Universidade Estadual de Campinas
MC970 - Introdução ao Processamento de Imagem Digital
Prof. Dr. Hélio Pedrini

Trabalho 3

Mateus de Lima Almeida
242827

21 de maio de 2025

1 Introdução

O presente trabalho aborda a implementação, realização e exploração de algoritmos para alinhamento automático de imagens de documentos sugeridos em sala na matéria de Introdução ao Processamento de Imagem Digital (MC920). Com o objetivo de introduzir, solidificar e explorar o conhecimento aos conceitos de técnicas de alinhamento automático de imagens através da utilização da técnica de projeção horizontal e da Transformada de Hough. Neste presente relatório, imagens foram transformadas, processadas, e seus resultados foram analisados.

Este relatório detalha a metodologia adotada na execução dos exercícios sugeridos e também realiza a discussão de seus resultados, bem como a exploração de hipóteses surgidas no processo, junto também com a discussão de seus testes e resultados.

2 Materiais e Métodos

Para todas as tarefas a seguir, os materiais utilizados foram a linguagem Python 3.9 junto com as bibliotecas OpenCV2, Scikit-Image, Numpy e Pillow. Para utilização das bibliotecas e execução dos códigos, também foi utilizado um ambiente virtual (venv), onde foram instaladas as bibliotecas citadas.

2.1 Organização de Arquivos

A pasta do projeto foi organizada de modo a facilitar a visualização dos resultados de cada exercício de maneira independente. Dentro da pasta estão:

- Arquivo Python *alinhar.py*
- Arquivo Python *analiseOCR.py*
- Arquivo *requirements.txt* com as dependências necessárias do projeto.
- Pasta *images* que contém:
  - Imagens PNG que serão os Inputs dos exercícios
  - Pasta *outputs* que contém 2 pastas (projecao e hough) contendo as imagens-saída de cada modo.

2.2 Versão Python e Bibliotecas

A versão de Python utilizada, junto com as bibliotecas e suas versões foram as seguintes:

- Python 3.9.1
- Pillow 11.1.0
- Opencv-python 4.11.0.86
- Numpy 2.0.2
- Matplotlib 3.9.4
- Pytesseract 0.3.13

2.3 Execução do código principal

Os códigos feitos para realizar as operações e transformações sugeridas neste trabalho foram todos executados dentro de um ambiente virtual *venv*, então é recomendável que para a reprodução seja feito o mesmo. É necessário que as dependências de *requirements.txt* sejam instaladas.

Para executar o programa, basta inserir o comando básico para rodar um arquivo *.py* no terminal (certifique-se de estar na pasta correta).

```
python alinhar.py imagem_entrada.png modo imagem_saida.png
```

A imagem selecionada deve estar presente na pasta *images*. A seguir, uma lista das imagens presentes na pasta, que podem ser usadas como input, e dos modos presentes para alinhamento:

- Imagens:
  - neg_4.png
  - neg_28.png
  - partitura.png
  - pos_24.png
  - pos_41.png
  - sample1.png
  - sample2.png
- Modos:
  - projecao
  - hough

Caso seja desejado, mais imagens podem ser adicionadas manualmente na pasta *images* para serem usadas como input.

Por fim, o argumento *imagem_saida.png* é o nome que a imagem de saída terá. **É necessário adicionar *.png* nos nomes de entrada e saída, caso contrário o código não funcionará**. É aconselhado manter o nome de saída igual ao nome da imagem de entrada, principalmente caso queira utilizar o script de análise com Tesseract OCR, pois as imagens precisam ter o mesmo nome para seu funcionamento.

2.4 Execução do código de análise com Tesseract OCR

Em adição às análises feitas nas seções anteriores, um novo script python foi criado para realizar a análise de caracteres reconhecidos nas imagens. O script (*analiseOCR.py*) funciona com o seguinte comando:

```
python analiseOCR.py nome.png modo
```

Onde *nome* é o nome da imagem que será analisada (ex: sample1.png) e *modo* é a técnica que gerou a imagem que será comparada à imagem original. Por exemplo, se desejamos comparar a eficiência do alinhamento de *sample1.png* com a Transformada de Hough, o comando utilizado deve ser:

```
python analiseOCR.py sample1.png hough
```

Dessa forma, a comparação feita será entre a imagem original e a alinhada gerada através da técnica com Transformada de Hough.

É importante destacar que, para que o código funcione, o alinhamento da imagem desejada precisa ter sido realizado anteriormente pelo método desejado, pois esse script analisa as imagens resultado na pasta de outputs (e as imagens só estarão lá se já tiverem sido alinhadas pelo código principal).

3 Resultados, testes e discussão

3.1 Técnica baseada em projeção horizontal

O objetivo da primeira tarefa proposta foi utilizar uma técnica de detecção e correção de inclinação baseada em projeção horizontal. Essa técnica consiste em encontrar o ponto onde o histograma de projeção horizontal da imagem otimiza uma função objetivo dada, que nesse caso foi a soma dos quadrados das diferenças dos valores em células adjacentes, assim como sugerido para a tarefa.

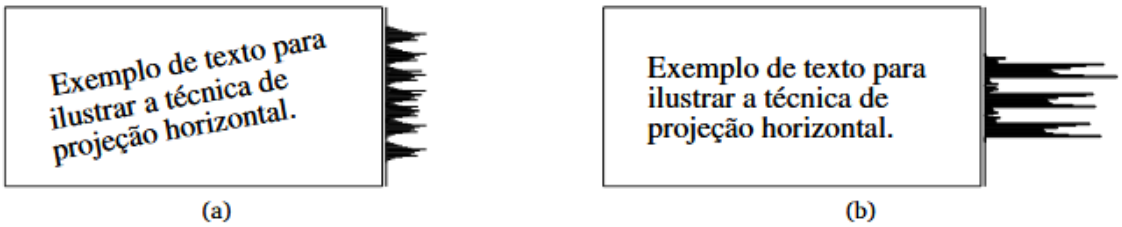


Figura 1: Exemplo de imagens com (e sem) inclinação e seus histogramas de projeção horizontal

A primeira imagem usada para testar o método é a *sample1.png* (Figura 2).

Figura 2: sample1.png

O primeiro passo para aplicação desta técnica foi a binarização da imagem entrada, isto é, transformar a imagem em greyscale em uma imagem binária onde todo pixel é 0 ou 255 (preto ou branco). Para isso, foi utilizado o seguinte trecho:

```
img_bin = np.where(img_cinza > 127, 0, 255).astype(np.uint8)
```

Onde *img_cinza* é a imagem entrada em greyscale, e *img_bin* é a imagem binarizada. Como é possível observar no trecho de código acima, o threshold escolhido para binrização foi 127, ou seja, todos os pixels com valor maior que 127 se tornaram 255, e os com valor menor ou igual à 127 se tornaram 0. O resultado pode ser visto na Figura 3.

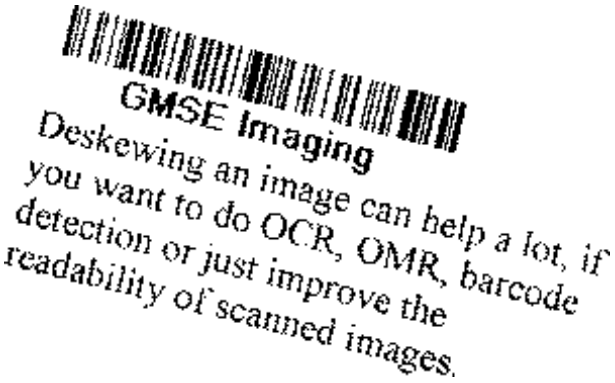


Figura 3: sample1.png binarizada

Após a binarização da imagem, é realizado o processo de encontrar o melhor ângulo de rotação, isto é, o ângulo que maximiza a soma dos quadrados das diferenças de cada linha da projeção. Isso significa que, na prática, o código busca encontrar o ângulo onde há a maior diferença de amplitude entre as linhas (consecutivas) da imagem, pois isso significa que ela está bem alinhada pois imagens com textos alinhados possuem linhas com muitos (ou todos) pixels brancos seguidas de linhas com muitos pixels pretos.

O algoritmo feito itera sobre um intervalo de ângulos [-30, 30], com um passo de 0,5. Os valores do intervalo foram escolhidos levando em consideração que as imagens de documentos não costumam possuir desalinhamentos maiores que esses valores, mas em casos onde uma imagem, por algum motivo, se encontra extremamente desalinhada, isso obviamente introduz uma limitação ao código, onde inclinações maiores que 30^{o} não serão reconhecidas corretamente, nesse caso, é possível aumentar o intervalo (com o aumento notável do custo computacional e de tempo). O valor do passo foi escolhido de forma quase arbitrária, considerando que não deveria ser pequeno demais (para não aumentar muito o tempo/custo do código), mas também não deveria ser grande demais (para que o resultado ainda seja satisfatório e a discretização não atrapalhe o resultado).

Para melhor visualização das explicações, o trecho do código que seleciona o melhor ângulo será apresentado a seguir:

```
def encontrar_angulo_projecao(imagem_bin):
    melhor_angulo = 0
    melhor_valor = -1

    for angulo in np.arange(-30, 30.1, 0.5):
        (h, w) = imagem_bin.shape
        matriz_rotacao = cv2.getRotationMatrix2D((w // 2, h // 2), angulo, 1.0)
        imagem_rotacionada = cv2.warpAffine(imagem_bin, matriz_rotacao, (w, h),
        flags=cv2.INTER_LINEAR, borderValue=0)
        projecao = np.sum(imagem_rotacionada, axis=1)
        valor = np.sum(np.diff(projecao) ** 2) #soma dos quadrados das diferenças
```

```
        if valor > melhor_valor:
            melhor_valor = valor
            melhor_angulo = angulo

    return melhor_angulo
```

Os valores em que *melhor_angulo* e *melhor_valor* são iniciados são intuitivos, ambos os valores são pensados para evitarem falsos positivos e valores incorretos. Dentro do loop, primeiramente, as medidas da imagem são obtidas, e em seguida, utilizando *getRotationMatrix2D()*, a matriz de rotação do ângulo da iteração é calculada, ou seja, na primeira iteração, será calculada a matriz de rotação para o ângulo -30, o parâmetro 1.0 dis respeito ao redimensionamento, que no caso não é aplicado pois o valor 1.0 foi utilizado.

Em seguida, *warpAffine()* aplica a matriz de rotação na imagem. Os parâmetros escolhidos para essa transformação são importantes: os primeiros três parâmetros são intuitivos (imagem a ser transformada, matriz de transformação e dimensões), o parâmetro *flags=cv2.INTER_LINEAR* diz respeito ao método de interpolação, isto é, calcula uma média ponderada de pixels vizinhos para evitar distorções na transformação (nesse caso rotação) da imagem. Já o parâmetro *borderValue=0* diz respeito aos pixels faltantes que serão preenchidos após a rotação. Nesse caso, como a imagem está sendo rotacionada apenas para encontrar o melhor ângulo, o preenchimento é feito com 0, isto é, com pixels pretos, o que evita a adição de artefatos que podem prejudicar o reconhecimento das linhas.

Após a rotação ser feita de acordo com o ângulo da iteração, o valor da soma dos quadrados das diferenças da projeção horizontal é calculado e comparado com o melhor (maior) valor guardado até o momento, se o novo valor for maior, ele será guardado como o novo melhor valor e o ângulo que o gerou será guardado como o novo melhor ângulo. Dessa forma, ao final de todas as iterações, teremos tanto o maior valor da soma dos quadrados quanto o ângulo que o gerou. Como apenas o ângulo será utilizado posteriormente, a função retorna apenas o valor do ângulo.

Após a obtenção do ângulo ótimo, uma nova rotação é feita usando:

```
matriz_rot = cv2.getRotationMatrix2D((w // 2, h // 2), angulo, 1.0)
cv2.warpAffine(imagem, matriz_rot, (w, h), flags=cv2.INTER_LINEAR, borderValue=255)
```

É possível observar que o processo de rotação acima é idêntico ao anterior exceto pelo fato de estarmos aplicando a rotação na imagem original (e não na binarizada) e pelo valor do parâmetro *borderValue=255*, que agora preenche os pixels faltantes com o valor 255 em vez de 0. Isso é feito pois esta rotação está sendo aplicada para gerar a imagem definitiva, nesse caso, como a maioria dos documentos possui fundo branco (ou próximo disso), preencher as pequenas lacunas com pixels brancos gera um bom resultado.

Após aplicar a técnica baseada em projeção horizontal, temos o resultado seguinte:

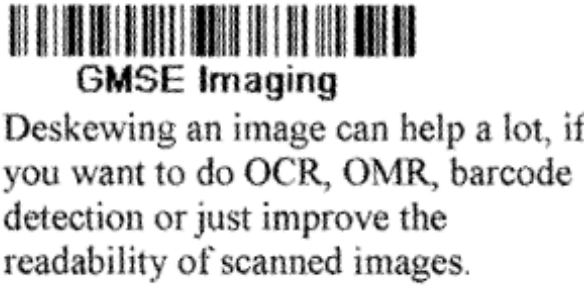


Figura 4: sample1.png após alinhamento

Para visualizar melhor a diferença entre as projeções das imagens apresentadas, segue os histogramas de projeção horizontal da imagem original e da imagem alinhada (Figuras 5 e 6 respectivamente):

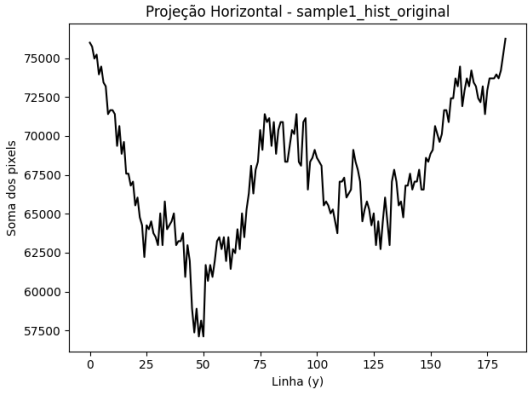


Figura 5: histograma de projeção da imagem original

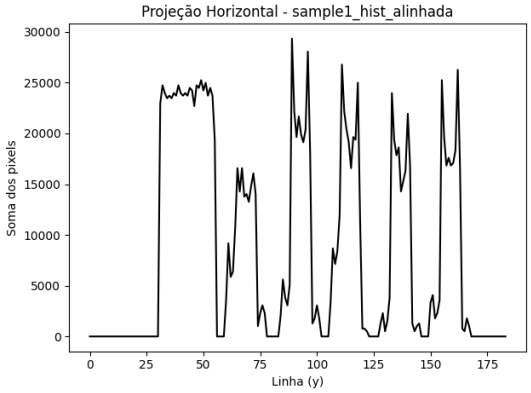


Figura 6: histograma de projeção da imagem alinhada

Como é possível observar, no histograma da imagem alinhada existem picos bem mais definidos quando comparado ao histograma da imagem original. Esses picos com grande amplitude e claramente separados por vales é justamente a característica procurada pela função de soma dos quadrados das diferenças utilizadas no algoritmo.

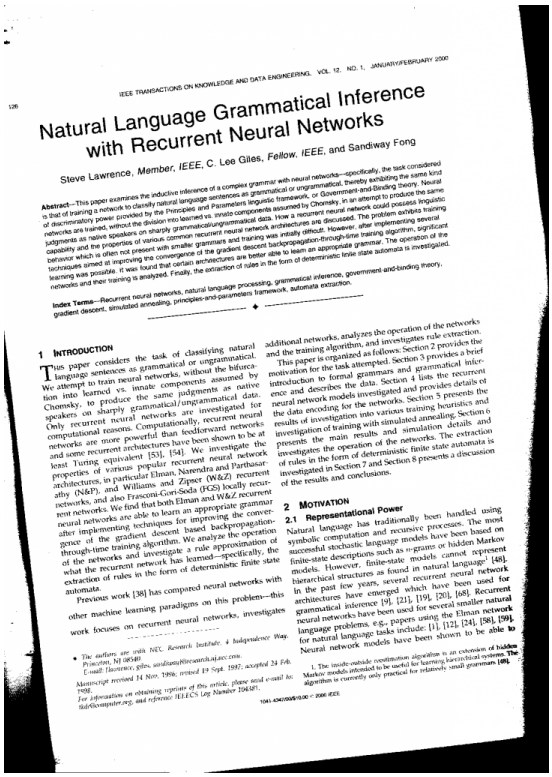
Realizando a análise de caracteres entre a imagem original e a alinhada (Figuras 2 e 4) com a utilização do Tesseract OCR obtemos o seguinte resultado:

Métrica	**Valor**
Caracteres reconhecidos antes	0
Caracteres reconhecidos depois	129
Diferença	+129

Tabela 1: Resumo da análise OCR antes e depois do alinhamento

Isso confirma as observações feitas anteriormente, onde o alinhamento havia sido considerado eficiente de fato.

Embora a imagem utilizada como exemplo seja relativamente simples, o algoritmo também funciona com imagens e digitalizações um pouco mais completas. Para demonstrar isso, será utilizada a seguir a imagem *sample2.png*.



IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 12, NO. 1, JANUARY/FEBRUARY 2000

Natural Language Grammatical Inference with Recurrent Neural Networks

Steve Lawrence, *Member, IEEE*, C. Lee Giles, *Fellow, IEEE*, and Sandiway Fong

Abstract—This paper examines the inductive inference of a complex grammar with neural networks—specifically, the task considered is that of training a network to classify natural language sentences as grammatical or ungrammatical, thereby exhibiting the same kind of discriminatory power provided by the Principles and Parameters linguistic framework, or Government-and-Binding theory. Neural networks are trained, without the division into learned vs. innate components assumed by Chomsky, in an attempt to produce the same judgments as native speakers on sharply grammatical/ungrammatical data. How a recurrent neural network could possess linguistic capability and the properties of various common recurrent neural network architectures are discussed. The problem exhibits training behavior which is often not present with smaller grammars and training was initially difficult. However, after implementing several techniques aimed at improving the convergence of the gradient descent backpropagation-through-time training algorithm, significant learning was possible. It was found that certain architectures are better able to learn an appropriate grammar. The operation of the networks and their training is analyzed. Finally, the extraction of rules in the form of deterministic finite state automata is investigated.

Index Terms—Recurrent neural networks, natural language processing, grammatical inference, government-and-binding theory, gradient descent, simulated annealing, principles-and-parameters framework, automata extraction.

1 Introduction

Thus paper considers the task of classifying natural language sentences as grammatical or ungrammatical. We attempt to train neural networks, without the bifurcation into learned vs. innate components assumed by Chomsky, to produce the same judgments as native speakers on sharply grammatical/ungrammatical data. Only recurrent neural networks are investigated for computational reasons. Computationally, recurrent neural networks are more powerful than feedforward networks and some recurrent architectures have been shown to be at least Turing equivalent [53], [54]. We investigate the properties of various popular recurrent neural network architectures, in particular Elman, Narendra and Parthasarathy (N&P), and Williams and Zipser (W&Z) recurrent networks, and also Frasconi-Gori-Soda (FGS) locally recurrent networks. We find that both Elman and W&Z recurrent neural networks are able to learn an appropriate grammar after implementing techniques for improving the convergence of the gradient descent based backpropagation-through-time training algorithm. We analyze the operation of the networks and investigate a rule approximation of what the recurrent network has learned—specifically, the extraction of rules in the form of deterministic finite state automata.

Previous work [38] has compared neural networks with other machine learning paradigms on this problem—this work focuses on recurrent neural networks, investigates additional networks, analyzes the operation of the networks and the training algorithm, and investigates rule extraction.

This paper is organized as follows: Section 2 provides the motivation for the task attempted. Section 3 provides a brief introduction to formal grammars and grammatical inference and describes the data. Section 4 lists the recurrent neural network models investigated and provides details of the data encoding for the networks. Section 5 presents the results of investigation into various training heuristics and investigation of training with simulated annealing. Section 6 presents the main results and simulation details and investigates the operation of the networks. The extraction of rules in the form of deterministic finite state automata is investigated in Section 7 and Section 8 presents a discussion of the results and conclusions.

2 Motivation

2.1 Representational Power

Natural language has traditionally been handled using symbolic computation and recursive processes. The most successful stochastic language models have been based on finite-state descriptions such as n-grams or hidden Markov models. However, finite-state models cannot represent hierarchical structures as found in natural language[1] [48]. In the past few years, several recurrent neural network architectures have emerged which have been used for grammatical inference [9], [21], [19], [20], [68]. Recurrent neural networks have been used for several smaller natural language problems, e.g., papers using the Elman network for natural language tasks include: [1], [12], [24], [58], [59]. Neural network models have been shown to be able to

• The authors are with NEC Research Institute, 4 Independence Way, Princeton, NJ 08540.
E-mail: {lawrence, giles, sandiway}@research.nj.nec.com.

Manuscript received 14 Nov. 1996; revised 19 Sept. 1997; accepted 24 Feb. 1998.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104581.

1. The inside-outside reestimation algorithm is an extension of hidden Markov models intended to be useful for learning hierarchical systems. The algorithm is currently only practical for relatively small grammars [39].

1041-4347/00/$10.00 © 2000 IEEE

Figura 7: sample2.png

Os resultados são:

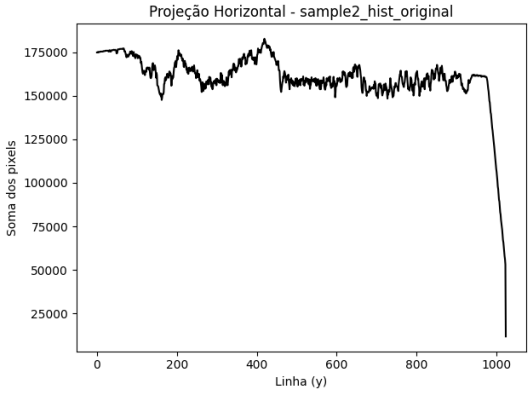
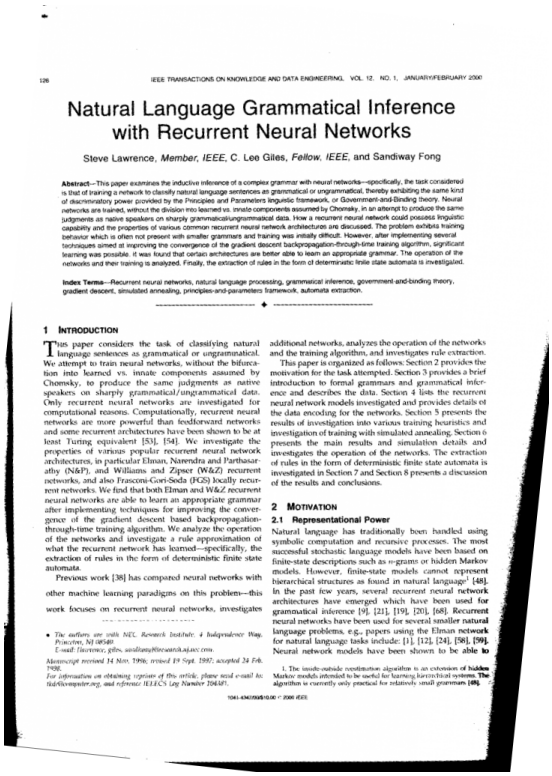


Figura 8: histograma de projeção da imagem original



IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 12, NO. 1, JANUARY/FEBRUARY 2000

Natural Language Grammatical Inference with Recurrent Neural Networks

Steve Lawrence, *Member, IEEE*, C. Lee Giles, *Fellow, IEEE*, and Sandiway Fong

Abstract—This paper examines the inductive inference of a complex grammar with neural networks—specifically, the task considered is that of training a network to classify natural language sentences as grammatical or ungrammatical, thereby exhibiting the same kind of discriminatory power provided by the Principles and Parameters linguistic framework, or Government-and-Binding theory. Neural networks are trained, without the division into learned vs. innate components assumed by Chomsky, in an attempt to produce the same judgments as native speakers on sharply grammatical/ungrammatical data. How a recurrent neural network could possess linguistic capability and the properties of various common recurrent neural network architectures are discussed. The problem exhibits training behavior which is often not present with smaller grammars and training was initially difficult. However, after implementing several techniques aimed at improving the convergence of the gradient descent backpropagation-through-time training algorithm, significant learning was possible. It was found that certain architectures are better able to learn an appropriate grammar. The operation of the networks and their training is analyzed. Finally, the extraction of rules in the form of deterministic finite state automata is investigated.

Index Terms—Recurrent neural networks, natural language processing, grammatical inference, government-and-binding theory, gradient descent, simulated annealing, principles-and-parameters framework, automata extraction.

1 Introduction

Thus paper considers the task of classifying natural language sentences as grammatical or ungrammatical. We attempt to train neural networks, without the bifurcation into learned vs. innate components assumed by Chomsky, to produce the same judgments as native speakers on sharply grammatical/ungrammatical data. Only recurrent neural networks are investigated for computational reasons. Computationally, recurrent neural networks are more powerful than feedforward networks and some recurrent architectures have been shown to be at least Turing equivalent [53], [54]. We investigate the properties of various popular recurrent neural network architectures, in particular Elman, Narendra and Parthasarathy (N&P), and Williams and Zipser (W&Z) recurrent networks, and also Frasconi-Gori-Soda (FGS) locally recurrent networks. We find that both Elman and W&Z recurrent neural networks are able to learn an appropriate grammar after implementing techniques for improving the convergence of the gradient descent based backpropagation-through-time training algorithm. We analyze the operation of the networks and investigate a rule approximation of what the recurrent network has learned—specifically, the extraction of rules in the form of deterministic finite state automata.

Previous work [38] has compared neural networks with other machine learning paradigms on this problem—this work focuses on recurrent neural networks, investigates additional networks, analyzes the operation of the networks and the training algorithm, and investigates rule extraction.

This paper is organized as follows: Section 2 provides the motivation for the task attempted. Section 3 provides a brief introduction to formal grammars and grammatical inference and describes the data. Section 4 lists the recurrent neural network models investigated and provides details of the data encoding for the networks. Section 5 presents the results of investigation into various training heuristics and investigation of training with simulated annealing. Section 6 presents the main results and simulation details and investigates the operation of the networks. The extraction of rules in the form of deterministic finite state automata is investigated in Section 7 and Section 8 presents a discussion of the results and conclusions.

2 Motivation

2.1 Representational Power

Natural language has traditionally been handled using symbolic computation and recursive processes. The most successful stochastic language models have been based on finite-state descriptions such as n-grams or hidden Markov models. However, finite-state models cannot represent hierarchical structures as found in natural language[1] [48]. In the past few years, several recurrent neural network architectures have emerged which have been used for grammatical inference [9], [21], [19], [20], [68]. Recurrent neural networks have been used for several smaller natural language problems, e.g., papers using the Elman network for natural language tasks include: [1], [12], [24], [58], [59]. Neural network models have been shown to be able to

1. The inside-outside contimation algorithm is an extension of hidden Markov models intended to be useful for learning hierarchical systems. The algorithm is currently only practical for relatively small grammars [48].

• The authors are with NEC Research Institute, 4 Independence Way, Princeton, NJ 08540.
E-mail: {lawrence, giles, sandiway}@research.nj.nec.com.

Manuscript received 14 Nov. 1996; revised 19 Sept. 1997; accepted 24 Feb. 1998.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104649.

Figura 9: sample2.png alinhada

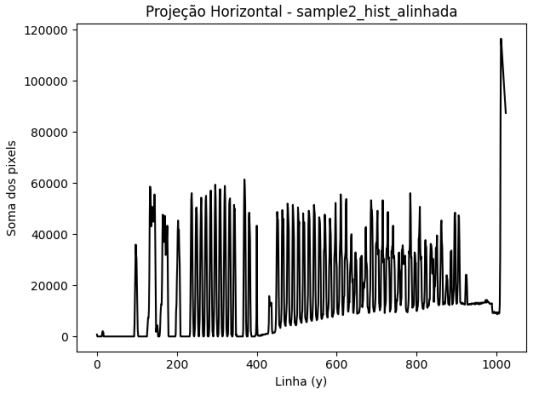
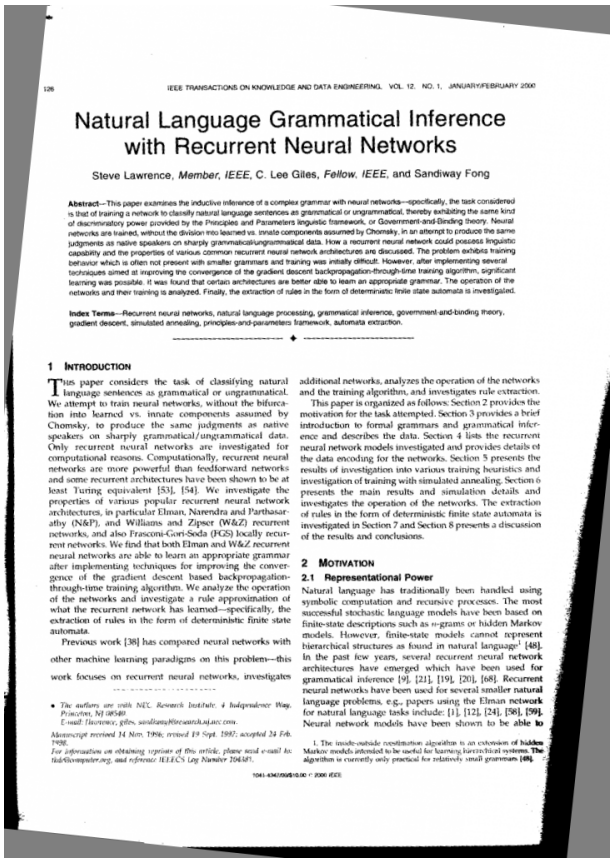


Figura 10: histograma de projeção da imagem alinhada

Vendo os resultados acima, é possível observar que após rotacionar a imagem original, alguns espaços vazios são introduzidos nos quatro cantos da imagem, e, assim como explicado anteriormente, essas partes são preenchidas por pixels brancos. Para ilustrar melhor, a mesma imagem será gerada, mas agora as lacunas serão preenchidas por pixels cinzas de valor 127.

```
cv2.warpAffine(imagem, matriz_rot, (w, h), flags=cv2.INTER_LINEAR, borderValue=127)
```

O resultado pode ser visto na Figura 11.



126 IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 12, NO. 1, JANUARY/FEBRUARY 2000

Natural Language Grammatical Inference with Recurrent Neural Networks

Steve Lawrence, *Member, IEEE*, C. Lee Giles, *Fellow, IEEE*, and Sandiway Fong

Abstract—This paper examines the inductive inference of a complex grammar with neural networks—specifically, the task considered is that of training a network to classify natural language sentences as grammatical or ungrammatical, thereby exhibiting the same kind of discriminatory power provided by the Principles and Parameters linguistic framework, or Government-and-Binding theory. Neural networks are trained, without the division into learned vs. innate components assumed by Chomsky, in an attempt to produce the same judgments as native speakers on sharply grammatical/ungrammatical data. How a recurrent neural network could possess linguistic capability and the properties of various common recurrent neural network architectures are discussed. The problem exhibits training behavior which is often not present with smaller grammars and training was initially difficult. However, after implementing several techniques aimed at improving the convergence of the gradient descent backpropagation-through-time training algorithm, significant learning was possible. It was found that certain architectures are better able to learn an appropriate grammar. The operation of the networks and their training is analyzed. Finally, the extraction of rules in the form of deterministic finite state automata is investigated.

Index Terms—Recurrent neural networks, natural language processing, grammatical inference, government-and-binding theory, gradient descent, simulated annealing, principles-and-parameters framework, automata extraction.

1 Introduction

This paper considers the task of classifying natural language sentences as grammatical or ungrammatical. We attempt to train neural networks, without the bifurcation into learned vs. innate components assumed by Chomsky, to produce the same judgments as native speakers on sharply grammatical/ungrammatical data. Only recurrent neural networks are investigated for computational reasons. Computationally, recurrent neural networks are more powerful than feedforward networks and some recurrent architectures have been shown to be at least Turing equivalent [53], [54]. We investigate the properties of various popular recurrent neural network architectures, in particular Elman, Narendra and Parthasarathy (N&P), and Williams and Zipser (W&Z) recurrent networks, and also Frasconi-Gori-Soda (FGS) locally recurrent networks. We find that both Elman and W&Z recurrent neural networks are able to learn an appropriate grammar after implementing techniques for improving the convergence of the gradient descent based backpropagation-through-time training algorithm. We analyze the operation of the networks and investigate a rule approximation of what the recurrent network has learned—specifically, the extraction of rules in the form of deterministic finite state automata.

Previous work [38] has compared neural networks with other machine learning paradigms on this problem—this work focuses on recurrent neural networks, investigates additional networks, analyzes the operation of the networks and the training algorithm, and investigates rule extraction.

This paper is organized as follows. Section 2 provides the motivation for the task attempted. Section 3 provides a brief introduction to formal grammars and grammatical inference and describes the data. Section 4 lists the recurrent neural network models investigated and provides details of the data encoding for the networks. Section 5 presents the results of investigation into various training heuristics and investigation of training with simulated annealing. Section 6 presents the main results and simulation details and investigates the operation of the networks. The extraction of rules in the form of deterministic finite state automata is investigated in Section 7 and Section 8 presents a discussion of the results and conclusions.

2 Motivation

2.1 Representational Power

Natural language has traditionally been handled using symbolic computation and recursive processes. The most successful stochastic language models have been based on finite-state descriptions such as *n*-grams or hidden Markov models. However, finite-state models cannot represent hierarchical structures as found in natural language[1] [48]. In the past few years, several recurrent neural network architectures have emerged which have been used for grammatical inference [9], [21], [19], [20], [68]. Recurrent neural networks have been used for several smaller natural language problems, e.g., papers using the Elman network for natural language tasks include: [1], [12], [24], [58], [59]. Neural network models have been shown to be able to

• The authors are with NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. E-mail: {lawrence, giles, sandiway}@research.nj.nec.com.

Manuscript received 14 Nov. 1996; revised 19 Sept. 1997; accepted 24 Feb. 1998.

For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104681.

1. The inside-outside reestimation algorithm is an extension of hidden Markov models intended to be useful for learning hierarchical systems. The algorithm is currently only practical for relatively small grammars [48].

1041-4347/00/$10.00 © 2000 IEEE

Figura 11: sample2.png alinhada com preenchimento cinza

Dessa forma, fica fácil notar os pixels que foram introduzidos pelo algoritmo para preencher a figura após a rotação.

3.2 Técnica baseada na Transformada de Hough

O objetivo da segunda tarefa proposta foi utilizar uma técnica de detecção e correção de inclinação baseada na Transformada de Hough. Essa técnica consiste na aplicação de Sobel para detecção de bordas e em seguida na aplicação da transformada de Hough para detecção de linhas. Com as linhas detectadas, a média de suas inclinações é calculada (ângulo de desalinhamento), fornecendo um valor de ângulo para a inclinação da imagem. Em seguida, a imagem é rotacionada utilizando o valor obtido, gerando como resultado uma imagem alinhada. Dessa vez, a imagem utilizada será *pos_41.png* (Figura 12).

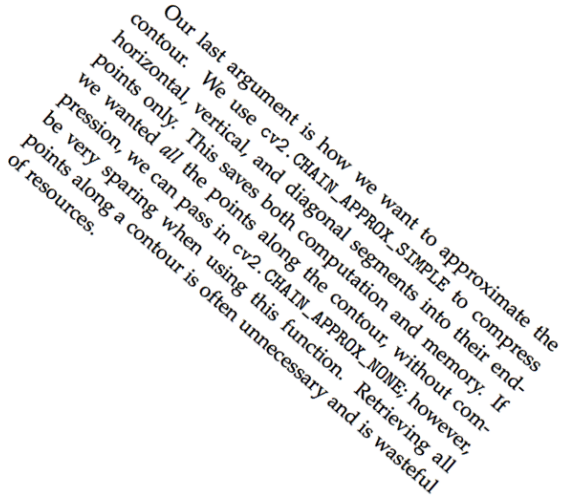


Figura 12: pos_41.png

O primeiro passo, assim como no caso anterior, é a binarização da imagem, que ocorre da exata mesma forma (Figura 13).

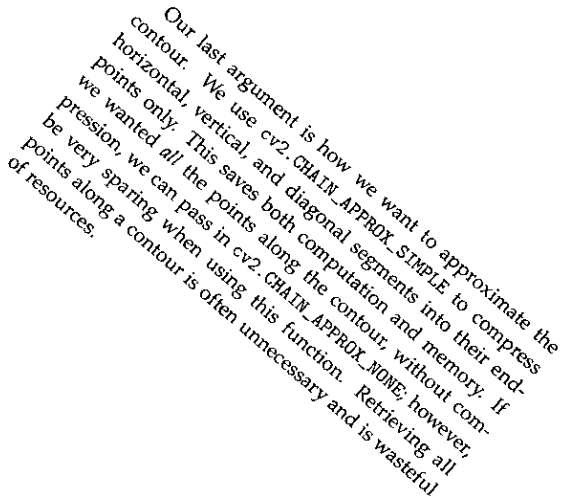


Figura 13: pos_41.png binarizada

Após a binarização da imagem, é realizada a aplicção do *filtro de Sobel* na imagem binarizada, utilizando o operador na direção horizontal (derivada na direção X). O objetivo é destacar as bordas verticais da imagem, isto é, as transições abruptas entre fundo e texto, que são comuns em documentos digitalizados. A aplicação do Sobel é feita com o trecho de código:

```
bordas = cv2.Sobel(imagem_bin, cv2.CV_8U, 1, 0, ksize=3)
```

O resultado da operação pode ser visualizadoa seguir na Figura 14. Essa imagem representa a intensidade das bordas verticais presentes no documento, que servirão de entrada para a próxima etapa.

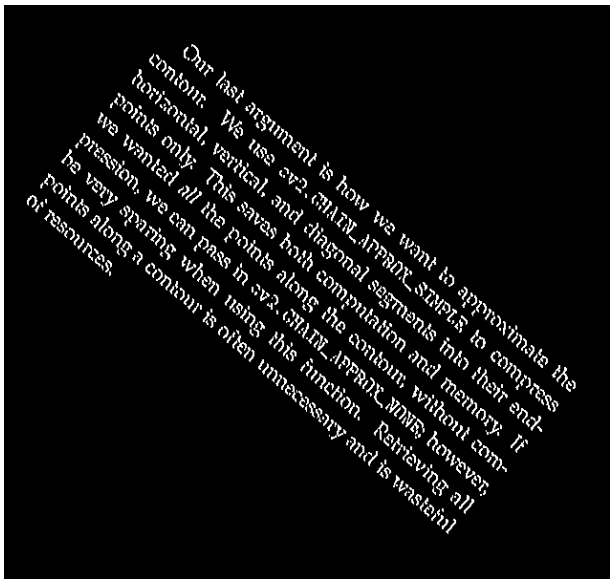


Figura 14: Detecção de bordas com filtro de Sobel na imagem pos_41.png

A seguir, a transformada de Hough é aplicada sobre a imagem de bordas, utilizando o seguinte trecho de código:

```
linhas = cv2.HoughLines(bordas, 1, np.pi / 180, threshold=100)
```

No trecho de código acima, os parâmetros possuem grande importância:

- O primeiro parâmetro (1) representa a resolução de ρ, isto é, a distância radial será quantizada em incrementos de 1 pixel.
- O segundo parâmetro ($\pi/180$) define a resolução angular θ em radianos, o que equivale a 1 grau por incremento.
- O parâmetro *threshold* determina o número mínimo de interseções no espaço de Hough para que uma linha seja detectada. Um valor de 100 foi escolhido empiricamente para garantir que apenas as linhas mais relevantes (como as linhas de texto) sejam detectadas, evitando ruídos.

A transformada de Hough retorna uma lista de linhas detectadas, cada uma representada por um par (ρ, θ). Esses valores indicam a posição e a orientação da linha no espaço polar. Como o objetivo é detectar a inclinação horizntal do texto, os valores de θ são convertidos para graus e normalizados em torno de 0^{o} utilizando a fórmula:

```
angulo = (theta * 180 / np.pi) - 90
```

Em seguida, todos os ângulos detectados são armazenados em uma lista. Para obter o valor mais representativo da inclinação da imagem, a mediana dos ângulos é utilizada:

```
return np.median(angulos)
```

Essa escolha é motivada pela robustez da mediana contra outliers, o que torna o método mais confiável mesmo em casos onde algumas linhas com valores extremos são detectadas.

Para efeito de visualização, também foi gerada uma imagem contendo as linhas detectadas pela transformada de Hough sobrepostas à imagem de bordas. Isso foi feito utilizando a função *cv2.line()*, com o seguinte resultado:

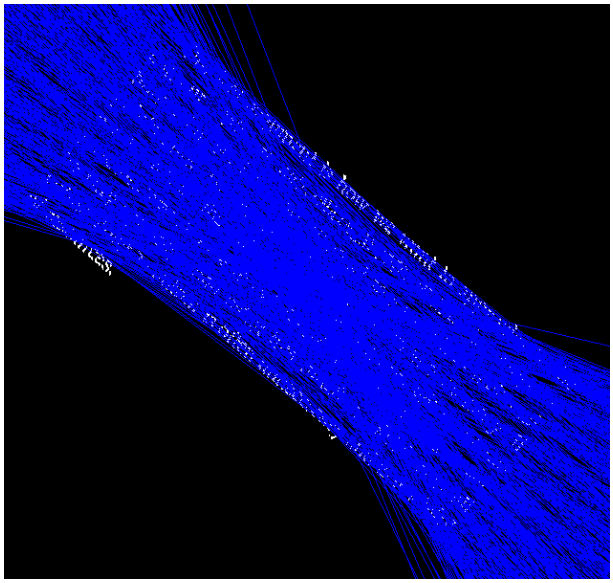


Figura 15: Linhas detectadas pela transformada de Hough sobre as bordas de pos_41.png

Após o cálculo da inclinação média, o ângulo é utilizado para rotacionar a imagem original. Assim como na técnica anterior, a rotação é feita com:

```
matriz_rot = cv2.getRotationMatrix2D((w // 2, h // 2), angulo, 1.0)
cv2.warpAffine(imagem, matriz_rot, (w, h), flags=cv2.INTER_LINEAR, borderValue=255)
```

Note que, assim como na técnica anterior, o parâmetro *borderValue* é 255, ou seja, as lacunas (espaços de pixels faltantes) da imagem resultante após a rotação serão preenchidas com pixels de valor 255 (brancos).

A Figura 16 mostra o resultado final da imagem *pos_41.png* alinhada com a técnica de Hough, enquanto as Figuras 17 e 18 apresentam seus histogramas de projeção horizontal originais e após o alinhamento.

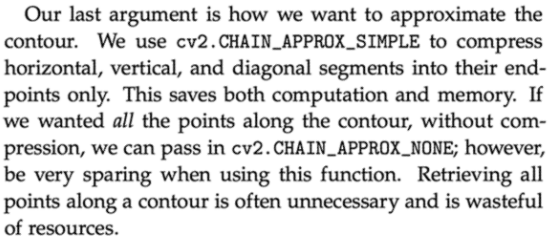

Our last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their endpoints only. This saves both computation and memory. If we wanted *all* the points along the contour, without compression, we can pass in cv2.CHAIN_APPROX_NONE; however, be very sparing when using this function. Retrieving all points along a contour is often unnecessary and is wasteful of resources.

Figura 16: pos_41.png alinhada com Hough

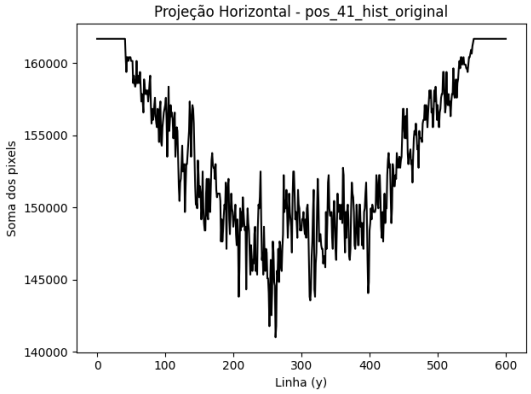


Figura 17: histograma de projeção original

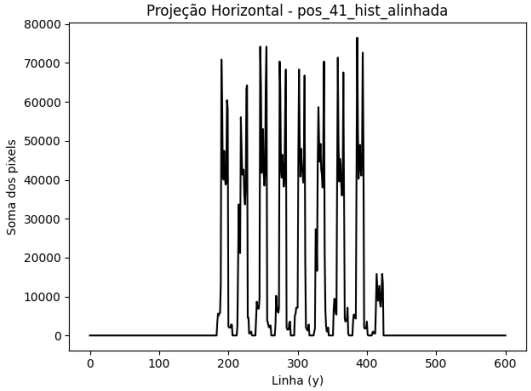


Figura 18: histograma de projeção após alinhamento

Um dos parâmetros mais sensíveis da função *cv2.HoughLines()* é o *threshold*, que representa o número mínimo de interseções no espaço de Hough necessárias para que uma linha seja considerada válida. Esse valor afeta diretamente a quantidade e a qualidade das linhas detectadas, e, portanto, tem impacto direto no resultado da inclinação estimada.

Valores mais altos para o *threshold* (como o valor 100 utilizado) tornam o critério de aceitação de uma linha mais rígido. Isso significa que apenas linhas muito bem definidas e com alta consistência no espaço de Hough serão consideradas. Na prática, isso tende a remover ruídos e linhas espúrias, preservando apenas linhas dominantes (o que é desejável em imagens de documentos bem digitalizados) com texto claro e contrastante. No entanto, esse comportamento também pode fazer com que a função não detecte nenhuma linha em imagens com menos contraste, baixa resolução, ruído ou desalinhamento severo, prejudicando a detecção da inclinação.

Por outro lado, ao utilizar valores menores de *threshold*, mais linhas serão detectadas, incluindo bordas menos evidentes ou até mesmo artefatos indesejados (ruídos). Isso pode

ser útil em casos onde as linhas de texto estão fragmentadas, pouco definidas ou parcialmente visíveis. No entanto, essa abordagem aumenta o risco de detectar linhas falsas que não correspondem à estrutura real do documento, o que pode gerar um valor incorreto de inclinação média. Como consequência, a rotação aplicada pode piorar o alinhamento ao invés de corrigi-lo.

A seguir, estão alguns exemplos de *pos_41.png* alinhada com threshold 10 e 300.

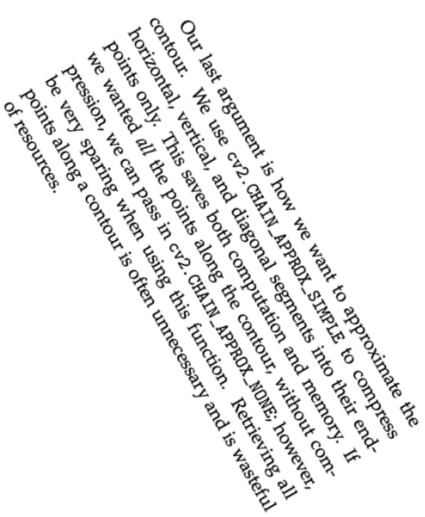


Figura 19: pos_41.png alinhada com threshold 10

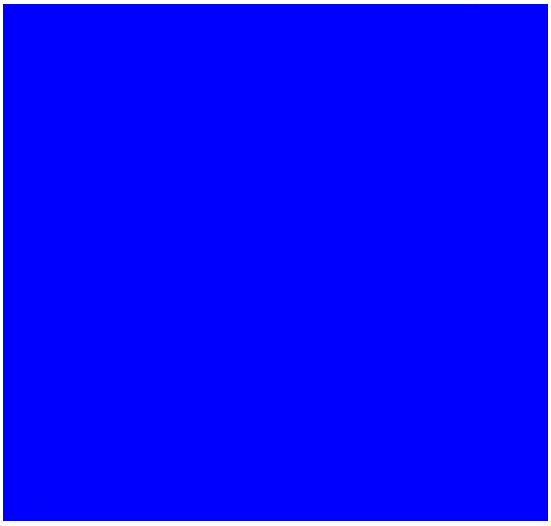


Figura 20: Linhas detectadas pela transformada de Hough com threshold 10

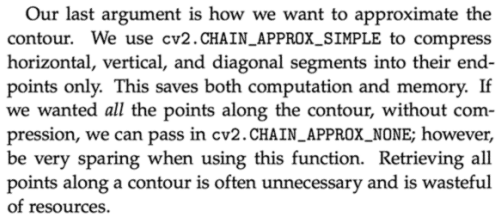


Figura 21: pos_41.png alinhada com threshold 300

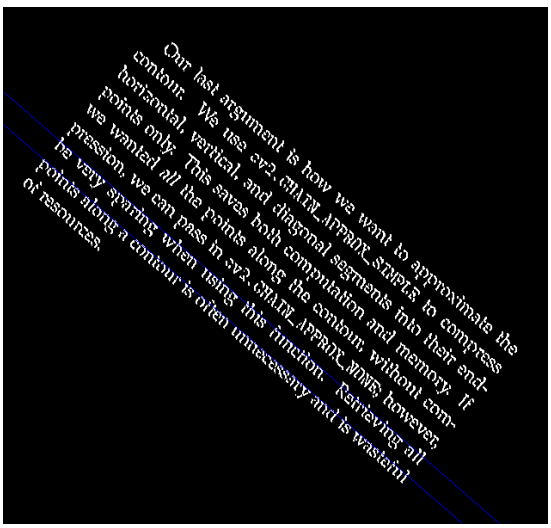


Figura 22: Linhas detectadas pela transformada de Hough com threshold 300

Como é possível ver acima, um threshold baixo fez com que muitas linhas fossem detectadas (a imagem mostrando as linhas foi tomada completamente pelas inúmeras linhas detectadas), e isso fez com que o alinhamento fosse completamente incorreto, tornando o resultado pior do que a imagem original. Já o threshold mais alto, nesse caso, não causou

problemas visíveis, porém, é possível ver que apenas 2 linhas foram reconhecidas na imagem, ou seja, caso o threshold fosse um pouco mais alto, nenhuma linha seria reconhecida e nenhuma rotação seria aplicada, mantendo a imagem final igual à original.

Dessa forma, a escolha do valor ideal de *threshold* depende diretamente das características da imagem de entrada. Um valor intermediário, como 100, foi escolhido neste trabalho após testes empíricos, visando um equilíbrio entre rigidez e sensibilidade. Para garantir maior robustez em aplicações mais gerais, uma possível melhoria seria adaptar esse valor dinamicamente com base nas características da imagem ou aplicar uma pré-avaliação da quantidade de linhas detectadas para ajustar o *threshold* automaticamente.

Porém, a escolha empírica do valor de *threshold* gera um problema: para algumas imagens, o alinhamento simplesmente não funciona. Para mostrar isso, a seguir, estão os resultados do alinhamento com Transformada de Hough e *threshold* 100 aplicaods na imagem *partitura.png*:

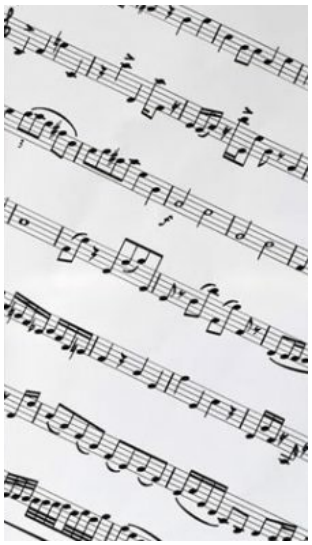


Figura 23: partitura.png

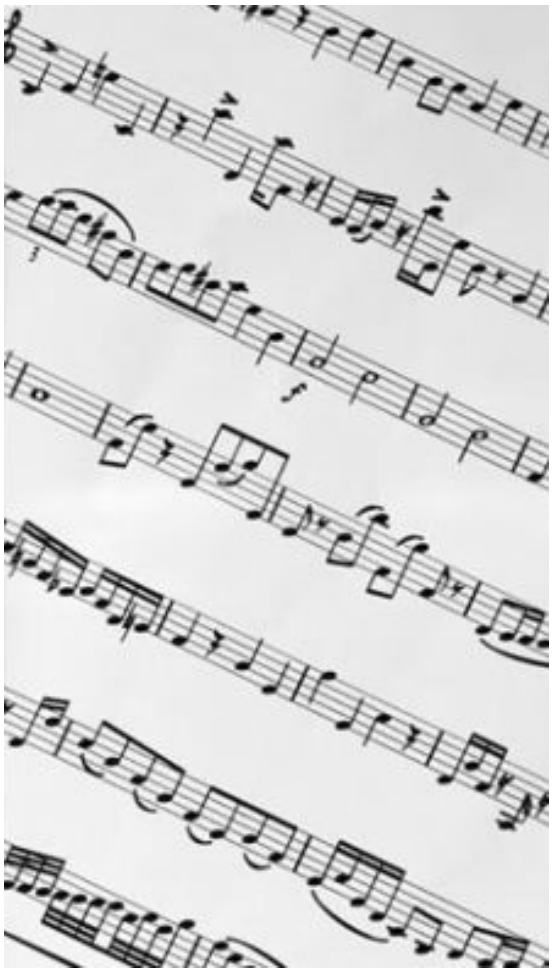


Figura 24: partitura.png alinhada com Hough

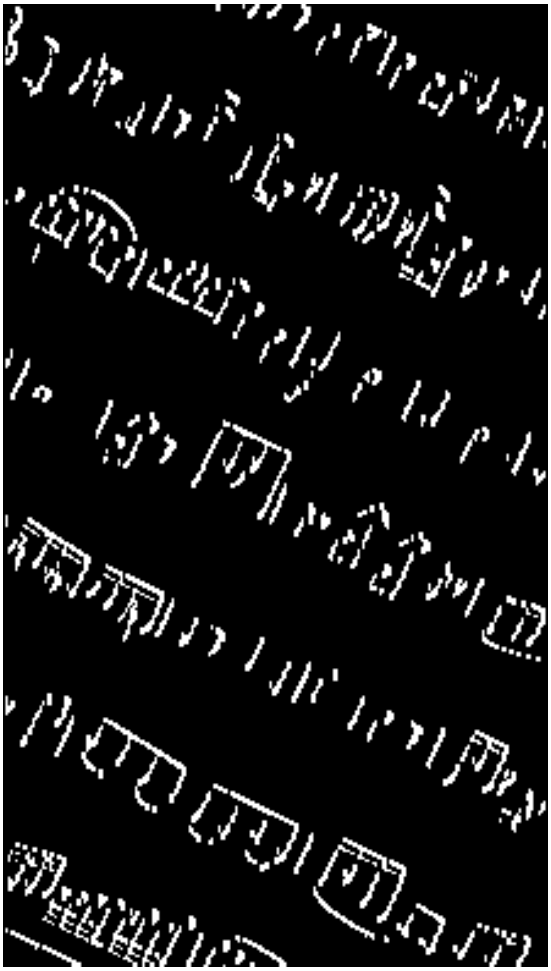


Figura 25: linhas detectadas

Como é possível observar, nenhum ângulo foi aplicado no alinhamento pois nenhuma linha foi detectada. Isso se deve ao valor de *threshold* não ser compatível com o contraste e estutura geral da imagem.

Por causa da importância do parâmetro *threshold*, uma solução desenvolvida para melhorar e dinamizar a escolha desse parâmetro foi criar uma função que, com base na imagem de bordas, calcula estatísticas relevantes e utiliza essas informações para ajustar o valor de *threshold* de forma mais refinada. A função considera tanto a densidade de bordas presentes na imagem quanto a variação de intensidade (contraste) dessas bordas, medidas respectivamente pela proporção de pixels diferentes de zero e pelo desvio padrão da imagem de bordas.

A densidade de bordas é útil para estimar o quanto de estrutura (linhas, traços, contornos) está presente na imagem: imagens com poucas bordas geralmente exigem um *threshold* mais baixo, enquanto imagens com muitas bordas devem usar um valor mais alto para evitar sobrecarga de detecções. Por outro lado, o desvio padrão das intensidades da imagem de bordas é uma boa estimativa do contraste local: um desvio alto indica bordas bem definidas e permite o uso de um *threshold* maior; um desvio baixo sugere que as bordas são fracas ou mal definidas, exigindo um *threshold* menor para não ignorá-las.

A função combina esses dois fatores de forma ponderada para calcular um ajuste

multiplicativo que será aplicado a um valor base (como 100), com os valores finais sendo limitados a um intervalo seguro, como entre 30 e 200. A função utilizada é a seguinte:

```
def calcular_threshold_dinamico(bordas, base=100, min_th=30, max_th=200):
    total_pixels = bordas.shape[0] * bordas.shape[1]
    qtd_bordas = np.count_nonzero(bordas)
    densidade = qtd_bordas / total_pixels

    media = np.mean(bordas)
    desvio = np.std(bordas)

    peso_densidade = (1 - densidade)
    peso_contraste = desvio / 128

    ajuste = (peso_densidade * 0.6 + peso_contraste * 0.4)
    threshold = int(base * ajuste)
    threshold = max(min_th, min(max_th, threshold))

    return threshold
```

Dessa forma, o valor do *threshold* se adapta suavemente de acordo com as condições reais da imagem de entrada. Isso evita o uso de valores arbitrários e contribui para tornar o método mais robusto, generalizável e confiável, mesmo diante de variações em qualidade de digitalização, contraste ou presença de ruído.

Para mostrar a eficácia da nova forma de determinação de *threshold*, o alinhamento será aplicado novamente em *partitura.png*.

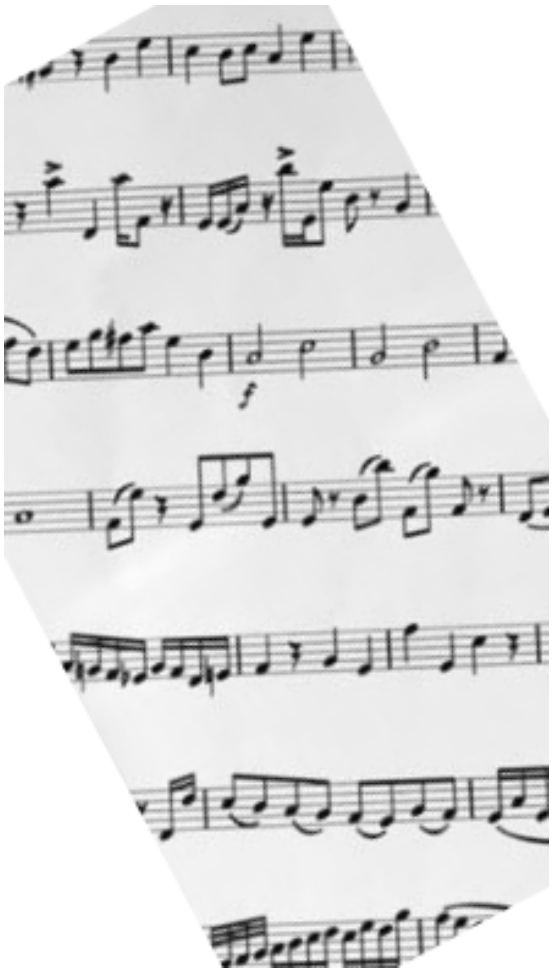


Figura 26: partitura.png alinhada com Hough usando threshold dinâmico

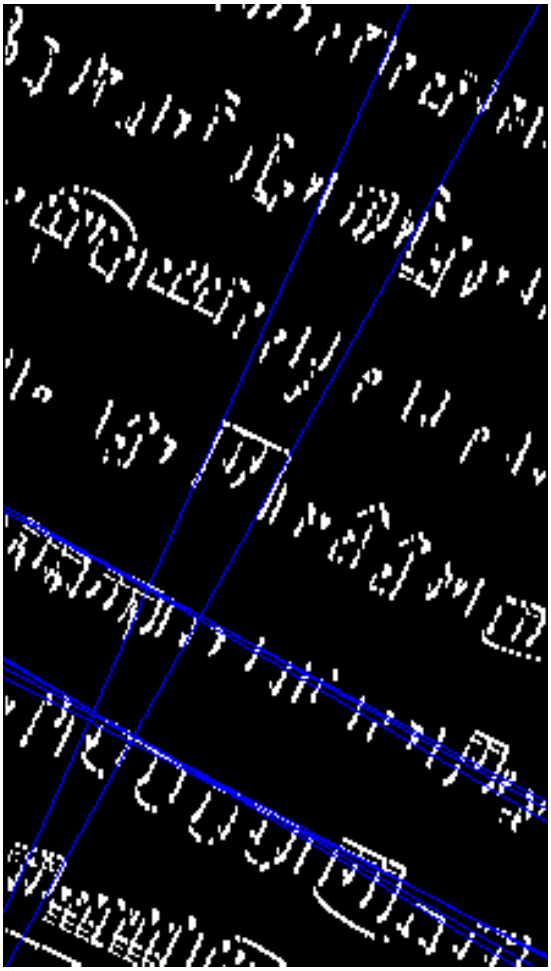


Figura 27: linhas detectadas

Com isso, é possível ver que o alinhamento foi eficaz, e linhas foram detectadas pois o valor de *threshold* dessa vez foi escolhido com base na imagem de entrada, e não de forma arbitrária.

O método, porém, não é perfeito, por exemplo, ao reaplicar *pos_41.png* utilizando a nova abordagem, temos:

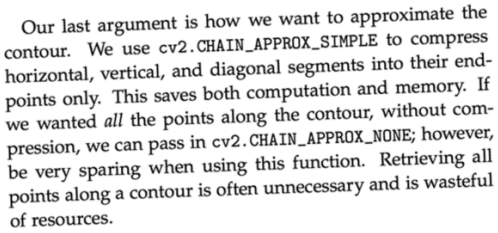


Figura 28: pos_41.png alinhada com Hough

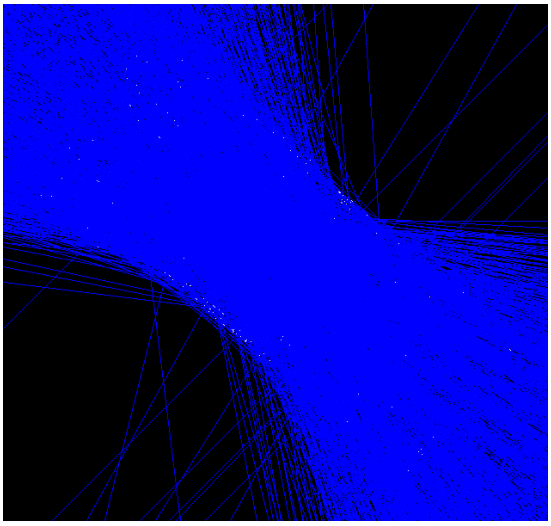


Figura 29: linhas detectadas

Como pode ser visto acima, o resultado alinhando *pos_41.png* com a nova técnica de seleção de *threshold* gerou um resultado ligeiramente menos alinhado que o resultado inicial com *threshold* 100. Isso indica que a função utilizada pode ser refinada, para que os resultados de *threshold* obtidos por ela sejam ainda melhores (alterando os pesos por exemplo).

Para efeito de comparação, a técnica com a Transformada de Hough e *threshold* 100 também foi aplicada na imagem *sample1.png* (a mesma utilizada na técnica baseada em projeção horizontal), os resultados (imagem e histograma) podem ser vistos a seguir nas Figuras 32 e 33 mostram os respectivos resultados visuais e analíticos.

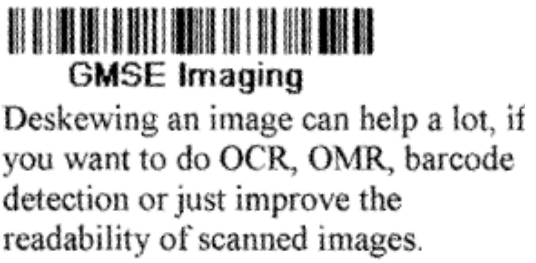


Figura 30: sample1.png alinhada com Hough e threshold 100

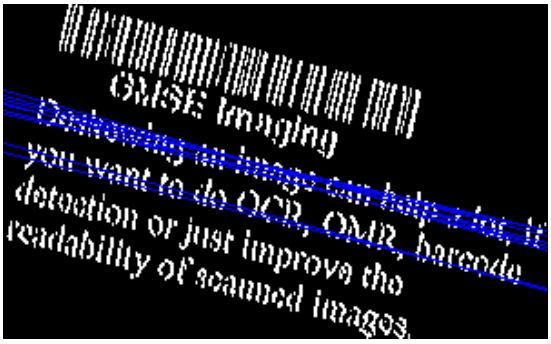


Figura 31: linhas detectadas

Com base nos resultados observados, é possível concluir que a técnica baseada em transformada de Hough é eficaz na detecção de inclinação, especialmente em imagens com texto bem definido e contrastante. Entretanto, por depender da detecção explícita de linhas, a performance pode variar em documentos com ruído, baixa resolução ou pouca definição nas bordas.

Realizando a análise de *sample1.png* alinhada com Hough com o Tesseract OCR, obtemos:

Métrica	Valor
Caracteres reconhecidos antes	0
Caracteres reconhecidos depois	131
Diferença	+131

Tabela 2: Resumo da análise OCR antes e depois do alinhamento

Observando o resultado acima, mais uma vez podemos notar a eficiência do alinhamento, e dessa vez, comparando com o alinhamento da mesma imagem pela técnica baseada em projeção horizontal, é possível notar que mais caracteres foram reconhecidos, o que sugere que a técnica por Transformada de Hough apresenta resultados ligeiramente melhores.

Por fim, a mesma imagem será alinhada utilizando a técnica baseada na Transformada de Hough, mas utilizando a função de seleção dinâmica de *threshold*. Os resultados podem ser vistos a seguir:

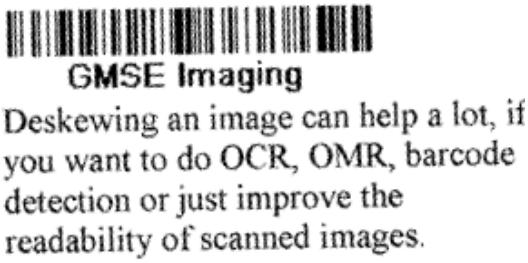


Figura 32: sample1.png alinhada com Hough e threshold dinâmico

Figura 33: linhas detectadas

Realizando a análise de *sample1.png* alinhada com Hough com o Tesseract OCR, obtemos:

Métrica	Valor
Caracteres reconhecidos antes	0
Caracteres reconhecidos depois	122
Diferença	+122

Tabela 3: Resumo da análise OCR antes e depois do alinhamento

Comparando os resultados das análise com Tesseract OCR entre os três alinhamentos (projeção, Hough com threshold 100 e Hough com thresolh dinâmico) é possível observar que a técnica utilizando seleção de *threshold* dinâmica é ligeriamente menos eficaz no que diz respeito a precisão do alinhamento, visto que teve apenas 122 caracteres reconhecidos, enquanto o alinhamento por projeção teve 129, e o por Hough com *threshold* 100 teve 131 caracteres reconhecidos. Porém, a grande vantagem da técnica que utiliza seleção de *threshold* dinâmica está no fato de que ela funciona com imagens que outrora sequer seriam alinhadas com as outras técnicas, como o exemplo dado com *partitura.png*.