Introdução ao Processamento Digital de Imagem

Trabalho 03

Filipe Alves Sampaio
RA 249092

I. Introdução

Este trabalho tem como objetivo explicar o funcionamento, implementação e resultados do trabalho 03, desenvolvido na disciplina MO443. O trabalho tem como objetivo implementar dois algoritmos para detecção e correção de inclinação de documentos, um baseado em projeção horizontal e outro baseado na transformada de hough.

A seção II inclui exemplos de execução do programa, bem como os parâmetros necessários para sua execução e uma descrição, tanto dos parâmetros de entrada como das saídas geradas pelo script.

II. Execução

Dentro do zip referente ao trabalho, encontra-se o arquivo: *alinhar.py*. Esse arquivo, escrito em Python, foi desenvolvido para realizar as tarefas especificadas na descrição do trabalho. Nessa seção apresenta-se os seguintes tópicos:

A. Dependências necessárias para a execução do trabalho.

É opcional, mas aconselha-se utilizar um ambiente de desenvolvimento para isolar as bibliotecas utilizadas nesse trabalho.

Assim, como orientado pelo professor da disciplina, utilize o Anaconda ou semelhante e crie um ambiente separado.

Após isso, instale as seguintes dependências:

- **Python**: Foi utilizado a versão 3.12.2. Essa versão é opcional, dado que os scripts desenvolvidos são simples e não utilizam algoritmos sofisticados que não existam em versões anteriores do Python.
- **Numpy**: Usada para manipular vetores e otimizar o encode da esteganografia.
- **Sys**: Usada para manipular argumentos passados por linha de comando.
- **Opencv**: Biblioteca usada para manipular imagens.
- **Matplotlib**: Biblioteca usada para gerar histogramas das imagens.
- **Pytesseract**: é um software de reconhecimento ótico de caracteres de código aberto.

B. Funcionalidades implementadas e suas ações sobre as tarefas exigidas no trabalho.

Essa seção tem como finalidade explicar brevemente o funcionamento do que foi implementado. Uma análise mais profunda dos algoritmos usados será descrita na seção III e uma análise dos resultados será discutida na seção IV.

Como descrito pelo trabalho, o objetivo é implementar dois algoritmos para detectar o ângulo de rotação de um documento e realizar a correção para uma inclinação horizontal.

Para isso, é requisitada a implementação da projeção horizontal e da transformada de hough para verificar o desempenho de ambas em relação a

detecção e ajuste dos ângulos de inclinação do documento.

Após isso, deve ser verificado, utilizando um algoritmo de OCR (Optical Character Recognition), se é possível obter o texto do documento antes e após a correção de inclinação. Na descrição do trabalho é sugerido a utilização do Tesseract. Assim, foi utilizado esse algoritmo.

C. Parâmetros.

Neste tópico, é tratado os parâmetros possíveis para executar o *script* implementado. Seguindo o que foi informado pela descrição do trabalho, foi utilizado os seguintes parâmetros:

- **<imagem_entrada.png>**: imagem no formato PNG antes do alinhamento.
- **<modo>**: técnica utilizada no alinhamento da imagem (0: projeção horizontal e 1: transformada de hough).
- **<imagem_saida.png>**: imagem no formato PNG após o alinhamento.

Além desses parâmetros, é informado na seguinte lista, os arquivos que serão utilizados para execução do programa:

- **alinhar.py:** arquivo principal que irá executar o experimento de detecção e correção de alinhamento.

D. Exemplos de execução.

No diretório do programa há uma pasta, chamada: **/imgs**. Essa pasta contém as imagens de teste utilizadas na implementação do trabalho.

Ao finalizar a execução do *script* ***alinhar.py***, será gerado uma única imagem no diretório do projeto (resultante da correção do alinhamento). Durante a execução do programa, serão plotados alguns resultados do experimento, como gráficos de projeções horizontais em diferentes ângulos e imagens antes e após correção de inclinação.

O programa pode ser executado da seguinte forma:

python alinhar.py imagem_entrada.png modo imagem_saida.png

Para exemplificar, suponha que queira identificar o ângulo de inclinação da imagem **/imgs/neg_28.png**, com o modo **0** (ou seja, para usar o algoritmo de projeção horizontal) e com o nome da imagem de saída **imagem_final.png**. Segue comando:

python alinhar.py /imgs/neg_28.png 0 imagem_final.png

Ao executar esse comando acima, serão plotados: a imagem de entrada, um gráfico contendo todas as tentativas de ângulos na projeção horizontal e a imagem final corrigida.

Uma observação é que o gráfico de ângulos de tentativa é plotado apenas no modo 0, ou seja, somente quando se utiliza o algoritmo de projeção horizontal, pois esse algoritmo precisa realizar algumas rotações na imagem para que seja possível projetar os *pixels* pretos e calcular a função objetivo.

III. Implementação

Nessa seção serão descritos os passos e motivações que levaram ao estado final da implementação do programa. Juntamente, será feita uma descrição do funcionamento de cada etapa.

Abrindo o arquivo alinhar.py, será possível visualizar todas as funções implementadas para realizar a projeção horizontal e transformada de hough.

A. Código da projeção horizontal

Neste trabalho, utilizou-se uma implementação simplificada para realizar a projeção horizontal, seguindo o pseudocódigo sugerido na descrição do trabalho. Optou-se por utilizar já na primeira versão do algoritmo a biblioteca

Numpy para otimizar o código e evitar refatorações.

Conforme orientado no pseudocódigo, a função **projecao_horizontal()** espera uma imagem de entrada. Dentro dessa função é feito uma conversão da imagem para tons de cinza.

Na sequência, são inicializados dois vetores, um para as projeções, que armazena todas as projeções horizontais dos *pixels* pretos, e outro para valores, que armazena os valores finais da função objetivo.

É estabelecido um intervalo de -45º a 45º, com saltos de 1º. Para cada ângulo é feito uma rotação na imagem em tons de cinza, depois é feita a binarização da imagem e depois uma inversão de pixels pretos para brancos.

A projeção horizontal é feita usando a função **np.sum()** do **Numpy**, dividindo a imagem binarizada por 255 apenas no eixo 1, para que seja possível obter a projeção horizontal.

A função objetivo é calculada fazendo a diferença do perfil adjacente e depois somando seus quadrados, conforme sugerido pela descrição do trabalho.

Depois é plotado todas as projeções em um único gráfico, para visualizar as projeções realizadas.

No fim, o ângulo final será o perfil com maior valor. A função **projecao_horizontal()** retorna então o ângulo detectado e a imagem corrigida.

Para corrigir a inclinação da imagem foi implementado a função **giralmagem()**, que recebe como parâmetros a imagem original e o ângulo que deve-se realizar a rotação. Essa função é utilizada tanto no algoritmo de projeção horizontal quanto na transformada de hough.

B. Código da transformada de hough

Para detectar a inclinação utilizando transformada de hough, foi implementado a função **transformada_hough(),** que recebe a imagem original. É feito a conversão da imagem original em tons de cinza também.

Para conseguir executar todos os exemplos dados na descrição do trabalho (contidos na pasta /imgs) foi necessário fazer um pré-processamento na imagem em tons de cinza. Isso porque algumas imagens possuem ruídos, que atrapalham o algoritmo detectar as retas reais da imagem. Sem esse pré-processamento, o algoritmo acaba detectando retas nos ruídos, levando o código a nunca convergir a detecção do ângulo de inclinação real. Outro caso de problema é que, devido aos ruídos, as retas acabam não sendo detectadas em nenhum ponto.

Assim, foi aplicado suavização gaussiana com *kernel* 5x5 e uma equalização do histograma da imagem suavizada. Isso foi o suficiente para conseguir obter bons resultados em todos os exemplos dados pelo trabalho.

Na sequência, é feita a binarização da imagem equalizada, seguida da aplicação do algoritmo de Canny para detectar as bordas.

Com isso, aplicou-se a função **cv2.HoughLines(),** disponível na biblioteca opencv, informando as bordas detectadas, o valor rho, theta e um limiar. Nos testes, a melhor configuração para o limiar foi de 130.

Após isso, é salvo todos os ângulos detectados pela função **cv2.HoughLines()** em um vetor, e, a partir desse vetor, calculou-se a moda dos ângulos obtidos. Nos testes, avaliou-se obter a média dos ângulos obtidos e também o valor máximo, porém, devido às imagens com ruídos dos exemplos do trabalho, somente a estratégia da moda resultou em bons resultados.

Logo depois, o valor obtido da moda é subtraído de 90º. a função

transformada_hough() retorna então o ângulo detectado e a imagem original corrigida, utilizando a função **giralmagem().**

C. Validação e apresentação dos dados

Ao final da execução do programa, no diretório do trabalho, será gerado a imagem final, que é a imagem de entrada com a devida correção de ângulo de inclinação.

Durante a execução do programa, além das imagens e gráficos que serão plotados na tela, serão apresentados no terminal o ângulo detectado e a identificação dos textos antes e depois do processo, utilizando a biblioteca Tesseract.

IV. Resultados

Nessa seção é apresentado alguns experimentos realizados durante a implementação.

O primeiro experimento é executando o comando:

python alinhar.py <img> 0 img_saida.png

Onde, em **<img>** foram testadas todas as imagens de testes disponibilizados na descrição do trabalho (disponíveis na pasta **/imgs**), utilizando o modo 0, ou seja, utilizando o algoritmo de projeção horizontal.

Em todos os testes, obteve-se praticamente a detecção do mesmo valor de ângulo de inclinação feito na imagem. Isso deve-se às rotações de 1º em cada projeção. Observe as Figuras 1 a 8 os exemplos de inclinação na imagem e suas respectivas correções.

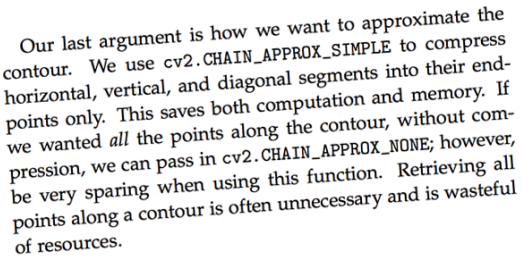


Figura 1. Imagem neg_4, com -4º de rotação.

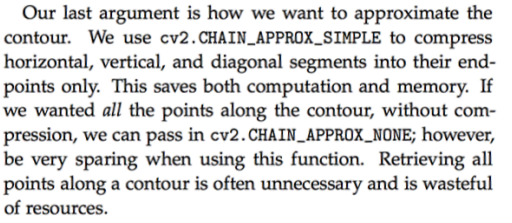


Figura 2. Imagem neg_4 corrigida.

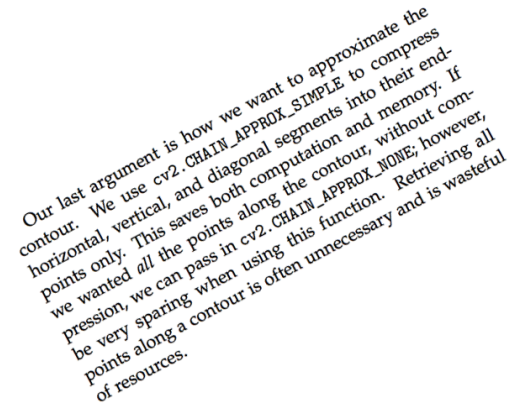


Figura 3. Imagem neg_28 com -28º de rotação.

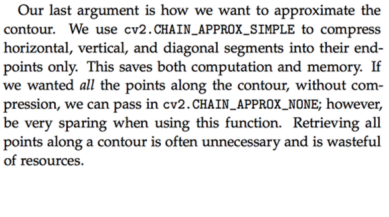


Figura 4. Imagem neg_28 corrigida.

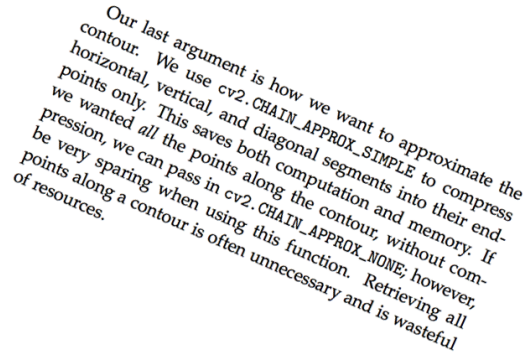


Figura 5. Imagem pos_24 com 24º de rotação.

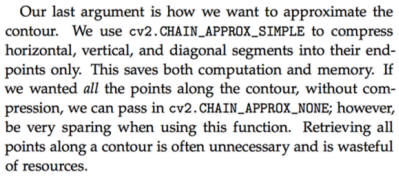


Figura 6. Imagem pos_24 corrigida.

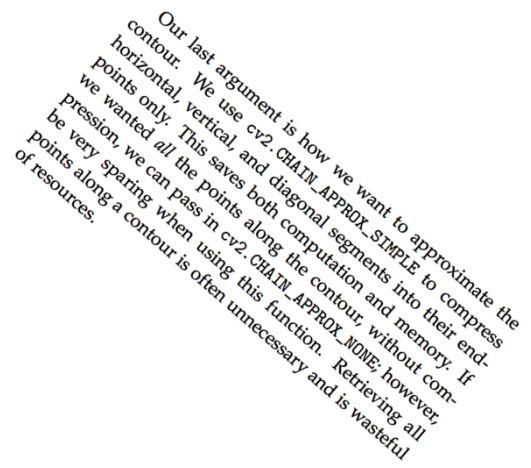


Figura 7. Imagem pos_41 com 41º de rotação.

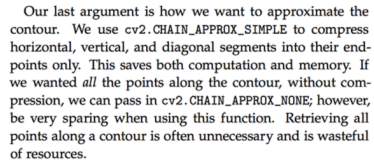


Figura 8. Imagem pos_41 corrigida.

Na figura **neg_4.png**, devido a rotação baixa de -4º, o Tesseract consegue detectar o texto com essa inclinação. O mesmo ocorre após corrigir a inclinação. Porém isso não ocorre para as figuras **neg_28.png, pos_24.png** e **pos_41.png**, onde o Tesseract só consegue detectar o texto após o ajuste de inclinação.

Executando as imagens **partitura.png, sample1.png e sample2.png** observa-se maiores problemas devido a existência de ruídos, que foram corrigidos utilizando filtro de suavização gaussiano e equalização de histograma. Segue as Figuras 9 a 14.

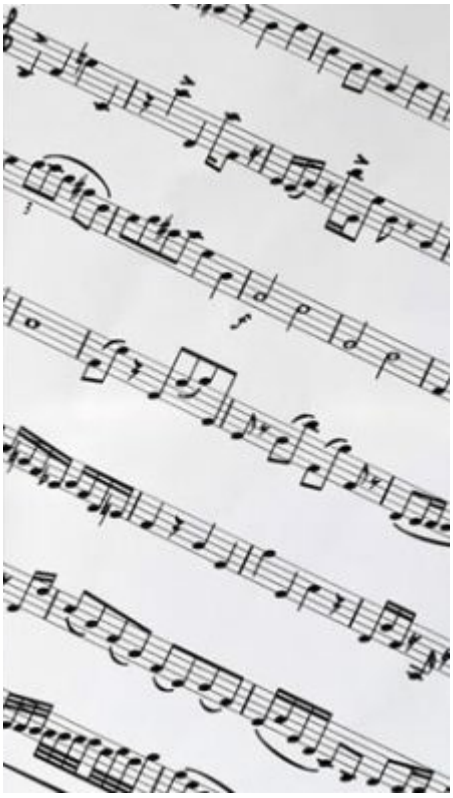


Figura 9. Imagem partitura com inclinação positiva desconhecida e com ruídos na imagem.

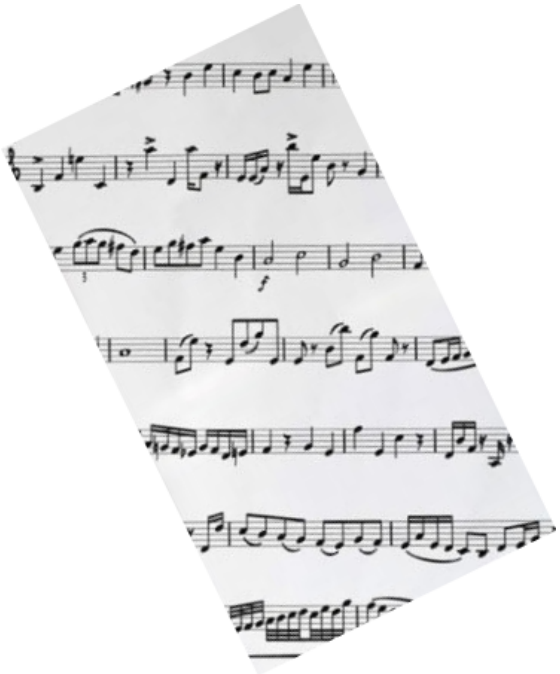


Figura 10. Imagem partitura com a devida correção de inclinação de 26º.

Figura 11. Imagem sample1 com inclinação positiva desconhecida.

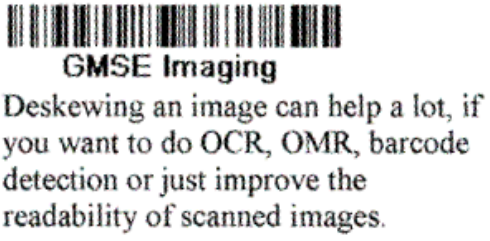


Figura 12. Imagem sample1 com a devida correção de 14º.

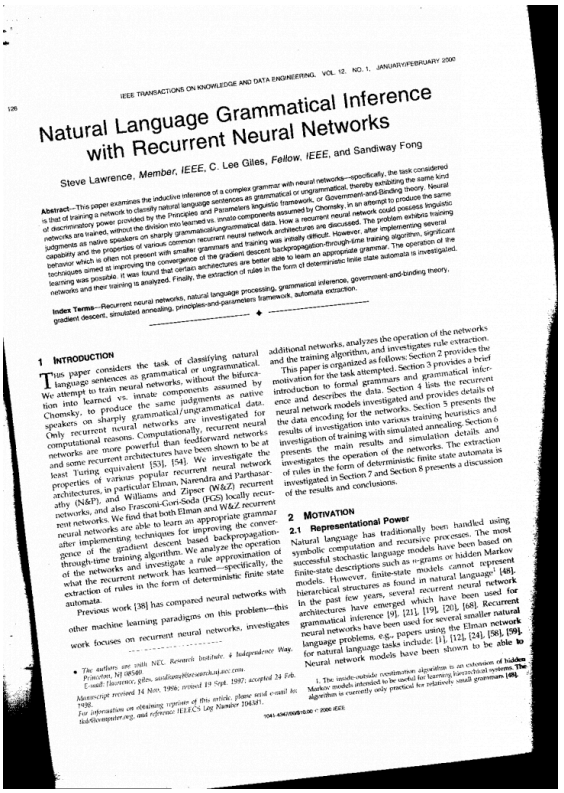
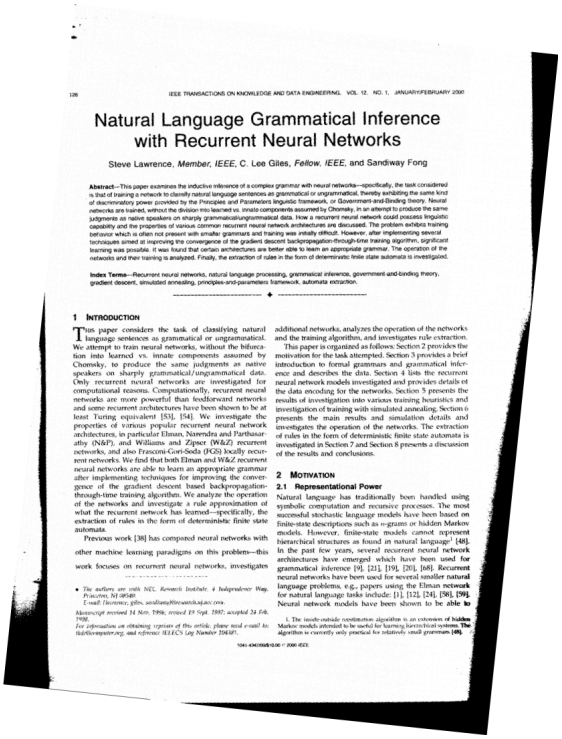


Figura 13. Imagem sample2 com inclinação negativa desconhecida e com ruídos na imagem.



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 [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 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].

Figura 14. Imagem sample2 com a devida correção de inclinação -6º.

Como é apresentado nas Figuras 9 a 14, foi possível identificar e corrigir as suas rotações desconhecidas de forma satisfatória. Utilizando o OCR Tesseract, não foi possível identificar o texto nessas imagens com inclinações, porém, após realizar a correção, foi possível identificar textos nelas. Apenas uma ressalva, devido a qualidade das Figuras, principalmente **sample2.png**, o Tesseract acabou detectando palavras erradas.

Sobre as projeções horizontais, foram feitas plotagem de gráficos apresentando linhas de *pixels* por número de pixels projetados. Como exemplo, na Figura 15 é apresentado o gráfico das projeções horizontais de todos os ângulos testados na imagem **neg_4.png**. É Possível obter uma melhor visualização executando o seguinte comando:

python alinhar.py imgs/neg_4.png 0 img_saida.png

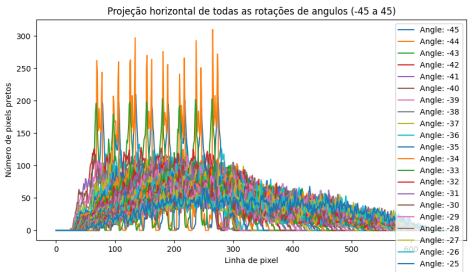


Figura 15. Plotagem de todas as projeções horizontais de -45º a 45º da imagem neg_4.png.

Essas plotagens de gráficos foram feitas para observar que, quanto mais próximo de 0º a imagem ficar em relação a sua rotação, maior será a quantidade de *pixels* pretos no perfil horizontal. Com esse comportamento, basta selecionar o ângulo que fornece essa maior variação. Com essa informação, foi possível implementar a projeção horizontal.

O segundo experimento é executando o comando:

python alinhar.py <img> 1 img_saida.png

Onde, em **<img>** foram testadas todas as imagens de testes disponibilizados na descrição do trabalho (disponíveis na pasta **/imgs**), utilizando o modo 1, ou seja, utilizando o algoritmo da transformada de hough.

Os resultados são praticamente todos iguais às figuras 1 a 14, com algumas observações. Na imagem **neg_4.png** o ângulo detectado foi -4.00001º. Na imagem **neg_28.png** o ângulo identificado foi -28.00001º. Na imagem **pos_24.png** o ângulo identificado foi 23.99992º. Na imagem **pos_41.png** 40.99998º. Na imagem **partitura.png** o ângulo obtido foi 25.99992º. Na imagem **sample1.png** o ângulo obtido foi 13.99992º. Na imagem **sample2.png** o ângulo obtido foi -6º.

Usando Tesseract, o resultado foi exatamente igual ao experimento com o algoritmo de projeção horizontal.

Com isso, observa-se que, aplicando ambos os algoritmos, os resultados são praticamente idênticos, com ressalva a transformada de hough, que dá um valor de ângulo mais exato da inclinação.

V. Conclusões

Neste trabalho foi possível explorar a aplicação dos algoritmos de projeção horizontal e transformada de hough para detectar e corrigir inclinações em imagens.

Ambos os algoritmos aparentam ter bons desempenhos, porém, dependendo da inclinação da imagem, a projeção horizontal pode ter alto custo computacional devido seus testes de rotações e projeções horizontais.

O algoritmo de projeção horizontal aparenta uma maior facilidade de compreensão de execução e maior facilidade de interpretação de como detectar inclinações, porém a transformada de hough aparenta maior robustez em aplicações diversas.

Nos experimentos, ambos os algoritmos obtiveram bons resultados, porém, dependendo da inclinação, a transformada de hough apresentou melhores resultados. Caso a imagem tenha muitos detalhes e ruídos, é preciso aplicar pré-processamento na mesma para que a transformada de hough não detecte os ruídos como retas. Nisso a projeção horizontal apresenta vantagem, pois basta realizar projeções em ângulos diferentes e selecionar o que apresentar maior quantidade de *pixels* pretos projetados.