *pooling*: 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. 4ª Camada convolucional: 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 como parâmetro um valor de probabilidade de manter cada peso da rede neural.

13. Camada completamente conectada: tem como entrada todas as ativações das camadas anteriores. Para habilitar a camada completamente conectada é preciso 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 a 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 em artigos citados ao longo da sessão, ou vieram dos exemplos e tutoriais citados na seção 4.1. Alguns parâmetros foram modificados ao longo dos testes.

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 em estudos anteriores[9], 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 é recebida 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 exemplos e tutoriais mencionados[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* do 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.

4.5 FASE DE TREINAMENTO

A fase de treinamento é o momento onde se juntam todas as peças da arquitetura e configuração da rede neural. Na implementação foi criada uma função *train* que é responsável por montar de fato o grafo para a rede e utilizar o objeto *Session* do *TensorFlow* para o treinamento.

4.5.1 *CRIAÇÃO DA SESSÃO DE TREINAMENTO*

O desenvolvimento começa com a abertura da sessão de treinamento. Esta sessão será destruída ao final das iterações para limpar todas as variáveis de treinamento carregadas em memória. Após a abertura da sessão é chamada a função *ocr_net* para que seja instanciada a arquitetura da rede neural. Todas as variáveis instanciadas até o momento de execução da sessão são apenas espaços reservados, assim como os *placeholders* criados para os dados de entrada do modelo. Com a arquitetura em mãos será calculada a perda, realizando uma soma das perdas para cada caractere de saída da rede neural. Em seguida é instanciado o otimizador (*MomentumOptimizer*) e por final as predições são agregadas em uma matriz transposta para que fiquem com o formato 5x36 (5 caractéres por 36 classes).

4.5.2 *INICIALIZAÇÃO DA SESSÃO DE TREINAMENTO*

Para inicializar a sessão de treinamento é invocado o método *initialize_all_variables* para popular os espaços reservados das variáveis. Com isso é possível executar a sessão pela primeira vez invocando *session.run* passando como parâmetro o retorno de *initialize_all_variables*.

4.5.3 *EXECUÇÃO DA SESSÃO DE TREINAMENTO*

Para manter o controle do treinamento é criada uma variável *step* que irá manter o estado da quantidade de passos executados em cada execução da sessão. Para cada passo é invocado o método *next_batch* da instância da classe *OCR_data* referente ao treinamento. Também será invocado *session.run* passando como parâmetro o otimizador e carga de dados através de um dicionário chamado *feed_dict*. Esse dicionário receberá o retorno do método *next_batch* e com isso é possível popular os *placeholders* da carga de imagens e de seus respectivos rótulos.

Em cada passo executado é calculada a perda para o treinamento. E a cada 100 passos é calculada a acurácia para a sessão de treinamento. A execução do treinamento fica em ciclos até que o número de passos multiplicado pela quantidade de carga (64) seja igual ao número de iterações (200 mil ou 500 mil). Ao final de todas as iterações é instanciado um *Saver*, o qual fará a escrita das variáveis de nosso modelo em um arquivo. Este arquivo será utilizado para a avaliação da acurácia em situações reais.

4.6 FASE DE TESTE

A implementação da fase de teste consiste na execução de apenas um passo da sessão aberta na seção 4.5. Ao final da otimização do modelo, é carregado o conjunto de imagens para teste, assim como os respectivos rótulos. A carga é feita utilizando o método *next_batch* da instância da classe *OCR_data* referente ao teste. Com isso é possível popular o dicionário *feed_dict* da sessão de teste e executar *session.run* com este parâmetro. Ao final teremos o valor da acurácia e da perda para o conjunto de dados do teste.

4.6.1 *ACURÁCIA*

O cálculo da acurácia para o modelo é o mesmo tanto para a fase de treinamento quanto para fase de teste. A função de acurácia recebe como parâmetro a matriz de predições gerada pelo modelo e a matriz de rótulos fornecida pela carga dos dados. O cálculo é feito armazenando o maior valor de cada *array* da matriz de predições e comparando as posições desses valores com as posiçõs da matriz de rótulos. A acurácia sobe para cada letra certa em uma imagem de CAPTCHA, sem levar em consideração se todas as letras estão certas ou não.

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 3. Os resultados foram satisfatórios para o contexto do trabalho. Para produzir os gráficos foi utilizada uma ferramenta chamada ***TensorBoard***, que vem junto com a instalação do *TensorFlow*.

5.1 TREINAMENTO COM 200 MIL ITERAÇÕES

Inicialmente é realizado 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 alcançado um valor máximo de acurácia igual a **84,38%** no conjunto de treinamento, **79,6%** no conjunto de teste e um valor mínimo de perda igual a **2,50** para o conjunto de treinamento, **2,91** para o conjunto de teste.

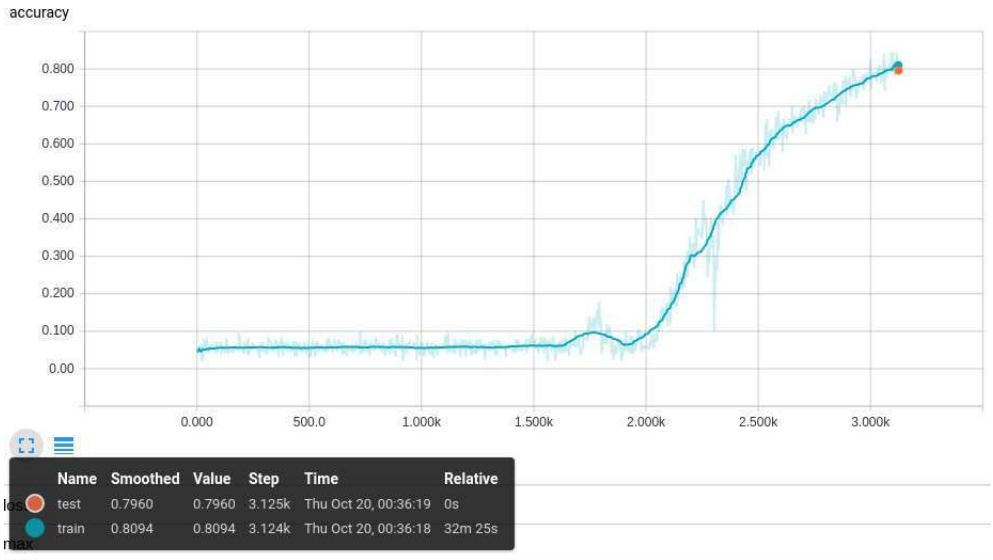

Figura 9: Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 200 mil iterações.

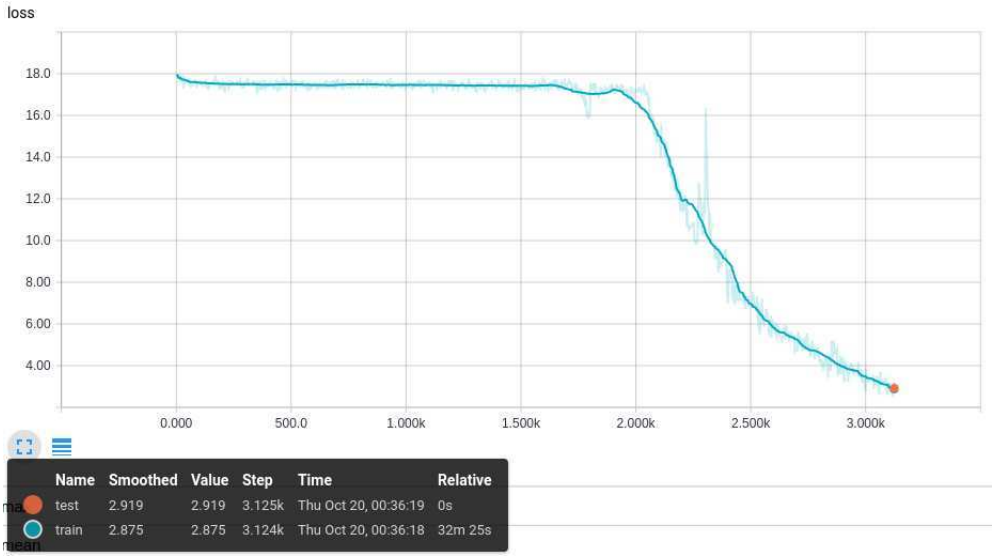

Figura 10: Gráfico da perda em relação ao número de passos para o treinamento da rede com 200 mil iterações.

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.

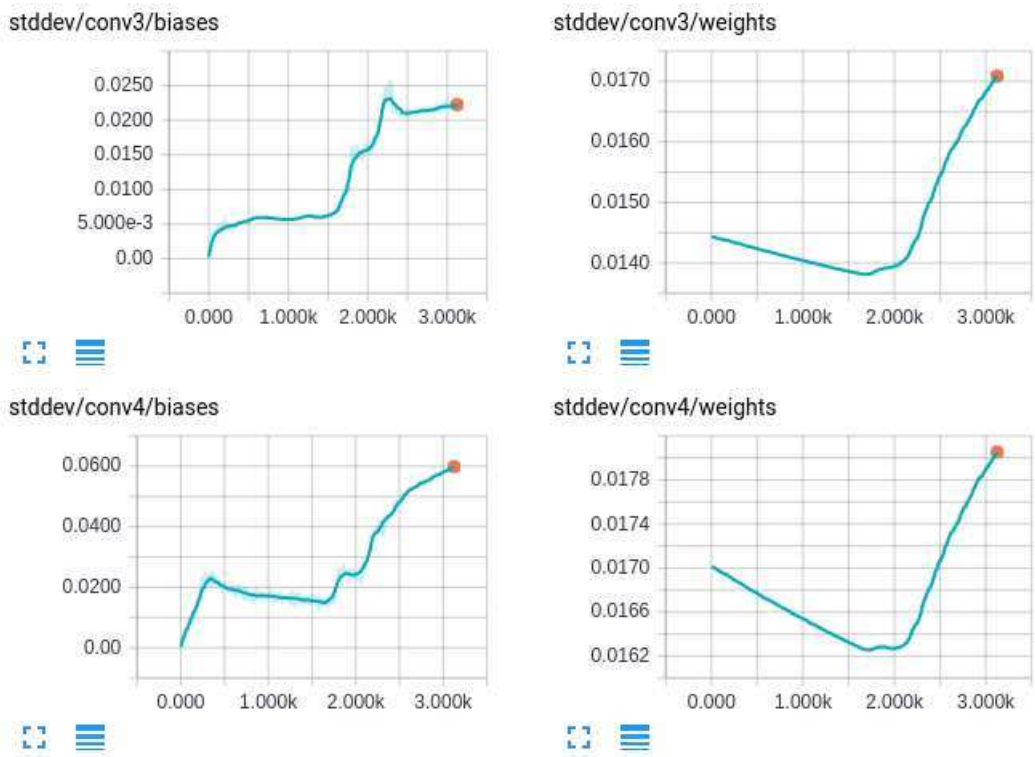

Figura 11: Desvio padrão dos pesos e *biases* em relação ao número de passos para o treinamento da rede com 200 mil iterações.

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 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 alcançado o valor máximo de acurácia igual a **98,75%** no conjunto de treinamento, **81,37%** no conjunto de teste e um valor mínimo de perda igual a **0,36** para o conjunto de treinamento, **13,52** para o conjunto de teste.

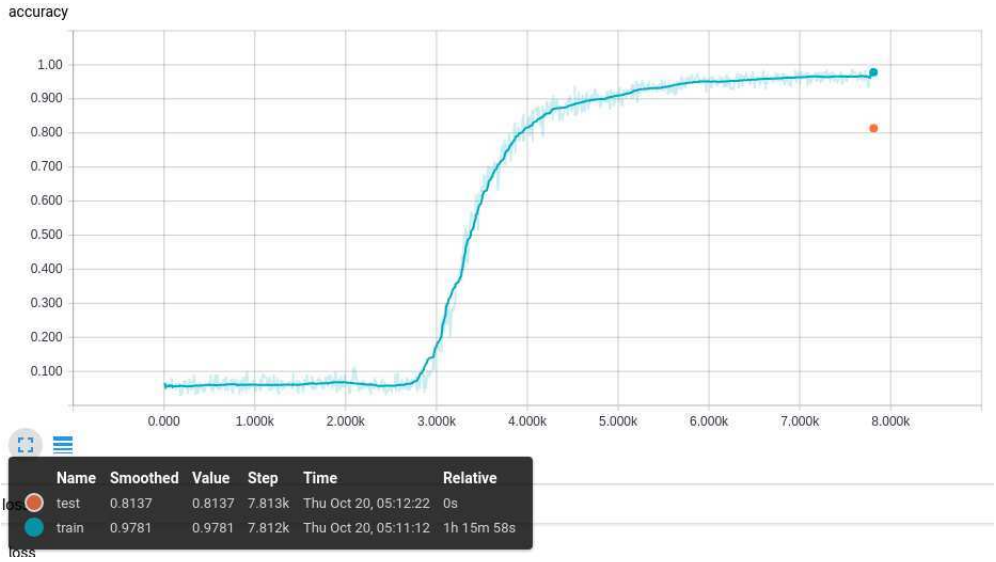

Figura 12: Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 500 mil iterações.

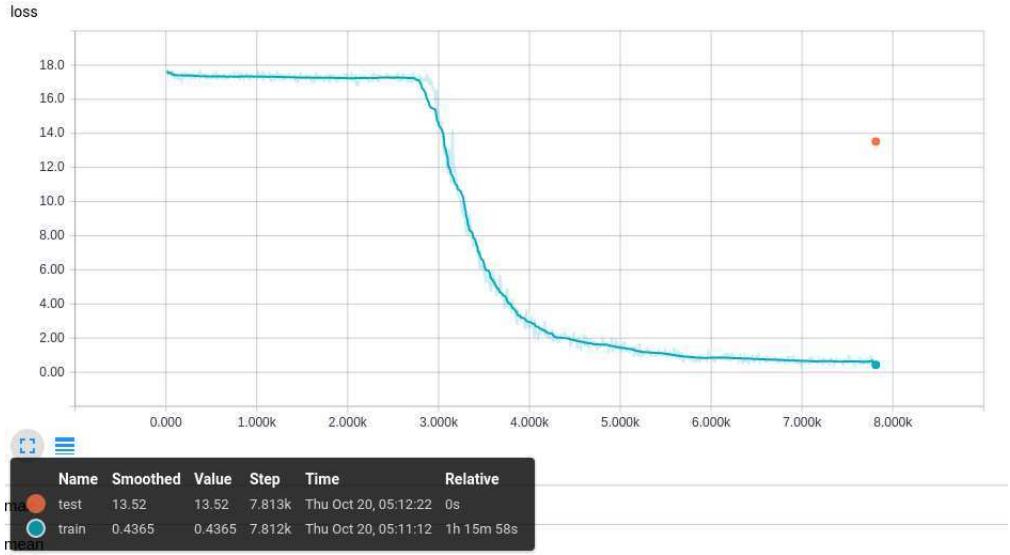

Figura 13: Gráfico da perda em relação ao número de passos para o treinamento da rede com 500 mil iterações.

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**.

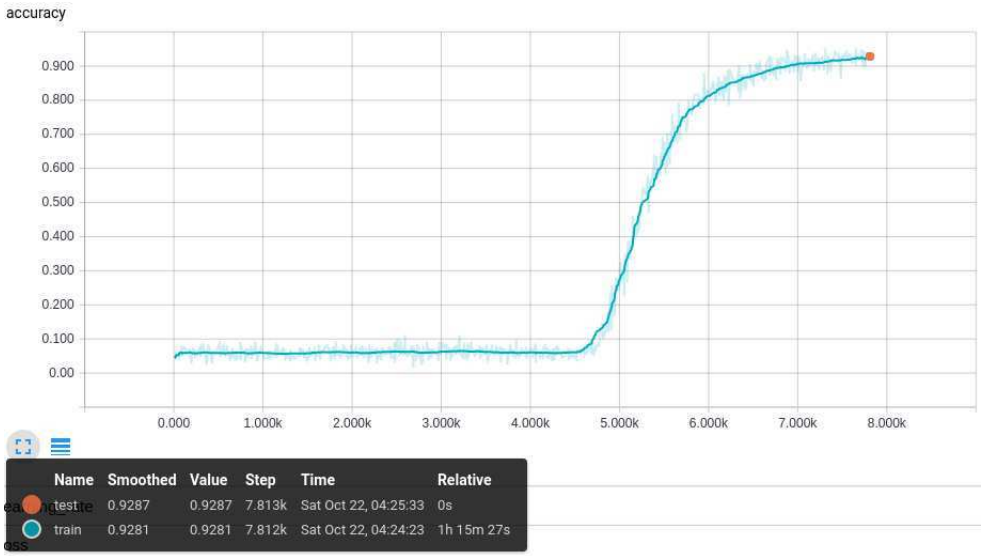

Figura 14: Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 500 mil iterações e probabilidade de *dropout* igual a 50%.

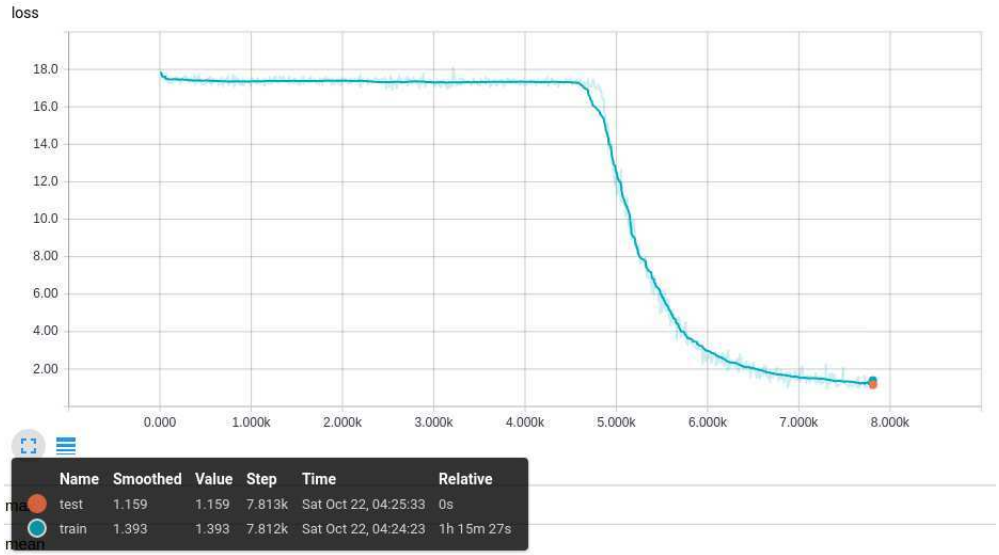

Figura 15: Gráfico da perda em relação ao número de passos para o treinamento da rede com 500 mil iterações e probabilidade de *dropout* igual a 50%.

Ao final do treinamento foi alcançado o valor máximo de acurácia igual a **95,31%** no conjunto de treinamento, **92,87%** no conjunto de teste e um valor mínimo de perda igual a **0,93** para o conjunto de treinamento, **1,15** para o conjunto de teste.

5.4 AVALIAÇÃO DA ACURÁCIA EM CASOS NOVOS

Como dito anteriormente na seção 4.6, o cálculo feito para determinar a acurácia nos casos de teste não contemplam um acerto completo de todas as letras em uma imagem de CAPTCHA. Por conta desse fator e também para testar o modelo em imagens novas, foi desenvolvido um script para carregar o modelo treinado. Esse script executa uma sessão do modelo da mesma forma em que foi feito o treinamento na seção 4.5. A execução do teste foi mediante **18.000** imagens novas para obter um resultado fiel a uma situação real de reconhecimento de CAPTCHAs.

O cálculo final é simples, para cada imagem em que o modelo decifra corretamente o texto, é incrementada uma variável **acertos**. Ao final da execução do teste para cada imagem, é dividido o número de acertos pelo total de imagens fornecidas. A taxa de acurácia alcançada ao executar a avaliação foi de **79,65%**, ou seja **14.337** imagens reconhecidas completa e corretamente.

5.5 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)	84,38%	98,75%	95,31%
Perda (teste)	2,91	13,52	1,15
Perda (treinamento)	2,50	0,36	0,93

Tabela 1: Desempenho geral do sistema.

	Avaliação do modelo de 500k it. e 50% *dropout*
Acurácia	**79,65%**
Imagens corretamente classificadas	14.337
Total de imagens	18.000

Tabela 2: Avaliação em casos novos do modelo treinado com 500 mil iterações e 50% de *dropout*.

Com a execução do script de acurácia citado na seção 5.4, foram identificados diversos casos complexos em que o modelo teve êxito no reconhecimento do texto completo.

Figura 16: Exemplos de casos complexos.

Para os casos complexos demonstrados na figura 16, o reconhecedor acertou o texto completo da imagem. Na primeira imagem é possível observar a semelhança entre uma letra “g” e o dígito “8”. Diante da proximidade entre as letras da segunda imagem, o modelo poderia ter classificado a penúltima letra como um “q” ao invés do “c” que foi a letra correta. Novamente é perceptível a proximidade entre as letras na terceira imagem. Abaixo das imagens se encontram os textos reconhecidos pelo modelo em cada caso respectivamente.

Contudo também foram observados casos em que o modelo não teve sucesso no reconhecimento do texto completo. Devido a semelhança de alguns caractéres, o modelo reconheceu parcialmente o texto de algumas imagens apresentadas.

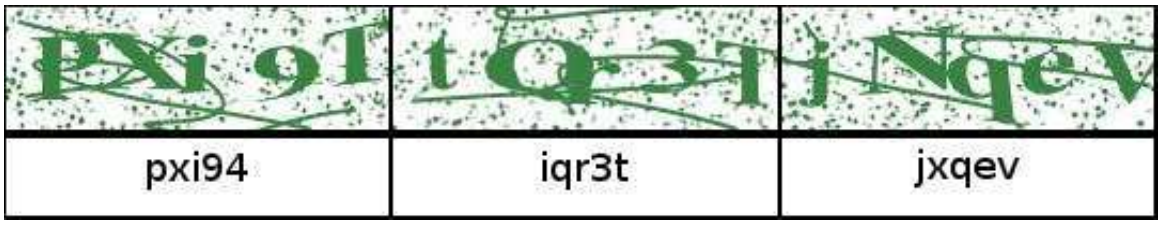

Figura 17: Exemplos de casos complexos em que o reconhecedor acertou parcialmente o texto da imagem.

Na primeira imagem da figura 17 é possível observar que o modelo confundiu a letra “T” maiúscula com um dígito “4”. Na segunda imagem, mesmo com a sobreposição dos caractéres “Q” e “r” no meio da imagem, esses caractéres foram reconhecidos corretamente. Enquanto o primeiro carctere “t” no início da imagem foi erroneamente reconhecido como uma letra “i”. Na terceira imagem a letra “N” maiúscula foi reconhecida erroneamente como uma letra “x”. Abaixo das imagens se encontram os textos reconhecidos pelo modelo em cada caso respectivamente.

6 CONCLUSÕES

Para desenvolver o projeto foi escolhido o software que até a presente data é o mais recomendado para tarefas de aprendizado de máquina. Um reconhecedor de texto em imagens de CAPTCHA foi desenvolvido ao longo do trabalho. Este reconhecedor conta com um alto nível de robustez diante dos testes realizados.

Os testes realizados em todos os casos mostraram ser possível atingir um resultado razoável na tarefa de reconhecimento de textos em imagens, isso com poucos ajustes à configuração de treinamento de redes neurais. Atualmente a quantidade de exemplos e tutoriais disponíveis para tarefas de aprendizado de máquina é imenso. Fica claro que é possível implementar classificadores mesmo com poucos recursos.

6.1 VULNERABILIDADE DE FONTES PÚBLICAS

Diante do objetivo alcançado pelo trabalho, fica aparente que fontes públicas de dados podem estar vulneráveis. Consultas automatizadas realizadas por *Web crawlers* podem não afetar diretamente a segurança das informações, isso porque as informações já estão sendo disponibilizadas publicamente. Entretanto a disponibilidade de tais fontes pode ser afetada quando o ambiente de um *website* não está preparado para um volume muito grande de consultas.

6.1.1 *EFICÁCIA DE CAPTCHAS*

Enquanto é possível discutir a eficácia dessas imagens de CAPTCHA e como gerar imagens mais difíceis de se reconhecer, também cabe uma discussão sobre a necessidade de imagens na tentativa de bloquear consultas automatizadas. Imagens usadas como CAPTCHA geralmente são frustrantes para usuários humanos de sistemas de consulta. Ao tentar dificultar o reconhecimento de imagens por máquinas, também se dificulta o acesso de um usuário comum às informações. Portanto é possível abrir espaço para estudos que buscam outras formas de bloqueio de *Web crawlers*.

Outra alternativa, inclusive mais interessante, seria disponibilizar outros tipos de consulta específicos para sistemas de terceiros que desejam utilizar dados públicos. Assim um *website* de fonte pública de dados poderia continuar a servir usuários humanos com robustez e ao mesmo tempo servir usuários sistêmicos com um formato mais adequado.

6.2 TRABALHOS FUTUROS

Como possíveis trabalhos futuros, cita-se:

- Fazer um melhor uso das informações geradas pelo processo de treinamento para gerar heurísticas mais inteligentes. Um exemplo seria utilizar outros tipos de otimizadores para a função da perda.
- Estender o sistema para realizar o reconhecimento de outros tipos de CAPTCHAs.
- Estender o sistema para realizar o reconhecimento de tipos de CAPTCHAs que possuem um tamanho de texto variável.
- Realizar um estudo sobre *Web crawlers* em fontes públicas que utilizam CAPTCHA, executando o sistema proposto neste trabalho.
- Implementar um sistema de reconhecimento de CAPTCHAs mais avançados que solicitam a classificação de uma cena completa ou identificação de objetos em imagens.
- Estudar um artifício mais efetivo para o bloqueio de consultas automatizadas em *websites*.

Considera-se de extrema importância a implementação de projetos desse tipo pois o mesmo auxilia na compreensão e aplicação de Inteligencia Artificial em casos específicos.

7 *ANEXO A – CÓDIGO FONTE DESENVOLVIDO AO LONGO DO PROJETO.*

Este anexo contém o código fonte de todo o software desenvolvido como experimento ao longo do projeto. para o conjunto de dados

7.1 IMPLEMENTAÇÃO DO LEITOR E PROCESSADOR DAS IMAGENS, ARQUIVO *CAPTCHA_DATA.PY*.

```
1
2  import glob, os
3  import numpy as np
4  import cv2
5  import random
6
7  class OCR_data(object):
8      def __init__(self, num, data_dir, num_classes, batch_size=50, len_code
           =5, height=60, width=180, resize_height=24, resize_width=88,
           num_channels=1):
9          self.num = num
10         self.data_dir = data_dir
11         self.num_classes = num_classes
12         self.batch_size = batch_size
13         self.len_code = len_code
14         self.height = height
15         self.width = width
16         self.resize_height = resize_height
17         self.resize_width = resize_width
18         self.num_channels = num_channels
19         self.index_in_epoch = 0
20         self._imgs = []
21         self._labels = []
22         for pathAndFilename in glob.iglob(os.path.join(data_dir, '*.png')):
23             img, label = self.create_captcha(pathAndFilename)
```

```
24              self._imgs.append(img)
25              self._labels.append(label)
26          self._imgs = np.array(self._imgs).reshape((-1, resize_height,
                resize_width, num_channels)).astype(np.float32)
27          self._labels = np.array(self._labels)
28
29      def create_captcha(self, pathAndFilename):
30          img = cv2.imread(pathAndFilename, cv2.IMREAD_GRAYSCALE)
31          img = cv2.resize(img,(self.resize_width, self.resize_height),
                interpolation=cv2.INTER_AREA)
32          filename, ext = os.path.splitext(os.path.basename(pathAndFilename))
33          label = self.create_label(filename)
34          return (img, label)
35
36      def create_label(self, filename):
37          label = []
38          for c in filename:
39              ascii_code = ord(c)
40              if ascii_code < 58:
41                  char_value = ascii_code - 48
42              else:
43                  char_value = ascii_code - 87
44              label.append(char_value)
45          return self.dense_to_one_hot(label, self.num_classes)
46
47      def dense_to_one_hot(self, labels_dense, num_classes):
48          num_labels = len(labels_dense)
49          index_offest = np.arange(num_labels) * num_classes
50          labels_one_hot = np.zeros((num_labels, num_classes))
51          labels_one_hot.flat[index_offest + labels_dense] = 1
52          labels_one_hot = labels_one_hot.reshape(num_labels*num_classes)
53          return labels_one_hot
54
55      def next_batch(self, batch_size):
56          start = self.index_in_epoch
57          self.index_in_epoch += batch_size
58          if self.index_in_epoch > self.num:
59              perm = np.arange(self.num)
60              np.random.shuffle(perm)
61              self._imgs = self._imgs[perm]
62              self._labels = self._labels[perm]
63              start = 0
64              self.index_in_epoch = batch_size
```

```
65              assert batch_size <= self.num
66          end = self.index_in_epoch
67          return self._imgs[start:end], self._labels[start:end]
```

7.2 IMPLEMENTAÇÃO DA FUNÇÃO QUE MONTA, CONFIGURA, TREINA E TESTA A REDE NEURAL DESENVOLVIDA, ARQUIVO *CAPTCHA_TRAIN.PY*.

```
1
2  import tensorflow as tf
3  import time
4  import numpy as np
5  from captcha_data import OCR_data
6  # Parameters
7  initial_learning_rate = 0.01
8  decay_steps = 1000
9  decay_rate = 0.9
10 momentum = 0.9
11 l2_beta_param = 3e-4
12 dropout_keep_prob = 0.5
13 training_iters = 500000
14 batch_size = 64
15 display_step = 100
16 summaries_dir = 'logs'
17
18 # Network Parameters
19 resize_width = 88
20 resize_height = 24
21 color_channels = 1
22 n_chars = 5
23 n_classes = 36  # 10+26
24 n_training_samples = 180000
25 n_test_samples = 8000
26 fc_num_outputs = 4096
27
28 # Calculate Elapsed time
29 start_time = time.time()
30
31 data_train = OCR_data(n_training_samples, './images/train', n_classes)
32 data_test = OCR_data(n_test_samples, './images/test', n_classes)
33
```

```
34 # tf Graph input
35 x = tf.placeholder(tf.float32, [None, resize_height, resize_width,
       color_channels])
36 y = tf.placeholder(tf.float32, [None, n_chars*n_classes])
37
38 def print_activations(t):
39         print(t.op.name, t.get_shape().as_list())
40
41 def weight_variable(name,shape):
42         return tf.get_variable(name, shape, initializer=tf.contrib.layers.
               xavier_initializer())
43
44 def bias_variable(shape):
45         initial = tf.constant(0.0, shape=shape)
46         return tf.Variable(initial, trainable=True)
47
48 def max_pool(x, k, name):
49         return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
               padding='SAME', name=name)
50
51 def variable_summaries(var, name):
52         """Attach a lot of summaries to a Tensor."""
53         with tf.name_scope('summaries'):
54                 mean = tf.reduce_mean(var)
55                 tf.scalar_summary('mean/' + name, mean)
56                 with tf.name_scope('stddev'):
57                         stddev = tf.sqrt(tf.reduce_mean(tf.square(var -
                           mean)))
58                 tf.scalar_summary('stddev/' + name, stddev)
59                 tf.scalar_summary('max/' + name, tf.reduce_max(var))
60                 tf.scalar_summary('min/' + name, tf.reduce_min(var))
61                 tf.histogram_summary(name, var)
62
63 def conv2d(x, W, B, name):
64         with tf.name_scope(name) as scope:
65                 with tf.name_scope('weights'):
66                         variable_summaries(W, name + '/weights')
67                 with tf.name_scope('biases'):
68                         variable_summaries(B, name + '/biases')
69                 conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='
                       SAME')
70                 bias = tf.nn.bias_add(conv, B)
71                 with tf.name_scope('Wx_plus_b'):
```

```
72                          tf.histogram_summary(name + '/pre_activations',
                                bias)
73                  conv = tf.nn.relu(bias, name=scope)
74                  tf.histogram_summary(name + '/activations', conv)
75                  return conv
76
77
78  def ocr_net(_x, _weights, _biases, keep_prob):
79          _x = tf.reshape(_x, shape=[-1, resize_height, resize_width,
                color_channels])
80          tf.image_summary('input', _x, 10)
81
82          conv1 = conv2d(_x, _weights['wc1'], _biases['bc1'], 'conv1')
83          print_activations(conv1)
84          lrn1 = tf.nn.local_response_normalization(conv1)
85          pool1 = max_pool(lrn1, k=2, name='pool1')
86          # pool1 = max_pool(conv1, k=2, name='pool1')
87          print_activations(pool1)
88
89          conv2 = conv2d(pool1, _weights['wc2'], _biases['bc2'], 'conv2')
90          print_activations(conv2)
91          lrn2 = tf.nn.local_response_normalization(conv2)
92          pool2 = max_pool(lrn2, k=2, name='pool2')
93          # pool2 = max_pool(conv2, k=2, name='pool2')
94          print_activations(pool2)
95
96          conv3 = conv2d(pool2, _weights['wc3'], _biases['bc3'], 'conv3')
97          print_activations(conv3)
98          lrn3 = tf.nn.local_response_normalization(conv3)
99          pool3 = max_pool(lrn3, k=2, name='pool3')
100         # pool3 = max_pool(conv3, k=2, name='pool3')
101         print_activations(pool3)
102
103         conv4 = conv2d(pool3, _weights['wc4'], _biases['bc4'], 'conv4')
104         print_activations(conv4)
105
106         dropout = tf.nn.dropout(conv4, keep_prob)
107
108         shape = dropout.get_shape().as_list()
109         reshaped = tf.reshape(dropout, [-1, _weights['wd1'].get_shape().
                as_list()[0]])
110
```

```
111         fc1 = tf.nn.relu(tf.matmul(reshaped, _weights['wd1']) + _biases['
                bd1'], name='fc1')
112         print_activations(fc1)
113 
114         fc21 = tf.nn.relu(tf.matmul(fc1, _weights['out1']) + _biases['out1'
                ], name='fc21')
115         print_activations(fc21)
116 
117         fc22 = tf.nn.relu(tf.matmul(fc1, _weights['out2']) + _biases['out2'
                ], name='fc22')
118         print_activations(fc22)
119 
120         fc23 = tf.nn.relu(tf.matmul(fc1, _weights['out3']) + _biases['out3'
                ], name='fc23')
121         print_activations(fc23)
122 
123         fc24 = tf.nn.relu(tf.matmul(fc1, _weights['out4']) + _biases['out4'
                ], name='fc24')
124         print_activations(fc24)
125 
126         fc25 = tf.nn.relu(tf.matmul(fc1, _weights['out5']) + _biases['out5'
                ], name='fc25')
127         print_activations(fc25)
128 
129         return [fc21, fc22, fc23, fc24, fc25]
130 
131 def accuracy_func(predictions, labels):
132         with tf.name_scope('accuracy'):
133                 y = tf.reshape(labels, shape=[-1, n_chars, n_classes])
134                 with tf.name_scope('prediction'):
135                         pred = tf.reshape(predictions, shape=[-1, n_chars,
                                n_classes])
136                 with tf.name_scope('correct_prediction'):
137                         correct_pred = tf.equal(tf.argmax(pred,2), tf.
                                argmax(y,2))
138                 with tf.name_scope('accuracy'):
139                         accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.
                                float32))
140                 tf.scalar_summary('accuracy',accuracy)
141                 return accuracy * 100.0
142 
143 def softmax_joiner(logits):
```

```
144         return tf.transpose(tf.pack([tf.nn.softmax(logits[0]), tf.nn.
                softmax(logits[1]), \
145                                      tf.nn.softmax(logits[2]), tf.nn.
                                         softmax(logits[3]), \
146                                      tf.nn.softmax(logits[4])]), perm =
                                         [1,0,2])
147
148 weights = {
149         'wc1': weight_variable('wc1',[5, 5, color_channels, 64]),
150         'wc2': weight_variable('wc2',[5, 5, 64, 128]),
151         'wc3': weight_variable('wc3',[5, 5, 128, 256]),
152         'wc4': weight_variable('wc4',[3, 3, 256, 512]),
153         'wd1': weight_variable('wd1',[(resize_height/8)*(resize_width/8)
                *512, fc_num_outputs]),
154         'out1': weight_variable('out1',[fc_num_outputs, n_classes]),
155         'out2': weight_variable('out2',[fc_num_outputs, n_classes]),
156         'out3': weight_variable('out3',[fc_num_outputs, n_classes]),
157         'out4': weight_variable('out4',[fc_num_outputs, n_classes]),
158         'out5': weight_variable('out5',[fc_num_outputs, n_classes])
159 }
160
161 biases = {
162         'bc1': bias_variable([64]),
163         'bc2': bias_variable([128]),
164         'bc3': bias_variable([256]),
165         'bc4': bias_variable([512]),
166         'bd1': bias_variable([fc_num_outputs]),
167         'out1': bias_variable([n_classes]),
168         'out2': bias_variable([n_classes]),
169         'out3': bias_variable([n_classes]),
170         'out4': bias_variable([n_classes]),
171         'out5': bias_variable([n_classes])
172 }
173
174 def train():
175         with tf.Session() as sess:
176                 saver = tf.train.Saver()
177
178                 logits = ocr_net(x, weights, biases, dropout_keep_prob)
179
180                 with tf.name_scope('loss'):
181                         loss = tf.reduce_mean(tf.nn.
                                softmax_cross_entropy_with_logits(logits[0],y
```

```
                    [:,0:36])) +\
182                     tf.reduce_mean(tf.nn.
                            softmax_cross_entropy_with_logits(logits
                            [1],y[:,36:72])) +\
183                     tf.reduce_mean(tf.nn.
                            softmax_cross_entropy_with_logits(logits
                            [2],y[:,72:108])) +\
184                     tf.reduce_mean(tf.nn.
                            softmax_cross_entropy_with_logits(logits
                            [3],y[:,108:144])) +\
185                     tf.reduce_mean(tf.nn.
                            softmax_cross_entropy_with_logits(logits
                            [4],y[:,144:180]))
186
187                 # adding regularizers
188                 regularizers = (tf.nn.l2_loss(weights['wc1']) + tf.
                        nn.l2_loss(biases['bc1']) +
189                                 tf.nn.l2_loss(weights['wc2']) + tf.
                                    nn.l2_loss(biases['bc2']) +
190                                 tf.nn.l2_loss(weights['wc3']) + tf.
                                    nn.l2_loss(biases['bc3']) +
191                                 tf.nn.l2_loss(weights['wc4']) + tf.
                                    nn.l2_loss(biases['bc4'])
192                 )
193                 # Add the regularization term to the loss.
194                 loss += l2_beta_param * regularizers
195                 tf.scalar_summary('loss', loss)
196
197             global_step = tf.Variable(0)
198             with tf.name_scope('learning_rate'):
199                 learning_rate = tf.train.exponential_decay(
                        initial_learning_rate, global_step, decay_steps,
                         decay_rate)
200                 tf.scalar_summary('learning_rate',learning_rate)
201             with tf.name_scope('train'):
202                 optimizer = tf.train.MomentumOptimizer(
                        learning_rate, momentum).minimize(loss,
                        global_step=global_step)
203             pred = softmax_joiner(logits)
204             accuracy = accuracy_func(pred, y)
205
206             init = tf.initialize_all_variables()
207             merged = tf.merge_all_summaries()
```

```
208                 train_writer = tf.train.SummaryWriter(summaries_dir + '/
                        train', sess.graph)
209
210                 sess.run(init)
211                 step = 1# Keep training until reach max iterations
212                 while step * batch_size < training_iters:
213                         batch = data_train.next_batch(batch_size)
214                         batch_labels = batch[1]
215                         # Fit training using batch data
216                         smry, _, l = sess.run([merged, optimizer, loss],
                                feed_dict={x: batch[0], y: batch_labels})
217                         if step % display_step == 0:
218                                 # Calculate batch accuracy
219                                 run_options = tf.RunOptions(trace_level=tf.
                                        RunOptions.FULL_TRACE)
220                                 run_metadata = tf.RunMetadata()
221                                 summary, acc = sess.run([merged,accuracy],
222                                                         feed_dict={x: batch
                                                            [0], y: batch
                                                            [1]},
223                                                         options=run_options
                                                            ,
224                                                         run_metadata=
                                                            run_metadata)
225                                 train_writer.add_run_metadata(run_metadata,
                                         'step%03d' % step)
226                                 train_writer.add_summary(summary, step)
227                                 print "Iter " + str(step*batch_size) + ",
                                        Minibatch Loss= " + "{:.6f}".format(l) +
                                         ", Training Accuracy= " + "{:.2f}%".
                                        format(acc)
228                         else:
229                                 train_writer.add_summary(smry, step)
230                         step += 1
231                 print "Optimization Finished!"
232                 train_writer.close()
233                 elapsed_time = time.time() - start_time
234                 hours = elapsed_time / 3600
235                 minutes = (elapsed_time % 3600) / 60
236                 seconds = (elapsed_time % 3600) % 60
237                 print "Total time was: " + "{:.0f}h".format(hours) + ",
                        {:.0f}m".format(minutes) + ", {:.0f}s".format(seconds)
238                 # Save the variables to disk.
```

```
239                 save_path = saver.save(sess, "models/sintegra_sc_model.ckpt
                        ")
240                 print("Model saved in file: %s" % save_path)
241
242                 test_writer = tf.train.SummaryWriter(summaries_dir + '/test
                        ')
243                 test_batch = data_test.next_batch(n_test_samples)
244                 summ, acc = sess.run([merged,accuracy], feed_dict={x:
                        test_batch[0], y: test_batch[1]})
245                 print "Testing Accuracy: " + "{:.2f}%".format(acc)
246                 test_writer.add_summary(summ,step)
247
248 if tf.gfile.Exists(summaries_dir):
249         tf.gfile.DeleteRecursively(summaries_dir)
250 tf.gfile.MakeDirs(summaries_dir)
251 if not tf.gfile.Exists('models'):
252         tf.gfile.MakeDirs('models')
253 train()
```

7.3 IMPLEMENTAÇÃO DA FUNÇÃO QUE EXECUTA O RECONHECIMENTO DE UMA IMAGEM DE CAPTCHA, ARQUIVO *CAPTCHA_RECOGNIZER.PY*.

```
1
2 import tensorflow as tf
3 import glob, os
4 import numpy as np
5 import cv2
6 import random
7
8 class OCR_recognizer(object):
9     def __init__(self, model_filename, num_classes=36, num_channels=1,
          num_chars=5, resize_height=24, resize_width=88):
10         self.model_filename = model_filename
11         self.num_classes = num_classes
12         self.num_channels = num_channels
13         self.num_chars = num_chars
14         self.resize_height = resize_height
15         self.resize_width = resize_width
16         # tf Graph input
17         fc_num_outputs = 4096
```

```
18          self.l2_beta_param = 3e-4
19          self.initial_learning_rate = 0.01
20          self.decay_steps = 1000
21          self.decay_rate = 0.9
22          self.momentum = 0.9
23          self.dropout_keep_prob = 0.5
24          self.x = tf.placeholder(tf.float32, [None, resize_height,
                resize_width, num_channels])
25          self.y = tf.placeholder(tf.float32, [None, num_chars*num_classes])
26          self.weights = {
27              'wc1': self.weight_variable('wc1',[5, 5, num_channels, 64]),
28              'wc2': self.weight_variable('wc2',[5, 5, 64, 128]),
29              'wc3': self.weight_variable('wc3',[5, 5, 128, 256]),
30              'wc4': self.weight_variable('wc4',[3, 3, 256, 512]),
31              'wd1': self.weight_variable('wd1',[(resize_height/8)*(
                    resize_width/8)*512, fc_num_outputs]),
32              'out1': self.weight_variable('out1',[fc_num_outputs,
                    num_classes]),
33              'out2': self.weight_variable('out2',[fc_num_outputs,
                    num_classes]),
34              'out3': self.weight_variable('out3',[fc_num_outputs,
                    num_classes]),
35              'out4': self.weight_variable('out4',[fc_num_outputs,
                    num_classes]),
36              'out5': self.weight_variable('out5',[fc_num_outputs,
                    num_classes])
37          }
38          self.biases = {
39              'bc1': self.bias_variable([64]),
40              'bc2': self.bias_variable([128]),
41              'bc3': self.bias_variable([256]),
42              'bc4': self.bias_variable([512]),
43              'bd1': self.bias_variable([fc_num_outputs]),
44              'out1': self.bias_variable([num_classes]),
45              'out2': self.bias_variable([num_classes]),
46              'out3': self.bias_variable([num_classes]),
47              'out4': self.bias_variable([num_classes]),
48              'out5': self.bias_variable([num_classes])
49          }
50          self.logits = self.ocr_net(self.x, self.weights, self.biases, self.
                dropout_keep_prob)
51          self.define_graph()
52
```

```
53      def __enter__(self):
54          return self
55
56      def __exit__(self, *err):
57          tf.reset_default_graph()
58
59      def create_captcha(self, pathAndFilename):
60          img = cv2.imread(pathAndFilename, cv2.IMREAD_GRAYSCALE)
61          img = cv2.resize(img,(self.resize_width, self.resize_height),
                interpolation=cv2.INTER_AREA)
62          filename, ext = os.path.splitext(os.path.basename(pathAndFilename))
63          label = self.create_label(filename)
64          return (img, label)
65
66      def create_label(self, filename):
67          label = []
68          for c in filename:
69              ascii_code = ord(c)
70              if ascii_code < 58:
71                  char_value = ascii_code - 48
72              else:
73                  char_value = ascii_code - 87
74              label.append(char_value)
75          return self.dense_to_one_hot(label)
76
77      def dense_to_one_hot(self, labels_dense):
78          num_labels = len(labels_dense)
79          index_offest = np.arange(num_labels) * self.num_classes
80          labels_one_hot = np.zeros((num_labels, self.num_classes))
81          labels_one_hot.flat[index_offest + labels_dense] = 1
82          labels_one_hot = labels_one_hot.reshape(num_labels*self.num_classes
                )
83          return labels_one_hot
84
85      def get_prediction_string(self, one_hot_predictions):
86          reshaped = one_hot_predictions.reshape(self.num_chars, self.
                num_classes)
87          correct_pred = np.argmax(reshaped, 1)
88          final_string = ""
89          for char_value in correct_pred:
90              if char_value > 9:
91                  final_string += chr(char_value + 87)
92              else:
```

```
93                  final_string += chr(char_value + 48)
94          return final_string
95
96      def image_to_batch(self, image_path):
97          img, label = self.create_captcha(image_path)
98          imgs = []
99          labels = []
100         imgs.append(img)
101         labels.append(label)
102         imgs = np.array(imgs).reshape((-1, self.resize_height, self.
                resize_width, self.num_channels)).astype(np.float32)
103         labels = np.array(labels)
104         return imgs[0:1], labels[0:1]
105
106     def weight_variable(self, name, shape):
107         return tf.get_variable(name, shape, initializer=tf.contrib.layers.
                xavier_initializer())
108
109     def bias_variable(self, shape):
110         initial = tf.constant(0.0, shape=shape)
111         return tf.Variable(initial, trainable=True)
112
113     def max_pool(self, x, k, name):
114         return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
                padding='SAME', name=name)
115
116     def conv2d(self, x, W, B, name):
117         conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
118         bias = tf.nn.bias_add(conv, B)
119         conv = tf.nn.relu(bias, name=name)
120         return conv
121
122     def ocr_net(self, _x, _weights, _biases, keep_prob):
123         _x = tf.reshape(_x, shape=[-1, self.resize_height, self.
                resize_width, self.num_channels])
124
125         conv1 = self.conv2d(_x, _weights['wc1'], _biases['bc1'], 'conv1')
126         lrn1 = tf.nn.local_response_normalization(conv1)
127         pool1 = self.max_pool(lrn1, k=2, name='pool1')
128
129         conv2 = self.conv2d(pool1, _weights['wc2'], _biases['bc2'], 'conv2'
                )
130         lrn2 = tf.nn.local_response_normalization(conv2)
```

```
131         pool2 = self.max_pool(lrn2, k=2, name='pool2')
132 
133         conv3 = self.conv2d(pool2, _weights['wc3'], _biases['bc3'], 'conv3'
                )
134         lrn3 = tf.nn.local_response_normalization(conv3)
135         pool3 = self.max_pool(lrn3, k=2, name='pool3')
136 
137         conv4 = self.conv2d(pool3, _weights['wc4'], _biases['bc4'], 'conv4'
                )
138 
139         dropout = tf.nn.dropout(conv4, keep_prob)
140 
141         shape = dropout.get_shape().as_list()
142         reshaped = tf.reshape(dropout, [-1, _weights['wd1'].get_shape().
                as_list()[0]])
143 
144         fc1 = tf.nn.relu(tf.matmul(reshaped, _weights['wd1']) + _biases['
                bd1'], name='fc1')
145 
146         fc21 = tf.nn.relu(tf.matmul(fc1, _weights['out1']) + _biases['out1'
                ], name='fc21')
147 
148         fc22 = tf.nn.relu(tf.matmul(fc1, _weights['out2']) + _biases['out2'
                ], name='fc22')
149 
150         fc23 = tf.nn.relu(tf.matmul(fc1, _weights['out3']) + _biases['out3'
                ], name='fc23')
151 
152         fc24 = tf.nn.relu(tf.matmul(fc1, _weights['out4']) + _biases['out4'
                ], name='fc24')
153 
154         fc25 = tf.nn.relu(tf.matmul(fc1, _weights['out5']) + _biases['out5'
                ], name='fc25')
155 
156         return [fc21, fc22, fc23, fc24, fc25]
157 
158     def softmax_joiner(self, logits):
159         return tf.transpose(tf.pack([tf.nn.softmax(logits[0]), tf.nn.
                softmax(logits[1]), \
160                                      tf.nn.softmax(logits[2]), tf.nn.
                                         softmax(logits[3]), \
161                                      tf.nn.softmax(logits[4])]), perm =
                                         [1,0,2])
```

```
162
163     def define_graph(self):
164         self.saver = tf.train.Saver()
165         loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.
                logits[0], self.y[:,0:36])) +\
166                 tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.
                    logits[1], self.y[:,36:72])) +\
167                 tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.
                    logits[2], self.y[:,72:108])) +\
168                 tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.
                    logits[3], self.y[:,108:144])) +\
169                 tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(self.
                    logits[4], self.y[:,144:180]))
170
171         # adding regularizers
172         regularizers = (tf.nn.l2_loss(self.weights['wc1']) + tf.nn.l2_loss(
                self.biases['bc1']) +
173                         tf.nn.l2_loss(self.weights['wc2']) + tf.nn.l2_loss(
                            self.biases['bc2']) +
174                         tf.nn.l2_loss(self.weights['wc3']) + tf.nn.l2_loss(
                            self.biases['bc3']) +
175                         tf.nn.l2_loss(self.weights['wc4']) + tf.nn.l2_loss(
                            self.biases['bc4']))
176         # Add the regularization term to the loss.
177         self.loss = loss + self.l2_beta_param * regularizers
178
179         global_step = tf.Variable(0)
180         learning_rate = tf.train.exponential_decay(self.
                initial_learning_rate, global_step, self.decay_steps, self.
                decay_rate)
181
182         self.optimizer = tf.train.MomentumOptimizer(learning_rate, self.
                momentum).minimize(loss, global_step=global_step)
183         self.pred = self.softmax_joiner(self.logits)
184
185     def recognize(self, image_path):
186         with tf.Session() as sess:
187             init = tf.initialize_all_variables()
188             sess.run(init)
189             self.saver.restore(sess, self.model_filename)
190
191             batch = self.image_to_batch(image_path)
192             # Fit training using batch data
```

```
193             _, predictions, l = sess.run([self.optimizer, self.pred, self.
                    loss], feed_dict={self.x: batch[0], self.y: batch[1]})
194         return self.get_prediction_string(predictions)
```

7.4 IMPLEMENTAÇÃO DA FUNÇÃO QUE CALCULA A ACURÁCIA DO MODELO DE REDE NEURAL, ARQUIVO *CAPTCHA_ACCURACY.PY*.

```
1
2  import glob, os
3  from captcha_recognizer import OCR_recognizer
4
5  recognizer = OCR_recognizer('models/sintegra_sc_model.ckpt')
6
7  def calculate_accuracy(dir, pattern):
8      corrects = 0
9      total = 0
10     for pathAndFilename in glob.iglob(os.path.join(dir, pattern)):
11         label, ext = os.path.splitext(os.path.basename(pathAndFilename))
12         prediction = recognizer.recognize(pathAndFilename)
13         if prediction == label:
14             corrects += 1
15         total += 1
16         print "Prediction: " + prediction
17         print "Label: " + label
18     accuracy = (float(corrects)/float(total)) * 100
19     print "The total accuracy was: " + "{:.2f}%".format(accuracy)
20     return accuracy
21
22 calculate_accuracy(r'./images/accuracy', r'*.png')
```

REFERÊNCIAS

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

[2] 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/>.

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

[4] 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>.

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

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

[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] 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>.

[9] 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>.

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

[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/>.

Redes Neurais Convolucionais de Profundidade para Reconhecimento de Textos em Imagens de CAPTCHA

Vitor Arins Pinto[1]

[1]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina (UFSC)
Santa Catarina – SC – Brazil

vitor.arins@grad.ufsc.br

Abstract. *Currently many applications on the Internet follow the policy of keeping some data accessible to the public. In order to do this, it's 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 may not always come from a human. Companies specializing in Big data have a great interest in data from public sources in order to make analysis 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 website. This website may not be prepared for such a great 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.*

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 website. Tal website pode não estar preparado para um volume de consultas tão grande 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.*

Introdução

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.

Objetivo

O objetivo geral do artigo é analisar o treinamento e aplicação de redes neurais convolucionais de profundidade para o reconhecimento de texto em imagens de CAPTCHA. Com isso será retratada 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.

Conceitos teóricos

Classificador Logístico

Um classificador logístico (geralmente chamado de regressão logística[Bengio and Courville 2016]) recebe como entrada uma informação, como por exemplo os pixels de uma imagem, e aplica uma função linear a eles para gerar suas predições. Uma função linear é 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.

$$WX + b = Y \tag{1}$$

Na equação 1, "X" é como chamaremos o 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.

Função *Softmax*

Como cada imagem pode ter um e somente um rótulo possível, é necessário transformar as pontuações geradas pelo classificador logístico em probabilidades. É essencial 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 transformar essas pontuações em probabilidades utiliza-se uma função chamada ***Softmax***[Bengio and Courville 2016].

One-Hot Encoding

Para facilitar o treinamento é preciso representar de forma matemática os rótulos de cada exemplo que iremos alimentar à 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 simples e geralmente chamada de *One-Hot Encoding*. Com isso é possível medir a eficiência do treinamento apenas comparando dois vetores.

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 coletar 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 é um conjunto de **mapas de características** da imagem original. Como exemplo, transforma-se 3 mapas de carcterísticas (canais de cores) para uma quantidade K de mapas de características.

Uma rede convolucional[Dumoulin et al. 2016] será basicamente uma rede neural de profundidade. Ao invés de empilhar camadas de multiplicação de matrizes, empilha-se convoluções. No começo haverá uma imagem grande que possui apenas os valores de pixel como informação. Em seguida são aplicadas convoluções que irão "espremer" as dimensões espaciais e aumentar a profundidade. No final é possivel conectar o classificador e ainda lidar apenas com parâmetros que mapeiam o conteúdo da imagem.

ReLU

Modelos lineares são simples e estáveis numericamente, mas podem se tornar ineficientes ao longo do tempo. Portanto, para adicionar mais camadas ao modelo será necessário introduzir alguns cálculos não lineares entre camadas. Em arquiteturas de profundidade, as funções de ativação dos neurônios se chamam *Rectified Linear Units* (ReLUs)[Bengio and Courville 2016], e são capazes de introduzir os cálculos necessários aos modelos que possuem mais de uma camada. Essas são as funções não lineares mais simples que existem. Elas são lineares ($y = x$) se **x** é maior que **0**, senão ficam iguais a **0** ($y = 0$). Isso simplifica o uso de *backpropagation* e evita problemas de saturação, fazendo o aprendizado ficar muito mais rápido.

Max pooling

Após a camada convolucional adiciona-se uma camada de *pooling* que irá receber todas as convoluções e combiná-las da seguinte forma[Dumoulin et al. 2016]. Para cada ponto nos mapas de características a execução desta camada olha para uma pequena vizinhança ao redor deste ponto. Com esses valores em mãos é possível calcular o valor máximo dessa vizinhança.

Dropout

Uma forma de regularização que previne o *overfitting*[1] é o *dropout*[Krizhevsky et al.]. Supondo que temos uma camada conectada à outra em uma 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, é atribuído o valor 0 para metade desses valores. Basicamente metade dos dados que estão fluindo pela rede neural é destruída aleatoriamente.

Camada completamente conectada

De acordo com Krizhevsky[Krizhevsky et al.], 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*.

Devido a quantidade de componentes presentes na estrutura de redes neurais é aparente a complexidade quanto ao entendimento do funcionamento geral. A figura 1 tenta explicar como esses componentes se conectam e em qual sequência. Os valores são completamente fictícios e não condizem com um cálculo real.

[1] O *overfitting* ocorre quando um modelo de rede neural se encaixa muito bem em um conjunto de dados e acaba memorizando propriedades do conjunto de treinamento que não servem para o conjunto de teste.

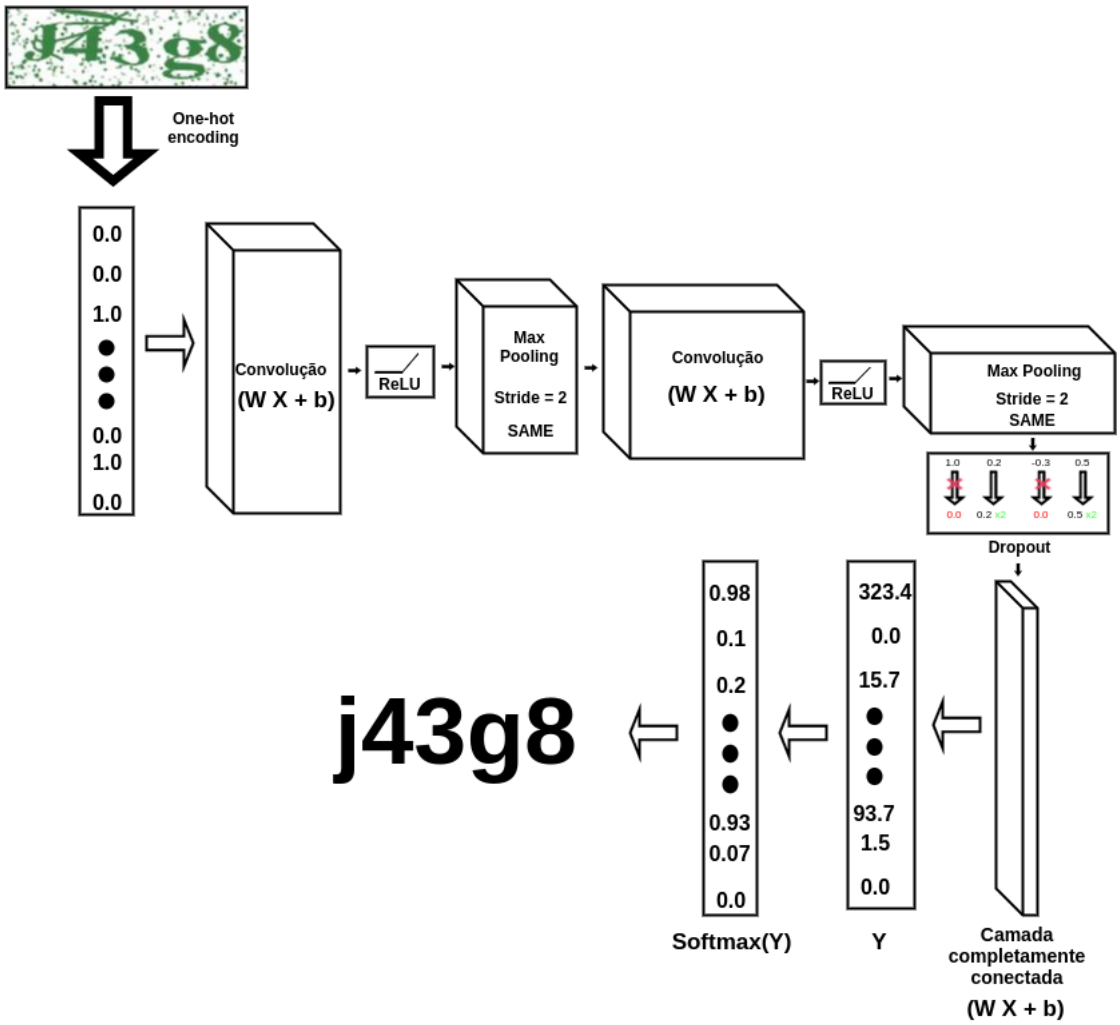

Figura 1. Exemplo da composição de todos os componentes presentes em redes neurais convolucionais de profundidade.

Experimento

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. Para o treinamento também será necessário um conjunto separado para teste que não possui nenhuma imagem presente no conjunto de treinamento. O *dataset* de treinamento terá cerca de 96% das imagens, e o *dataset* de testes terá 4% das imagens.

Junto com a geração do conjunto de dados, ocorre o pré-processamento das imagens. Durante o pré-processamento as imagens são transformadas para uma representação em escala de cinza. Por fim as imagens são redimensionadas para um tamanho menor, tornando os cálculos mais eficientes durante o treinamento da rede neural.

Treinamento

Após gerado o conjunto de dados, é possível trabalhar no treinamento do modelo da rede neural. Para isso será usado o *framework* ***TensorFlow***[TensorFlow] destinado à *Deep Learning*. Também será desenvolvido 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.

Avaliação de acurácia

Com uma nova amostra de imagens, será feita a execução do teste do modelo que obteve melhor performance durante o treinamento. 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.

Desenvolvimento

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) e o treinamento da rede neural é feito em uma *Session*.

O projeto é composto por 5 tarefas de implementação:

- Desenvolvimento do leitor e processador do conjunto de dados.
- Desenvolvimento da função que monta a rede neural.
- Configuração da rede neural para otimização dos resultados.
- Desenvolvimento da etapa de treinamento da rede neural.
- Desenvolvimento da etapa de teste e acurácia do modelo da rede neural.

Testes

Treinamento com 200 mil iterações

Inicialmente é realizado um treinamento com 200 mil iterações. 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 acurácia fica em torno de 5% até um momento que começa a subir. Ao final do treinamento foi alcançado um valor máximo de acurácia igual a **84,38%** no conjunto de treinamento, **79,6%** no conjunto de teste.

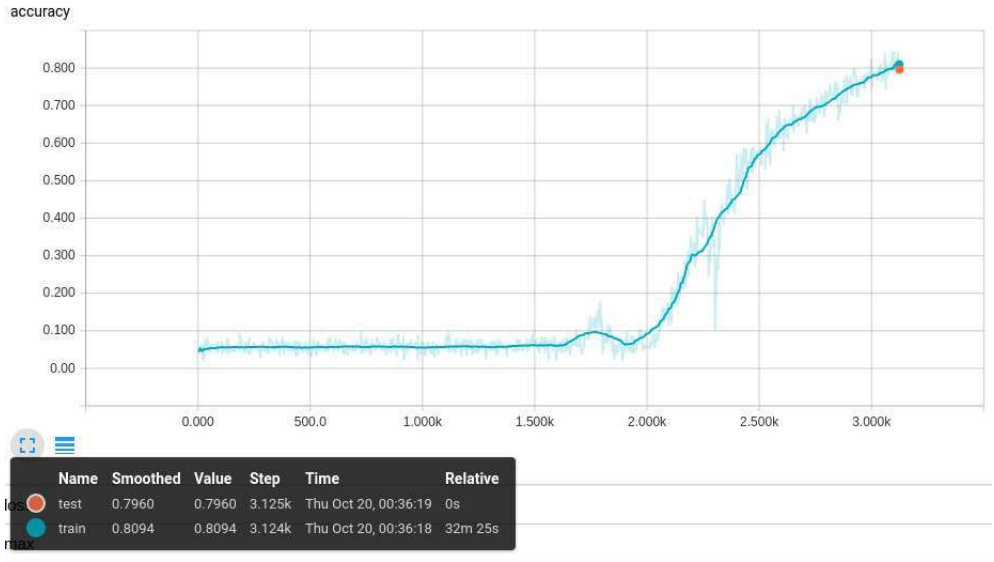

Figura 2. Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 200 mil iterações.

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. O tempo total de treinamento foi de **1 hora 18 minutos e 23 segundos**.

Ao final do treinamento foi alcançado o valor máximo de acurácia igual a **98,75%** no conjunto de treinamento, **81,37%** no conjunto de teste.

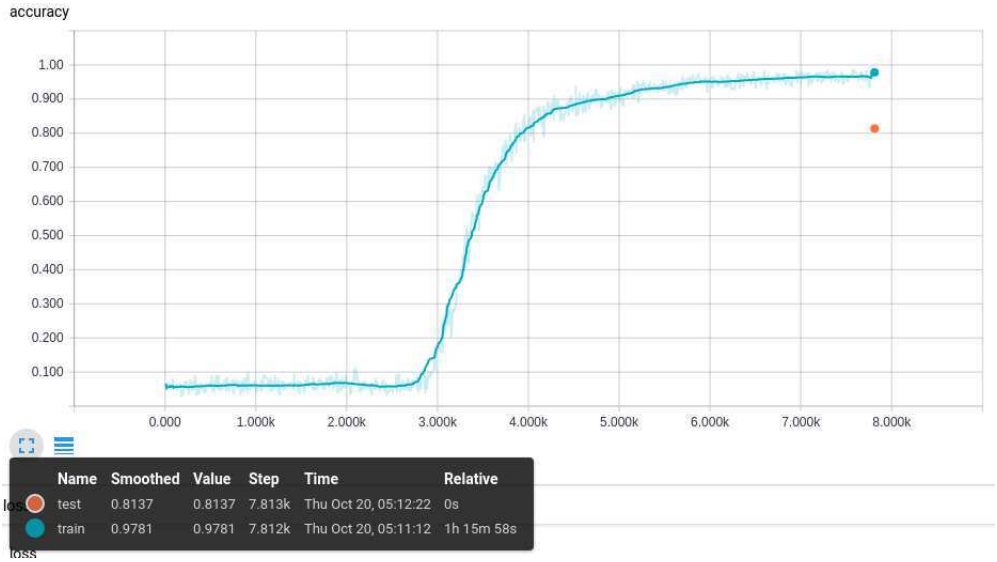

Figura 3. Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 500 mil iterações.

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**.

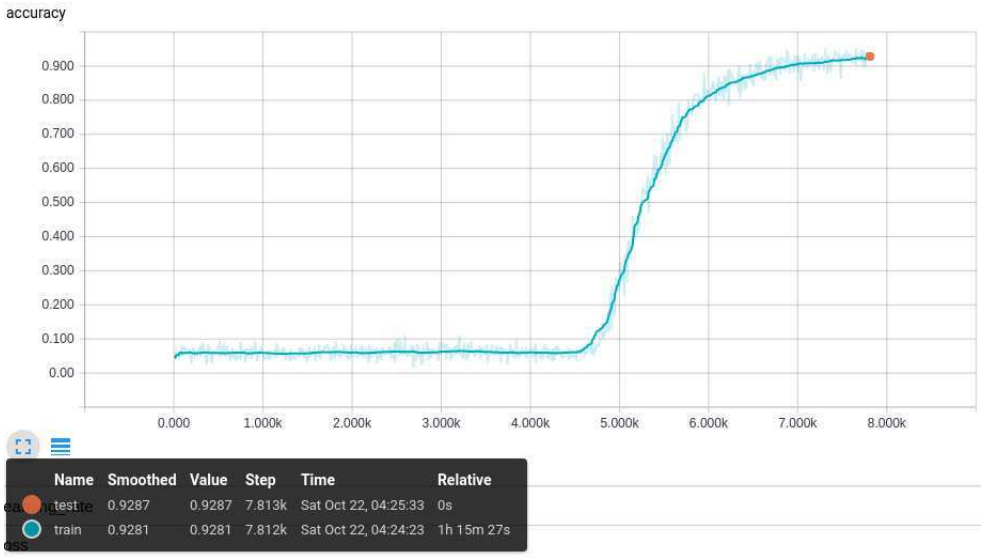

Figura 4. Gráfico da acurácia em relação ao número de passos para o treinamento da rede com 500 mil iterações e probabilidade de *dropout* igual a 50%.

Ao final do treinamento foi alcançado o valor máximo de acurácia igual a **95,31%** no conjunto de treinamento, **92,87%** no conjunto de teste.

Conclusão

Os testes realizados em todos os casos mostraram ser possível atingir um resultado razoável na tarefa de reconhecimento de textos em imagens, isso com poucos ajustes à configuração de treinamento de redes neurais. Atualmente a quantidade de exemplos e tutoriais disponíveis para tarefas de aprendizado de máquina é imenso. Fica claro que é possível implementar classificadores mesmo com poucos recursos.

Diante do objetivo alcançado pelo trabalho, fica aparente que fontes públicas de dados podem estar vulneráveis.

Trabalhos futuros

Como possíveis trabalhos futuros, cita-se:

- Fazer um melhor uso das informações geradas pelo processo de treinamento para gerar heurísticas mais inteligentes. Um exemplo seria utilizar outros tipos de otimizadores para a função da perda.
- Estender o sistema para realizar o reconhecimento de outros tipos de CAPTCHAs.
- Estender o sistema para realizar o reconhecimento de tipos de CAPTCHAs que possuem um tamanho de texto variável.
- Realizar um estudo sobre *Web crawlers* em fontes públicas que utilizam CAPTCHA, executando o sistema proposto neste trabalho.

- Implementar um sistema de reconhecimento de CAPTCHAs mais avançados que solicitam a classificação de uma cena completa ou identificação de objetos em imagens.
- Estudar um artifício mais efetivo para o bloqueio de consultas automatizadas em *websites*.

Considera-se de extrema importância a implementação de projetos desse tipo pois o mesmo auxilia na compreensão e aplicação de Inteligencia Artificial em casos específicos.

Referências

Bengio, I. G. Y. and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Dumoulin, V., Visin, F., and Box, G. E. P. (2016). A guide to convolution arithmetic for deep learning.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. Acessado: 21/10/2016.

TensorFlow. TensorFlow — an Open Source Software Library for Machine Intelligence. Acessado: 03/08/2016.