Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Matemática Aplicada e Computacional

Categorização de comandos de fala com redes neurais artificiais

Lucas Almeida da Silva

Monografia Final

Trabalho de
Formatura Supervisionado

Supervisor: Prof. Dr. Flavio Soares Correa da Silva

São Paulo
2023

"The human spirit must prevail over technology"
- Albert Einstein

Agradecimentos

Gostaria de expressar meus sinceros agradecimentos aqueles que desempenharam papéis fundamentais na jornada que até o desenvolvimento deste trabalho.

Primeiramente, agradeço a meus pais, Roberval e Doracy, por seu amor incondicional, apoio e pela dedicação incansável em me proporcionar oportunidades e incentivo ao longo de minha trajetória. Sem a base que vocês me proporcionaram, nada disso seria possível.

Também agradecer ao Profº Flávio Soares Corrêa da Silva, meu orientador, expresso minha profunda apreciação por sua orientação, seus direcionamentos em me inspirar e elucidar ao longo deste trabalho. Suas contribuições foram decisivas para o desenvolvimento.

Não posso deixar de agradecer aos meus colegas de faculdade, cuja a colaboração ao longo desses anos tornaram o aprendizado mais leve e enriquecedor. Juntos, enfrentamos desafios, celebramos conquistas e construímos memórias que levo comigo para o futuro.

Este trabalho é o resultado de esforços coletivos e apoio constante, e agradeço a todos que contribuíram, direta ou indiretamente, para o meu percurso acadêmico.

Resumo

Lucas Almeida da Silva. **Categorização de comandos de fala com redes neurais artificiais**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O processo de viabilização para tornar computadores capazes de interpretar a fala é especificamente estudado por uma subárea do campo de estudo do Processamento de Linguagem Natural (NLP), conhecida como Reconhecimento Automático de Fala (ASR). Atualmente, os resultados obtidos ao longo de décadas por essa área trazem diversas aplicações, destacando-se, em especial, causas nobres como a Tecnologia Assistencial (AT). Muitas das soluções oferecidas no mercado são robustas e abrangentes, no entanto, estão geralmente associadas a empresas de tecnologia de grande porte que, por sua vez, cobram pela utilização a partir de um certo limite. Tendo em vista que soluções simplificadas são suficientes para uma considerável parte das necessidades, criar soluções independentes pode ser uma alternativa. Pelas razões citadas, a monografia deste trabalho apresentará os passos necessários para construir um modelo baseado em Rede Neural Artificial (ANN) para a categorização de comandos a partir da fala. Os detalhes abrangerão desde a busca por dados e o pré-processamento até a estruturação e criação do modelo, o processo de treinamento e a inferência dos resultados obtidos. O objetivo é desenvolver um modelo eficiente e adaptável capaz de interpretar com eficácia um conjunto de palavras escolhido.

Palavras-chave: Processamento de Linguagem Natural. Tecnologia Assistiva. Comandos de Fala.

Abstract

Lucas Almeida da Silva. **Categorization of spoken commands with artificial neural networks**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

The process of making computers capable of interpreting speech is specifically studied by a subarea within the field of Natural Language Processing (NLP), known as Automatic Speech Recognition (ASR). Currently, the results obtained over decades in this field bring various applications, particularly standing out in noble causes such as Assistive Technology (AT). Many of the solutions offered in the market are robust and comprehensive; however, they are often associated with large technology companies that, in turn, charge for usage beyond a certain limit. Considering that simplified solutions are sufficient for a considerable portion of needs, creating independent solutions can be an alternative. For the reasons mentioned, the monograph of this work will present the necessary steps to build a model based on Artificial Neural Network (ANN) for the categorization of commands from speech. The details will cover everything from data search and preprocessing to the structuring and creation of the model, the training process, and the inference of the obtained results. The goal is to develop an efficient and adaptable model capable of effectively interpreting a chosen set of words.

Keywords: Natural Language Processing. Assistive Technology. Speach Commands.

Lista de abreviaturas

ASR	Reconhecimento Automático de Fala (*Automatic Speech Recognition*)
NLP	Processamento de Linguagem Natural (*Natural Language Processing*)
AT	Tecnologia Assistiva (*Assistive Technology*)
ANN	Rede Neural Artificial (*Artificial Neural Networks*)
STFT	Transformada de Fourier de Tempo Curto (*Short-Time Fourier Transform*)
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

Lista de figuras

Lista de programas

Sumário

Apêndices

Anexos

Capítulo 1

Introdução

1.1 Motivação

O estudo sobre a história do ASR apresentado por Furui (Furui, 2005), mostra que os estudos buscando viabilizar soluções para a conversão automática da fala em texto iniciaram-se nas décadas de 1950 e 1960. Desde então, diversos pesquisadores têm explorado e proposto abordagens para enfrentar os desafios do reconhecimento de fala, e entre essas abordagens está a utilização de ANN.

Atualmente, as grandes empresas de tecnologia, como Google, IBM e Microsoft, oferecem soluções robustas e eficientes no campo do reconhecimento de fala. Destaca-se o Speech-to-Text do Google, que emprega algoritmos de ANN e exibe uma alta capacidade de transcrever áudios em diversos idiomas com alta precisão. Entretanto, o desenvolvimento para criação e a disponibilização dessas tecnologias exigem investimentos significativos em infraestrutura e pesquisa por parte dessas empresas, por consequência, a utilização dessas tecnologias de forma gratuita é normalmente limitada e para utilização ilimitada possui um custo associado.

Por outro lado, também existem aplicações de ASR que não demandam soluções tão robustas, sendo direcionadas a atender necessidades mais simples ou específicas. Nesses casos, para um conjunto limitado e pequeno de palavras é possível viabilizar a construção de um modelo que seja capaz de atender as necessidades com uma precisão considerável e baixo ou nenhum custo.

Nesse contexto, as tecnologias AT ganham relevância, trazendo praticidade e impactando positivamente a vida para grupos com algum grau ou tipo de deficiência. Combinando essa demanda com o potencial da aplicação do ASR, surge o desenvolvimento de protótipos como o proposto por Peixoto (Peixoto *et al.*, 2015), que utiliza do reconhecimento de comandos de fala para possibilitar a movimentação de cadeiras de rodas.

1.2 Objetivo

Diante do cenário apresentado, a monografia desse trabalho tem como objetivo principal abordar a criação de um modelo de categorização da fala, especialmente de palavras únicas. Buscando alcançar resultados precisos e promover e guiar uma utilização acessível e flexível. Para isso, serão devidamente detalhados os passos essenciais para a construção do modelo, baseando nas estruturas de ANN, que têm se mostrado particularmente eficazes em tarefas de reconhecimento de padrões.

Este estudo então será estruturado de forma a explicitar os principais aspectos da construção do modelo, desde a busca e a compreensão dos dados e seu pré-processamento, a construção e o treinamento do modelo, por fim, a etapa de inferência para avaliação e análise dos resultados obtidos. Almeja-se que este trabalho contribua para o desenvolvimento e acessibilidade na criação tecnologias que envolvam o reconhecimento de fala, principalmente aquelas com enfoque assistencial, que busquem contribuir com a inclusão de indivíduos que enfrentam desafios específicos causados por alguma deficiência.

Capítulo 2

Dados

2.1 Base de dados

O primeiro passo essencial na construção deste projeto é localizar uma base de dados que possibilite a criação de um modelo eficiente de aprendizado de máquina. Como apresentado no capítulo anterior (1), nosso objetivo é encontrar conjuntos de gravações de áudio contendo falas de uma única palavra, acompanhadas de seus rótulos correspondentes.

Antes de selecionar uma base de dados, é importante considerar alguns pontos. Cada indivíduo possui características vocais únicas, semelhantes a uma espécie de "assinatura de voz". No entanto, de acordo com resultados apresentados por Shafran (Shafran *et al.*, 2003), em geral, algumas características vocais estão relacionadas a fatores como gênero e idade. Um dos objetivos deste trabalho é que o modelo construído demonstre um desempenho eficaz e consistente com vozes de diversos gêneros e faixas etárias. Portanto, é crucial que a base de dados escolhida exiba variabilidade em relação a essas características.

Inicialmente, o objetivo era encontrar dados de áudio em português do Brasil. Nesse sentido, existem algumas bases de dados disponíveis. Um destaque é a Common Voice da Mozilla, uma base de dados colaborativa onde pessoas ao redor do mundo podem contribuir com áudios de voz em diferentes idiomas e avaliar as sentenças correspondentes aos áudios de outros colaboradores. No entanto, essa base de dados apresenta uma limitação em relação à variedade de áudios correspondentes a uma única palavra, o que levou à decisão de descartar sua utilização.

A partir deste ponto, a pesquisa concentrou-se nas restrições específicas do nosso problema, que incluem o uso de bases de dados contendo apenas áudios de uma única palavra em português do Brasil. No entanto, essa busca não produziu resultados satisfatórios, e nenhuma base de dados adequada às nossas condições foi encontrada. Diante dessa situação, a solução mais próxima do nosso objetivo seria a criação de uma base de dados própria. No entanto, essa opção foi descartada devido à necessidade de envolver um grande número de colaboradores e ao tempo considerável necessário para sua construção. Portanto, a solução adotada foi flexibilizar a restrição de idioma mais popular, optando por buscar áudios em inglês.

Com as restrições atualizadas, a base de dados que atende aos nossos requisitos e será utilizada no desenvolvimento deste trabalho é a Mini Speech Commands, disponibilizada pelo Google. Ela inclui um total de 8000 arquivos de áudio no formato WAV, com 1000 arquivos para cada um dos comandos *"down"*, *"go"*, *"left"*, *"no"*, *"right"*, *"stop"*, *"up"* e *"yes"*.

Esses dados serão divididos de forma aleatória em conjuntos sendo, o de treinamento, validação e teste obedecendo respectivamente as proporções 80%,10%,10% para o total de 8000 dados, além disso para cada conjunto a quantidade de dados de cada comando deve ser a mesma.

2.2 Rótulos

Para que seja possível trabalhar de forma matemática e computacional com as representações de palavras é necessário partir para uma definição de como é feita transformação entre esses mundos.

Primeiramente é necessário definir uma ordem para o conjunto de n comandos, sendo assim, cada comando terá um respectivo índice numérico de representação, assim dado que sabemos qual comando é representado pelo índice i a sua representação geral será um vetor onde o índice i assume valor 1 e os demais índices assumem o valor 0. No caso específico desse trabalho se definem $n = 8$ e os índices seguem a ordem *"down"*, *"go"*, *"left"*, *"no"*, *"right"*, *"stop"*, *"up"* e *"yes"*.

A interpretação é de que o valor assumido em cada índice i do vetor representa a probabilidade de representação do comando i. Isso será importante pois nos resultados do modelo esses vetores apresentam valores probabilísticos e assumiremos que o índice i de maior probabilidade será o o comando.

2.3 Áudio

2.3.1 *Formato de onda*

Inicialmente, os arquivos de áudios são disponibilizados em formato digital pela extensão WAV, um formato capaz de armazenar sem perda de qualidade uma representação de ondas sonoras, ou seja, inicialmente nossos dados são representações em formato de onda, sendo a variação da amplitude de um sinal de áudio em relação ao tempo.

Por se tratar de um formato digital, temos a representação da onda de forma discreta, em que a quantidade de amostras e amplitudes é definida pela taxa de amostragem. Neste trabalho, a taxa de amostragem foi definida como 16 kHz, o que significa que a cada segundo, 16000 medições da amplitude do sinal de áudio são feitas. Os valores assumidos pelas amplitudes é um conjunto de tamanho $2^{16} = 65536$, e neste trabalho, esses valores variam de -1 até 1.

Graficamente, os sinais de áudio são representados como formas de onda, onde o eixo da abscissa representa o tempo e o eixo da ordenada representa as amplitudes. Para fins de comparação, os gráficos gerados mostram a representação em formato de onda para a fala

de cada palavra no conjunto de comandos na figura (2.1) e a representação de várias falas do mesmo comando *"down"* na figura (2.2).

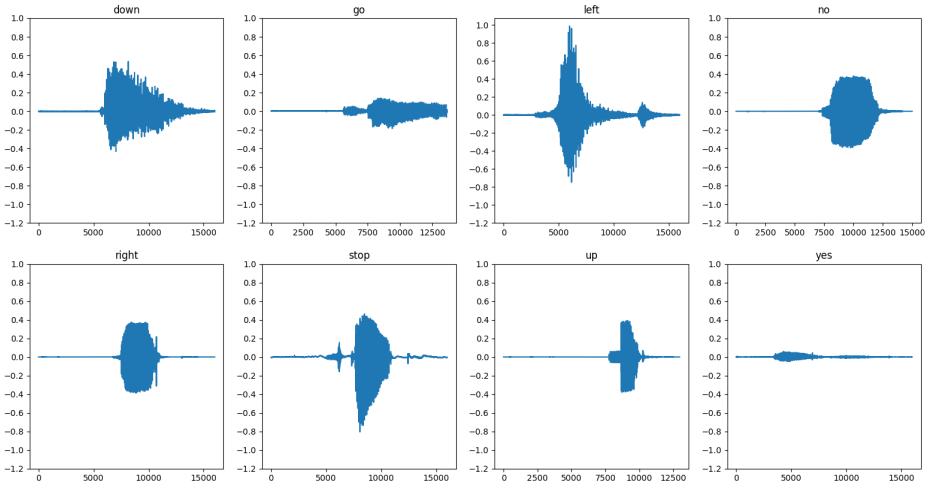

Figura 2.1: *Comandos em formato de onda*

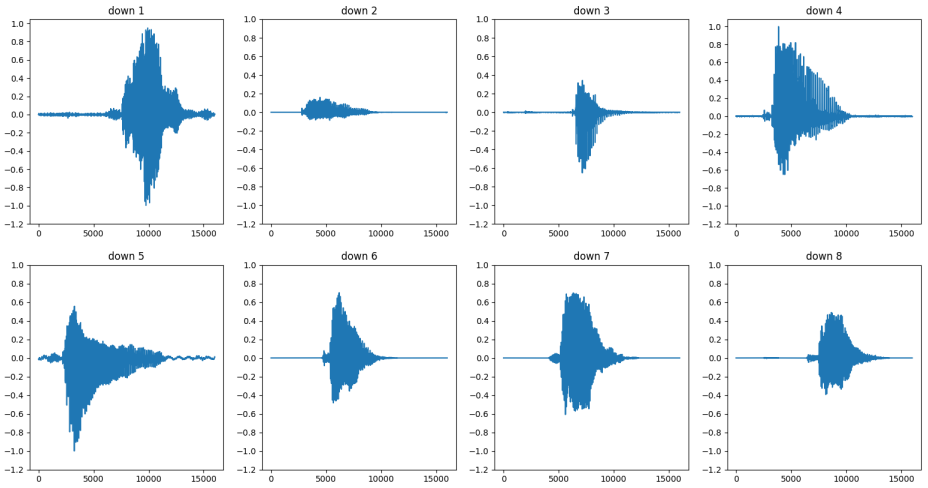

Figura 2.2: *Comando "down" em formato de onda*

2.3.2 Espectrograma STFT

A STFT é uma extensão da Transformada de Fourier convencional usada para analisar sinais que tem alterações espectrais ao longo do tempo ou seja, não estacionários. A estratégia desse método é dividir um sinal em segmentos, chamados de janelas, e aplicar a Transformada de Fourier convencional para cada janela. O resultado é um espectrograma que representa a variação das frequências no tempo.

Inicialmente, é necessário algumas definições, como o total de amostras N, a quantidade de amostras por janela M e o tamanho de amostras de sobreposição R, sendo as amostras que estão em mais de uma janela. Tendo isso é possível calcular a quantidade J de janelas pela equação (2.1) e a quantidade K de frequências que serão usadas, calculada pela equação (2.2).

$$J = \left\lfloor \frac{N - M}{R} \right\rfloor + 1 \tag{2.1}$$

$$K = \frac{M}{2} + 1 \tag{2.2}$$

Ainda é necessário a definição da função de janela, que pondera a amplitude do sinal ponto a ponto. Comumente utiliza-se a janela de Hanning, que por padrão será adotada nesse trabalho e é definida pela equação (2.3). Então, para cada uma das J janelas será calculada para cada uma das K frequências a STFT através da equação (2.4). E finalmente, para se obter o espectrograma do sinal de áudio, se mantém apenas a parte de real através da equação (2.5).

$$w[n] = \frac{1}{2}\left(1 - cos\left(\frac{2\pi \cdot n}{M - 1}\right)\right) \tag{2.3}$$

$$S_{m,k} = \sum_{n=0}^{M-1} x[n + mR] \cdot w[n] \cdot e^{-i2\pi \frac{k}{M} \cdot n} \tag{2.4}$$

$$Y_{m,k} = |S_{m,k}|^2 \tag{2.5}$$

O espectrograma gerado será uma representação bidimensional com as dimensões definidas por $K \times J$, sendo essas definidas a partir de N, M e R, no caso desse trabalho em específico os parâmetros definidos foram $N = 16000$, $M = 256$ e $R = 128$ resultando em dimensões finais 124×129. Essa será a formato de representação dos áudios utilizados nesse trabalho.

Para a representação gráfica, é aplicada uma soma, seguida pela aplicação do logaritmo conforme definido pela equação (2.6). Neste trabalho, o valor adicionado é $\epsilon = 10^{-15}$, o qual tem a finalidade de garantir que o cálculo do logaritmo seja bem definido. O objetivo final é amplificar as diferenças e torná-las mais visíveis no gráfico.

$$Z_{m,k} = ln(Y_{m,k} + \epsilon) \tag{2.6}$$

Nos gráficos o eixo da abscissa representa as variações de tempo, ou seja, a dimensão de tamanho J enquanto o eixo da ordenada representa as variações de frequência, sendo a dimensão de tamanho K. Enquanto os valores serão representados por uma escala de cores, sendo os de maior magnitude serão representados pela cor amarela clara, enquanto os de menor pela cor azul escura. Para fins de comparação, os gráficos gerados apresentam

a representação da fala de cada palavra no conjunto de comandos na figura (2.3) e a representação de várias falas do mesmo comando *"down"* na figura (2.4).

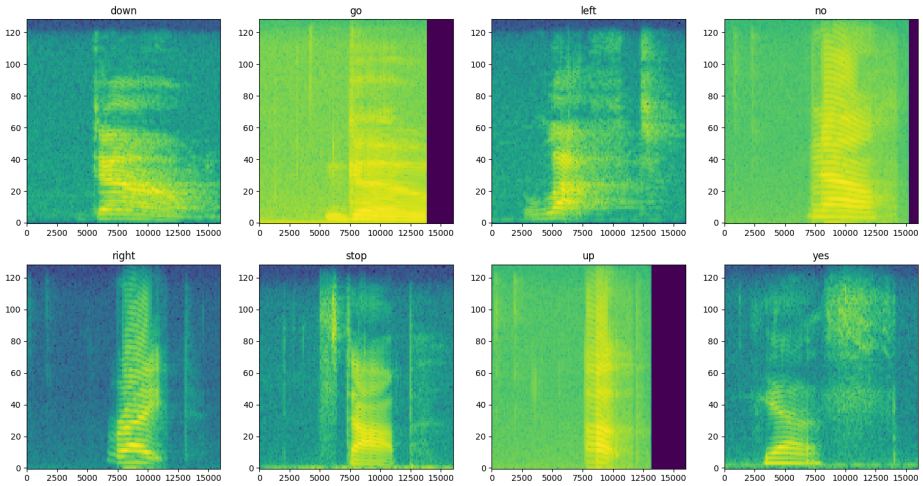

Figura 2.3: *Comandos em espectrogramas*

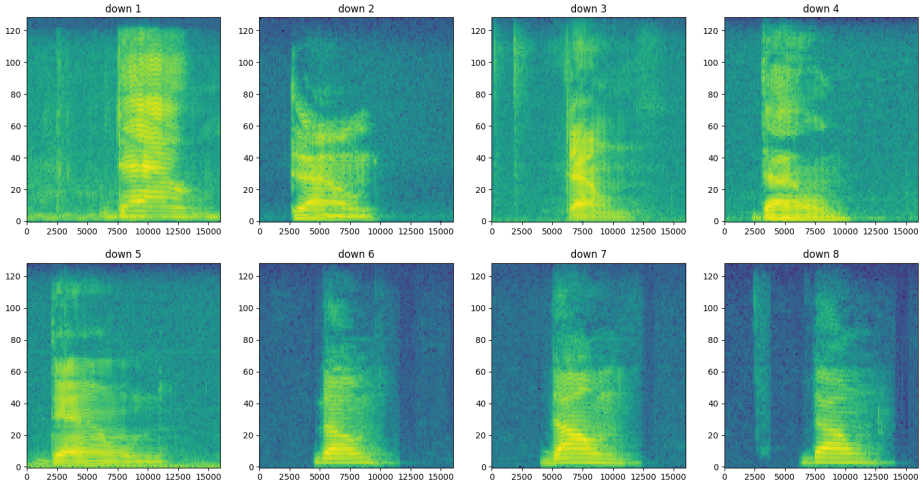

Figura 2.4: *Comando "down" em espectrogramas*

Capítulo 3

Modelo

Os princípios apresentados por Sivanandam (Sivanandam e Paulraj, 2009) demonstram que uma ANN é um sistema de processamento de informações projetado como uma generalização de um modelo matemático baseado na cognição humana. Sua função é gerar um padrão de saída com base em um padrão de entrada específico. Para alcançar esse objetivo, a rede se inspira no funcionamento biológico dos neurônios humanos e é composta essencialmente por neurônios artificiais. Nesse processo, é gerado um modelo que recebe uma entrada e executa uma série de operações para produzir um resultado de saída de interesse.

A rede possui uma estrutura organizacional baseada em camadas construídas e interconectadas para realizar operações matemáticas específicas. O propósito final dessa estrutura é capturar relações complexas nos dados, que podem ser difíceis de perceber para um computador, e produzir uma saída que represente um objeto de interesse.

Assim como o processo de aprendizado humano, a rede passa por uma etapa de aprendizado comumente chamada de treinamento. Nessa fase, as operações das camadas são ajustadas por meio de um processo de otimização não linear. O objetivo é minimizar uma função conhecida como função de perda, que mede a discrepância entre as saídas geradas pelo e as saídas reais. Esses ajustes envolvem iterações, chamadas de épocas, durante as quais ocorrem subiterações que processam uma parte dos dados, chamada lote. É nesse momento que as atualizações dos pesos do modelo são realizadas.

Neste estudo, optou-se por adotar uma arquitetura de ANN baseada no conceito do *encoder*, conforme apresentado no trabalho *"Attention is All You Need"* (Vaswani *et al.*, 2017), que foi responsável por introduzir a arquitetura *Transformer*. O *encoder* é reconhecido por sua eficácia no tratamento de sequências de dados, bem como por sua capacidade de capturar relações complexas. Essa versatilidade torna o *encoder* uma excelente escolha para diversas situações, permitindo sua adaptação a diferentes características dos dados.

Um passo importante na criação da arquitetura é compreender e estruturar as representações dos dados de entrada e saída. Portanto, com base no capítulo de dados (2) apresentada, o objetivo é desenvolver um modelo capaz de processar representações de áudio na forma de espectrograma STFT (2.3.2) e classificá-los em rótulos de comandos (2.2). Para alcançar esse objetivo, são necessários ajustes adicionais no *encoder* para tratar espe-

cificamente o caso deste trabalho, resultando em uma arquitetura que pode ser visualizada em forma de um fluxograma de camadas na figura (3.1). A partir dessa perspectiva geral, é possível avançar e aprofundar a compreensão das responsabilidades de cada camada e sua colaboração no processo.

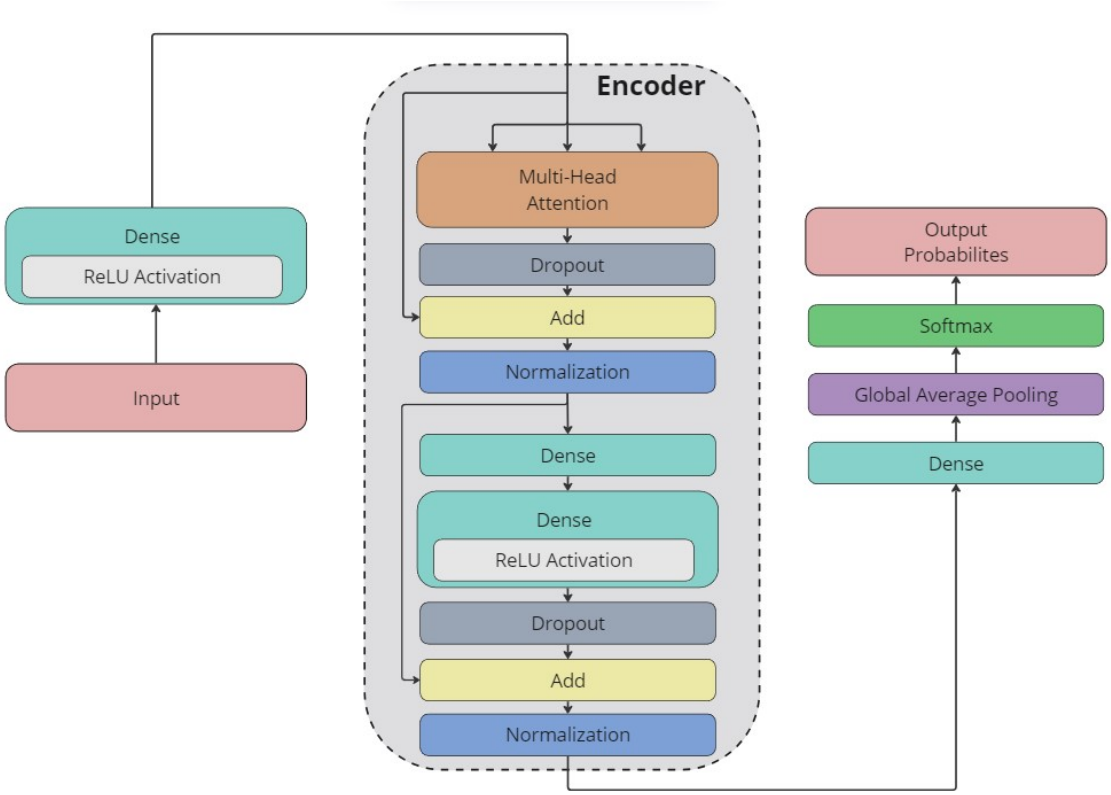

Figura 3.1: *Fluxograma do modelo*

Um fator importante a ser definido em modelos de aprendizado profundo, e que também será adotado neste trabalho, é a prática de processamento em lotes. Isso permite que a ANN aproveite ao máximo a capacidade de processamento paralelo fornecida pelo hardware, otimizando a eficiência durante a fase de treinamento. Para isso, é definida uma dimensão adicional que especifica o tamanho do lote, representado por B. Portanto, a entrada inicial do modelo neste trabalho será definida com dimensões $B \times 124 \times 129$ e a saída final com dimensões $B \times 8$.

3.1 Camadas

3.1.1 *Dense*

A camada de *Dense*, também conhecida como *Feed Forward*, é uma estrutura na qual os dados de entrada são submetidos a um processo de transformação que envolve ponderação, adição de viés e aplicação de funções de ativação, com o objetivo de gerar as saídas. A denominação dessa camada decorre da natureza direcional do fluxo de dados durante o processamento, caracterizado por ser unidirecional. Nesse sentido, não ocorre realimentação direta das saídas resultantes de volta para as entradas iniciais. Para o cálculo por

trás dessa camada, é considerada a entrada x, os pesos W e o viés b. Além desses termos, é necessária a definição de D, que determinará as dimensões da saída ponderada z. Dessa forma, o cálculo é definido da seguinte forma:

$$z = x \cdot W + b \tag{3.1}$$

Após o cálculo da saída ponderada z, a saída final y é determinada por meio de uma função de ativação. O objetivo dessa função é introduzir ou não a característica de não linearidade nas saídas. Quando há a introdução de não linearidade na saída, o propósito é permitir que o modelo aprenda a captar relações complexas nos dados. Existem várias funções de ativação comumente utilizadas, porém neste trabalho, duas funções de ativação diferentes serão empregadas em diferentes camadas *Dense* presentes na arquitetura do modelo, sendo elas, a função identidade (3.2) e a função retificadora (ReLU) (3.3) qual será responsável por introduzir não linearidade nos dados. É importante observar que, devido à natureza dos dados multidimensionais, a aplicação da função ReLU resulta em uma aplicação elemento por elemento.

$$y_{i,j,k} = z_{i,j,k} \tag{3.2}$$

$$y_{i,j,k} = \max(z_{i,j,k}, 0) \tag{3.3}$$

Para o caso de três dimensões qual será aplicado nesse trabalho a entrada x com dimensões $B \times M \times N$ resulta em uma saída y com dimensões $B \times M \times D$, onde D é definido como parâmetro na criação da camada. No contexto deste trabalho, a dimensão D escolhida em cada utilização da camada *Dense* varia, sendo que as quatro camadas utilizadas na estrutura terão a dimensão definida na seguinte ordem 128, 512, 128 e 8.

3.1.2 *Multi-Head Attention*

O mecanismo de atenção tem como ideia central permitir que cada elemento em uma sequência interaja com todos demais elementos, capturando assim relações e dependências relacionadas.Para isso é realizado o cálculo de uma pontuação a cada par de elementos em uma sequência. Essas pontuações de atenção determinam a importância relativa de cada elemento em relação aos outros. Para o cálculo, são criadas três ponderações K (Key), Q (Query) e V (Value), esses criados a partir de de uma entrada x, os pesos W e o viés b são então calculados:

$$K = x \cdot W_k + b_k \tag{3.4}$$

$$Q = x \cdot W_q + b_q \tag{3.5}$$

$$V = x \cdot W_v + b_v \tag{3.6}$$

As dimensões de K, Q e V seguem o mesmo padrão da camada *Dense* (3.1.1) onde as saídas terão dimensão definida por um parâmetro D resultando em dimensões $B \times M \times D$. Após a criação das ponderações é então calculado z em (3.7). Então com objetivo de transformar cada elemento $z_{i,j,k}$ em uma probabilidade, é aplicando a função exponencial normalizada *softmax* (3.8). Finalmente é calculado a saída inicial $\hat{z}$ a partir da operação

(3.9).

$$z = \frac{Q \cdot K^T}{\sqrt{D}} \tag{3.7}$$

$$softmax\left(z_{i,j,k}\right) = \frac{e^{z_{i,j,k}}}{\sum_{k=1}^{D} e^{z_{i,j,k}}} \tag{3.8}$$

$$\hat{z} = softmax\left(z\right) \cdot V \tag{3.9}$$

O processo então é feito paralelamente diversas vezes, o que justifica o nome da camada, sendo cada um desses processos chamados de *head* feitos uma ou múltiplas vezes. Cada *head* resultará em uma saída inicial, portanto, serão $\hat{z}^{(1)}, \hat{z}^{(2)}, ..., \hat{z}^{(n)}$ sendo n o número de *heads* escolhido, essas por sua vez serão concatenadas formando uma única saída inicial $\hat{z}$, que finalmente será ponderada por pesos $W_{\hat{z}}$ resultando em uma saída final y.

$$y = \hat{z} \cdot W_{\hat{z}} \tag{3.10}$$

Para a construção do modelo utilizado nesse trabalho, foi aplicada uma única camada *Multi-Head Attention* com duas *heads* e como parâmetro de dimensão $D = 128$.

3.1.3 *Dropout*

A camada de *Dropout* desempenha um papel crucial na construção do modelo. Sendo aplicada exclusivamente durante o treinamento, seu principal objetivo é combater o overfitting, evitando que o modelo se ajuste muito bem aos dados de treinamento, mas falhe em generalizar para dados desconhecidos, resultando em baixa precisão. A operação da camada de *Dropout* é baseada na desativação aleatória de neurônios.

Especificamente neste trabalho, consideramos uma entrada x com dimensões $B \times M \times N$ e uma taxa de *Dropout* p. Nesse contexto, é gerada uma máscara m com as mesmas dimensões de x de acordo com a propriedade (3.11). A saída é então calculada usando a expressão (3.12). Para as camadas de *Dropout* empregadas no modelo, foi utilizada uma taxa $p = 0.10$.

$$m_{i,j,k} = \begin{cases} 0, \text{ com probabilidade } p \\ 1, \text{ com probabilidade } (1-p) \end{cases} \tag{3.11}$$

$$y_{i,j,k} = x_{i,j,k} \cdot m_{i,j,k} \tag{3.12}$$

3.1.4 *Add & Normalization*

Inicialmente a camada *Add* tem objetivo combinar múltiplas entradas de mesma dimensão e soma-las resultando em uma saída. Isso é feito com a intenção de possibilitar a captura de dependências entre diferentes escalas nos dados, a operação realizada se resume

a quantidade I de entradas $x_{(t)}$ e a saída y, sendo relacionados por (3.13).

$$\sum_{t=1}^{T} x_{(t)} = y \tag{3.13}$$

Partindo para a camada *Normalization* seu papel é de estabilizar e tornando a otimização feita durante o processo de treinamento mais eficiente. No caso desse trabalho, onde os dados de entrada x dessa camada possuem três dimensões $B \times M \times N$, calcula-se a média (3.14) e variância (3.15) percorrendo a dimensão de de tamanho N, e em seguida aplica-se a operação apresentada em (3.16).

$$\mu_{i,j} = \frac{1}{N} \sum_{k=1}^{N} x_{i,j,k} \tag{3.14}$$

$$\sigma_{i,j}^2 = \frac{1}{N} \sum_{k=1}^{N} (x_{i,j,k} - \mu_i)^2 \tag{3.15}$$

$$\hat{x}_{i,j,k} = \frac{x_{i,j,k} - \mu_{i,j}}{\sqrt{\sigma_{i,j}^2 + \epsilon}} \tag{3.16}$$

No cálculo de $\hat{x}_{i,j}$ ainda é necessário adicionar um pequeno valor ϵ para evitar a divisão por zero, em especial nesse trabalho as camadas de normalização utilizarão $\epsilon = 1^{-6}$. Depois é então feito o escalonamento multiplicando $\hat{x}_{i,j,k}$ por $\gamma_{i,j}$ e deslocamento somando $\beta_{i,j}$, resultando na saída apresentada na equação (3.17). Sendo tanto o escalonamento $\gamma_{i,j}$ quanto o deslocamento $\beta_{i,j}$ ajustados durante a fase de treinamento.

$$\hat{y}_{i,j,k} = \gamma_{i,j} \cdot \hat{x}_{i,j,k} + \beta_{i,j} \tag{3.17}$$

3.1.5 *Global Average Pooling 1D*

A camada de *Global Average Pooling 1D* é comumente aplicada para redução de dimensionalidade, o objetivo é resumir a informação calculando-se a média dos elementos. A aplicação dessa camada é essencial para a construção desse trabalho pelo aspecto dos dados de entrada iniciais serem de dimensão maior que os de saída final. Pelo fato de que os dados de entrada x recebidos por essa camada possuir dimensões $B \times M \times N$, calcula-se então as médias percorrendo a dimensão de tamanho M que será eliminada, aplicando a operação (3.18) gerando a saída de dimensão $B \times N$.

$$y_{i,k} = \frac{1}{M} \sum_{j=1}^{M} x_{i,j,k} \tag{3.18}$$

Por fim, após a redução de dimensionalidade, a fim de transformar as saídas em vetores de probabilidade é aplicando então a função exponencial normalizada *softmax*, porém diferentemente da aplicada na camada *Multi-Head Attention* (3.1.2), dessa vez será aplicado de forma bidimensional em $B \times N$ portanto a operação aplicada se resume a equação

(3.19).

$$\hat{y}_{i,k} = softmax\left(y_{i,k}\right) = \frac{e^{y_{i,k}}}{\sum_{k=1}^{N} e^{y_{i,k}}} \tag{3.19}$$

3.2 Treinamento

A fase de treinamento é um estágio essencial onde o modelo é exposto aos dados de entrada e saída, com o propósito de aprender as relações necessárias para fazer previsões precisas em novos dados. Para viabilizar esse processo, diversas métricas e configurações precisam ser definidas.

Alguns dos passos importantes incluem a escolha da função de perda, responsável por medir a discrepância a entre as saídas do modelo e as saídas reais, a seleção do otimizador, que determina como a função de perda será minimizada, e a definição da métrica de precisão, que avalia o quão exato o modelo é em suas previsões. Além disso, outros parâmetros, como o número de épocas, são ajustados para otimizar o treinamento do modelo.

Todo esse processo é crucial para capacitar o modelo a generalizar a partir dos dados de treinamento e a realizar previsões úteis em situações do mundo real, tornando-o uma ferramenta valiosa para resolver problemas complexos.

3.2.1 Função de Perda

Existem algumas funções de perda comumente utilizadas, e neste trabalho, devido a se tratar de um problema de classificação, optou-se por adotar a função de perda *Categorical Crossentropy*. O cálculo da perda é realizado com base na saída y gerada pelo modelo e na saída real $\hat{y}$. No caso específico deste trabalho, as saídas são de duas dimensões $B \times N$. Para cada um dos B vetores de probabilidade, a perda é calculada conforme a equação (3.20), e em seguida, é calculada a perda do lote sendo a média das perdas (3.21).

$$L\left(\hat{y}_{i,k}, y_{i,k}\right) = -\sum_{k=1}^{N} \hat{y}_{i,k} \cdot ln\left(\max(y_{i,k}, \epsilon)\right) \tag{3.20}$$

$$L(\hat{y}, y) = \frac{1}{B} \sum_{i=1}^{B} L\left(\hat{y}_{i,k}, y_{i,k}\right) \tag{3.21}$$

Importante ressaltar que pelas definições impostas $0 \leq y_{i,k}, \hat{y}_{i,k} \leq 1$, para garantir que seja possível calcular a função logarítmica, é necessário que o valor mínimo calculado seja $ln(\epsilon)$, por padrão o valor definido é $\epsilon = 1^{-6}$. Além disso, pelas restrições de intervalo de $y_{i,k}$ e $\hat{y}_{i,k}$, e também pela definição de ϵ podemos garantir que $L\left(\hat{y}_{i,k}, y_{i,k}\right) \geq 0$.

Embora simplifiquemos o cálculo da perda ao tratá-la como uma função da saída final y e saída real $\hat{y}$, é importante lembrar que os dados de entrada x, que serão transformados em y, e a saída esperada $\hat{y}$ já estão determinados pelos dados. Portanto, de maneira geral,

podemos expressar y como uma função dos pesos do modelo w, ou seja, $y = f(w)$. Isso nos permite também representar a função de perda como $L(w)$.

3.2.2 Otimizador

Assim como a função de perda, há uma série de otimizadores comumente utilizados, entre os quais o *Adaptive Moment Estimation* (Adam), que foi escolhido como o otimizador para este trabalho. O algoritmo do Adam utiliza a função de perda $L(w_t)$ na iteração t, calculando tanto o primeiro gradiente $\nabla L(w_t)$ quanto o segundo gradiente $\nabla^2 L(w_t)$. Além disso, o método requer a definição de alguns parâmetros, incluindo os coeficientes de decaimento de primeiro e segundo momento, denotados como β_1 e β_2. Neste trabalho, os valores usados por padrão para esses coeficientes são $\beta_1 = 0.9$ e $\beta_2 = 0.999$. Com base nesses parâmetros, o algoritmo calcula o primeiro momento m_t conforme a equação (3.22) e o segundo momento v_t conforme a equação (3.23).

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla L(w_t) \tag{3.22}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \nabla^2 L(w_t) \tag{3.23}$$

Normalmente, como não há informações prévias para calcular os momentos, os valores iniciais dos momentos são definidos como $m_1 = v_1 = 0$, e esses valores são atualizados ao longo das iterações. Em seguida, são aplicadas as correções de viés, resultando no primeiro momento corrigido $\hat{m}_t$ conforme a equação (3.24) e no segundo momento corrigido $\hat{v}_t$ conforme a equação (3.24).

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.24}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.25}$$

Finalmente, os novos pesos w_{t+1} conforme a equação (3.27), esses são calculados com base nos momentos corrigidos, na taxa de aprendizado η, e o parâmetro ϵ, que é responsável por evitar a divisão por zero. Neste trabalho, o otimizador utiliza $\eta = 1^{-3}$ e $\epsilon = 1^{-7}$.

$$\Delta w_t = -\eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.26}$$

$$w_{t+1} = w_t + \Delta w_t \tag{3.27}$$

No processo de treinamento, uma época começa e, em seguida, o processamento ocorre por lotes. Durante esse processo, os pesos do modelo são atualizados com base nos resultados dos pesos aplicados ao lote anterior. Quando a época atual é concluída, uma nova época se inicia, e os dados são aleatoriamente embaralhados para formar novos lotes, diferentes dos usados na época anterior. Essa prática visa melhorar a generalização e evitar que a precisão do modelo dependa de uma ordem de apresentação específica dos dados.

3.2.3 Função de Precisão

Assim como a função de perda e no otimizador, a métrica de precisão também oferece uma variedade de funções comumente utilizadas, entre as quais escolhemos a *Categorical Accuracy* para este trabalho. O objetivo desta métrica é avaliar a precisão das previsões do modelo. Ela é uma métrica que não é diretamente usada no processo de otimização. Neste trabalho, estamos interessados em calculá-la em um conjunto de validação, ou seja, um conjunto de dados que não é usado para ajustar o modelo.

A cada época, é ajustado um modelo capaz de mapear os dados de entrada de validação x para uma saída de previsão y. Neste modelo em particular, a saída de previsão deve ser bidimensional, com M representando o tamanho do conjunto de validação e N o número de rótulos. Além disso, também temos os dados reais de validação $\hat{y}$. Para cada um dos M conjuntos de dados, tanto para y quanto para $\hat{y}$, determinamos os N índices com maior probabilidade usando as equações (3.28) e (3.29), respectivamente.

$$\text{argmax}(y_i) = \left\{ j \mid y_{i,j} = max(y_i), 1 \leq j \leq N \right\} \tag{3.28}$$

$$\text{argmax}(\hat{y}_i) = \left\{ j \mid \hat{y}_{i,j} = max(\hat{y}_i), 1 \leq j \leq N \right\} \tag{3.29}$$

Além disso, é preciso definir se a predição está ou não correta, tornando possível calcular qual a razão de predições corretas em relação a quantidade de predições totais, para isso se compara o índice máximo de cada um dos M dados de y e de $\hat{y}$ calculando-se então uma indicadora I_i a partir da equação (3.30).

$$I_i = \begin{cases} 1, \text{ se } \text{argmax}(y_i) = \text{argmax}(\hat{y}_i) \\ 0, \text{ c.c} \end{cases} \tag{3.30}$$

Finalmente, a precisão é calculada como a razão entre a soma de I_i e a quantidade total de previsões, que corresponde ao tamanho M do conjunto de dados. A precisão A é obtida pela seguinte equação (3.31).

$$A = \frac{1}{M} \cdot \sum_{i=1}^{M} I_i \tag{3.31}$$

Como critério adotado para a determinação do modelo, podemos definir $A^{(t)}$ como a precisão na época t. Portanto, a época T em que a maior precisão nos dados de validação é alcançada, ou seja, $A^{(T)} = \max(A^{(t)})$, será a época que determinará os pesos do modelo final.

Capítulo 4

Conclusão

4.1 Analise de resultados

Após a criação e treinamento do modelo, analisar os resultados é essencial para a avaliação e compreensão do desempenho e a eficácia do modelo. Além disso, a interpretação dos resultados serve de apoio para identificar as limitações e possíveis melhorias.

Inicialmente, é importante avaliar o desempenho do modelo durante o treinamento. Para isso, analisamos as métricas ao longo de cada época. A partir da observação do gráfico da função de perda no conjunto de treinamento (4.1), nota-se que a perda apresenta uma queda acentuada nas primeiras 50 épocas. Posteriormente, a função de perda continua a diminuir, porém, de maneira mais gradual, estendendo-se até aproximadamente a época 150, após isso, notamos oscilações mais pronunciadas entre o crescimento e a diminuição da perda, sugerindo fortemente que o modelo está próximo do seu limite.

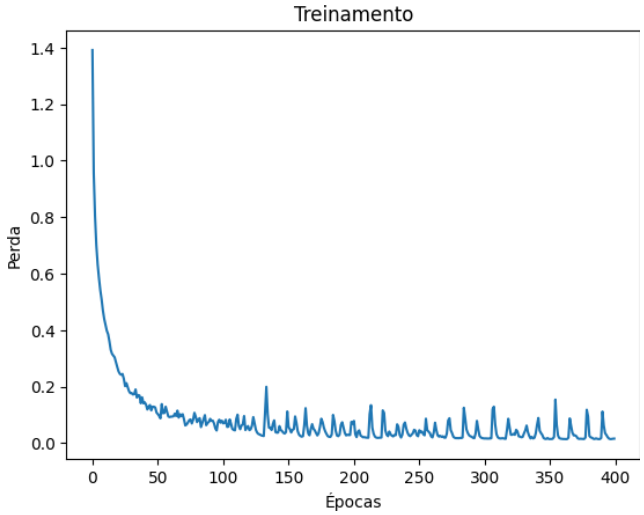

Figura 4.1: *Função de perda no conjunto de treinamento*

Outra métrica importante de se analisar graficamente é a precisão no conjunto de validação através do gráfico (4.2) onde é possível notar um crescimento abrupto nas primeiras

50 épocas. Em seguida a precisão ainda cresce porém de forma mais suavizada até aproximadamente a época 200 onde encontra o seu pico que indica a época que define os pesos do modelo final. Após essa época a precisão oscila mas sem crescer significativamente.

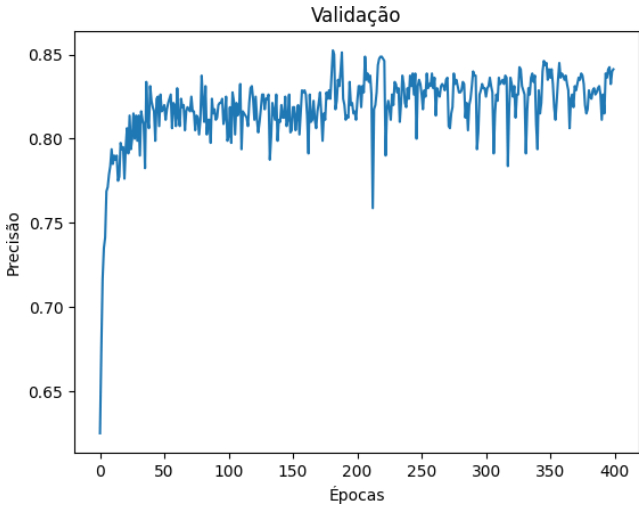

Figura 4.2: *Função de precisão no conjunto de validação*

Após definir o modelo, analisamos a precisão para cada conjunto. No conjunto de treinamento com 6400 dados, sendo 800 de cada comando, a alta precisão de 99.08% era esperada. A matriz de confusão apresentada na figura (4.3) nos permite comparar as categorias reais com as previsões feitas pelo modelo e mostra que a maioria das previsões estava correta, com muito poucos falsos positivos e falsos negativos.

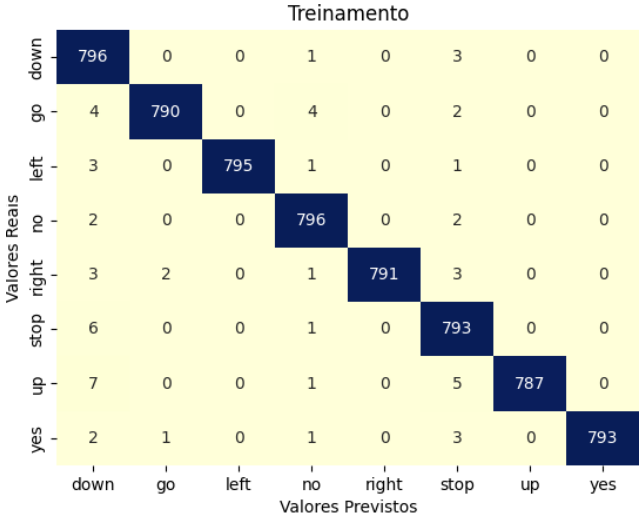

Figura 4.3: *Matriz de confusão do conjunto de treinamento*

No conjunto de validação, composto por 800 dados, sendo 100 de cada comando, alcançou-se uma precisão de 85.25%. Dado o papel crucial desse conjunto na escolha do modelo, essa precisão é a mais alta entre todas as épocas. A matriz de confusão desse conjunto é apresentada na figura (4.4), refletindo que a maioria das previsões foi precisa, com poucos casos de falsos positivos e falsos negativos. No entanto, é notável que o

comando *"down"* obteve a pior precisão, enquanto o comando *"yes"* teve a melhor precisão entre os comandos.

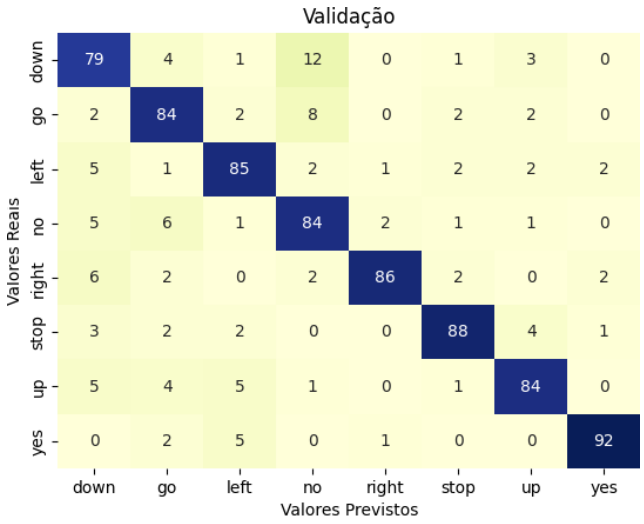

Figura 4.4: *Matriz de confusão do conjunto de validação*

O conjunto de teste, também com 800 dados, 100 de cada comando, é o único que não influencia a definição dos pesos do modelo. Surpreendentemente, a precisão alcançada foi de 85.87% superando a do conjunto de validação, o que indica uma capacidade positiva de generalização. A matriz de confusão figura (4.5) mostra que a maioria das previsões foi precisa, embora o comando *"right"* tenha tido a pior precisão. Similar ao conjunto de validação, o comando *"yes"* obteve a melhor precisão. Também é notável uma maior confusão entre os comandos *"go"* e *"no"*, que pode ser explicado devido à semelhança semântica entre eles.

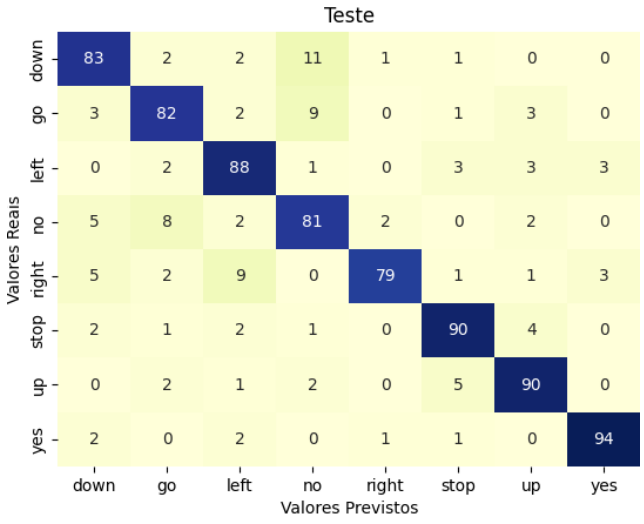

Figura 4.5: *Matriz de confusão do conjunto de teste*

Em linhas gerais, os resultados obtidos destacam a notável capacidade do modelo em categorizar de maneira eficaz, o que demonstra a possibilidade de criação de modelos simplificados e em grande parte independentes, porém altamente precisos. Essa capacidade

é fundamental para o avanço contínuo das áreas de ASR e NLP. A partir disso, surgem diversas oportunidades de aplicabilidade, com potencial real de estimular tecnologias e projetos de um custo relativamente baixo e potencial de impacto socialmente positivo.

4.2 Sugestões e melhorias

Apesar dos resultados positivos, é crucial ponderar sobre a qualidade e seleção dos dados. Nesse contexto, é possível explorar cenários adicionais na construção do modelo. Aumentar o tamanho do conjunto de dados pode resultar em maior complexidade, porém, potencialmente, acarretar em uma perda de eficácia. A inclusão de outro idioma no conjunto de dados é uma opção intrigante, com a expectativa de manter a eficácia. A possibilidade de mesclar idiomas no mesmo conjunto também é interessante, uma vez que as nuances linguísticas podem contribuir para uma eficácia aprimorada. Essas considerações ampliam as perspectivas para futuras investigações, destacando a importância de validações e testes desses cenários para uma compreensão mais aprofundada das capacidades.

Além dos cenários mais abrangentes, é relevante explorar casos específicos dentro do conjunto de dados. Uma avaliação pertinente seria analisar a construção do modelo com palavras homófonas, aquelas que são pronunciadas da mesma maneira, porém escritas de maneira distinta, como *"knight"* e *"night"* ou *"seção"* e *"sessão"*. Nesse contexto, é esperado que o modelo apresente confusão substancial e tenha dificuldade em categorizar com precisão. Portanto buscar maneiras de minimizar o problema seria um assunto interessante.

Outro aspecto a considerar, quando ponderamos sobre as aplicações do modelo, é sua capacidade de interpretar predições de comandos não previamente listados, além de identificar situações de silêncio ou ruído. Uma abordagem interessante para lidar com esses cenários seria categorizá-los de maneira específica, utilizando rótulos como por exemplo, *"_unknown_"*, *"_silence_"*, e *"_noise_"*.

Para os comandos desconhecidos, a proposta seria formar um conjunto diversificado com uma ampla gama de entradas não listadas. Durante o treinamento do modelo, seria possível avaliar os resultados de forma específica nessas categorias. No entanto, se a aplicação não exigir essa especificidade, uma alternativa seria agrupar os três cenários sob uma única classificação, por exemplo, *"_unclassified_"*, simplificando o treinamento e permitindo uma avaliação abrangente dos resultados.

Apêndice A

Implementações de código

A.1 Carregamento dos dados

Como padrão para as implementações desse trabalho a linguagem de programação utilizada foi o Python e para auxílio na criação dos métodos necessários foram usadas principalmente as bibliotecas Numpy, Tensorflow e Matplotlib.

Inicialmente, realizamos uma validação para verificar a existência dos dados no repositório especificado. Se os dados não forem encontrados, prosseguimos com o download da base de dados. Em seguida, listamos todos os arquivos e os ordenamos por comando em ordem alfabética. Para cada comando, aplicamos uma ordenação aleatória aos dados respectivos. Posteriormente, dividimos os dados em conjuntos de treinamento, validação e teste, com proporções de 80%, 10% e 10%, respectivamente. Após a divisão dos dados para cada comando, aplicamos outra ordenação aleatória a cada um dos conjuntos.

É crucial destacar que essas ordenações aleatórias devem produzir sempre o mesmo resultado, pois os conjuntos de treinamento, validação e teste devem permanecer consistentes em cada execução. Para implementar todo esse processo, foram criados métodos na classe `LoadData`, os quais estão detalhados no programa (A.1).

Programa A.1 Organização dos arquivos.

```
1  import os
2  import pathlib
3  import numpy as np
4  import tensorflow as tf
5  AUTOTUNE = tf.data.AUTOTUNE
6
7  class LoadData():
8      def __init__(self):
9          self.commands = self.get_commands()
10         base_data_path = 'files/data'
```

cont ⟶

⟶ *cont*

```
        data_path = f'{base_data_path}/mini_speech_commands'
        self.data_dir = pathlib.Path(data_path)
        if not self.data_dir.exists():
            self.download_data(base_data_path)
        self.train_files, self.validation_files, self.test_files =
         ↪  self.randomize_data()

    def get_commands(self):
        return {0: 'down', 1: 'go', 2: 'left', 3: 'no', 4: 'right', 5:
         ↪  'stop', 6: 'up', 7: 'yes'}

    def download_data(self, base_data_path):
        origin = "http://storage.googleapis.com/download.tensorflow.org/" \
                 "data/mini_speech_commands.zip"
        tf.keras.utils.get_file('mini_speech_commands.zip', origin=origin,
         ↪  extract=True, cache_dir='.', cache_subdir=base_data_path)

    def randomize_data(self):
        file_names = sorted(tf.io.gfile.glob(str(self.data_dir) + '/*/*'))
        quantity_for_commands = 1000
        train_files, validation_files, test_files = [], [], []
        files = [train_files, validation_files, test_files]
        for i in range(0, len(file_names), quantity_for_commands):
            np.random.seed(1)
            files_command = file_names[i:i + quantity_for_commands]
            np.random.shuffle(files_command)
            train_files += files_command[:800]
            validation_files += files_command[800:900]
            test_files += files_command[900:]
        for file_set in files:
            np.random.seed(1)
            np.random.shuffle(file_set)
        return files
```

Para realizar o processo de conversão entre comandos e rótulos, bem como o processo inverso, foram implementados métodos adicionais na classe `LoadData`. Os detalhes dessas implementações estão apresentados no programa (A.2).

Programa A.2 Conversão de comandos.

```
class LoadData():
    def get_label_by_command(self, parts):
        commands_values =
         ↪  tf.constant(list(self.get_commands().values()))
        index = tf.where(tf.equal(commands_values, parts))[0][0]
```

cont ⟶

```
    ⟶ cont
        len_commands = tf.shape(commands_values)[0]
        return tf.one_hot(index, len_commands, dtype=tf.float32)

    def get_command_by_label(self,one_hot):
        one_hot_np = one_hot.numpy()
        index = np.argmax(one_hot_np)
        return tf.convert_to_tensor(self.get_commands()[index])

    def get_label(self, file_path):
        parts = tf.strings.split(input=file_path, sep=os.path.sep)
        return self.get_label_by_command(parts[-2])
```

No processo de carregamento dos arquivos de áudio e sua conversão para o formato de onda e posteriormente para espectrograma, também são implementados métodos na classe `LoadData`. Os detalhes dessas implementações estão disponíveis no programa referenciado (A.2).

Programa A.3 Carregamento de áudio

```
class LoadData():
    def decode_audio(self, audio_binary):
        audio, _ = tf.audio.decode_wav(contents=audio_binary)
        return tf.squeeze(audio, axis=-1)

    def get_spectrogram(self, waveform):
        input_len = 16000
        waveform = waveform[:input_len]
        zero_padding = tf.zeros([16000] - tf.shape(waveform),
        ↪  dtype=tf.float32)
        waveform = tf.cast(waveform, dtype=tf.float32)
        equal_length = tf.concat([waveform, zero_padding], 0)
        spectrogram = tf.signal.stft(equal_length, frame_length=255,
        ↪  frame_step=128)
        spectrogram = tf.abs(spectrogram)
        return spectrogram
```

Finalmente, com todos os métodos construídos, os dados podem ser carregados usando o formato de `Dataset`. Para isso, é necessário definir métodos na classe `LoadData` que associem cada espectrograma ao seu respectivo rótulo e como os conjuntos de treinamento, validação e teste serão divididos. A implementação para carregar cada conjunto pode ser encontrada no programa (A.4).

Programa A.4 Métodos de carregamento de conjuntos

```
class LoadData():
    def get_stft_and_label(self, file_path):
        label = self.get_label(file_path)
        audio_binary = tf.io.read_file(file_path)
        waveform = self.decode_audio(audio_binary)
        spectrogram = self.get_stft(waveform)
        return spectrogram, label

    def create_ds_batch(self, files, batch_size):
        data = tf.data.Dataset.from_tensor_slices(files)\
            .map(map_func=self.get_stft_and_label,
            num_parallel_calls=AUTOTUNE)
        return data.batch(batch_size=batch_size) if batch_size is not
         ↪ None else data

    def get_data_train(self, batch_size):
        train_files = self.train_files
        train = self.create_ds_batch(files=train_files,
         ↪ batch_size=batch_size)
        return train

    def get_data_validation(self, batch_size):
        val_files = self.validation_files
        validation = self.create_ds_batch(files=val_files,
         ↪ batch_size=batch_size)
        return validation

    def get_data_test(self, batch_size):
        test_files = self.test_files
        test = self.create_ds_batch(files=test_files,
         ↪ batch_size=batch_size)
        return test_ds_batch(files=test_files, batch_size=batch_size)
        return test
```

A.2 Criação e treinamento do modelo

Implementar computacionalmente toda a construção e responsabilidade das camadas é uma tarefa complexa e desafiadora, porém a biblioteca TensorFlow disponibiliza implementações prontas e métodos para reunir camadas e criar arquiteturas. Para isso foi necessário apenas a construção de uma classe `EncoderOnlyModel` que tem relação de herança da classe `Model` onde serão definidas as camadas utilizadas e como as mesmas serão conectadas. Toda essa implementação é definida no programa (A.5).

Programa A.5 Arquitetura do modelo

```
import tensorflow as tf
from keras import layers

class EncoderOnlyModel(tf.keras.Model):
    def __init__(self, target_vocab_size, num_heads=2, d_model=128,
     ↪  dff=512, rate=0.1):
        super(EncoderOnlyModel, self).__init__()
        self.d_model = d_model
        self.enc_layers = tf.keras.Sequential([layers.Dense(d_model,
         ↪  activation='relu')])
        self.mha = layers.MultiHeadAttention(num_heads=num_heads,
         ↪  key_dim=d_model)
        self.ffn = tf.keras.Sequential([ layers.Dense(dff),
         ↪  layers.Dense(d_model, activation='relu') ])
        self.layer_norm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layer_norm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate=rate)
        self.dropout2 = layers.Dropout(rate=rate)
        self.final_layer = layers.Dense(target_vocab_size)
        self.global_average_pooling = layers.GlobalAveragePooling1D()

    def call(self, inputs, training=True):
        inputs = self.enc_layers(inputs)
        outputs = self.mha(inputs, inputs)
        outputs = self.dropout1(outputs, training=training)
        inputs += outputs
        inputs = self.layer_norm1(inputs)
        outputs = self.ffn(inputs)
        # Dropout and residual connection
        outputs = self.dropout2(outputs, training=training)
        inputs += outputs
        inputs = self.layer_norm2(inputs)
        outputs = self.final_layer(inputs)
        outputs = self.global_average_pooling(outputs)
        probabilities = tf.nn.softmax(outputs)
        return probabilities
```

Assim como a construção da arquitetura das camadas, a implementação computacional da aplicação desses métodos e dos cálculos de treinamento é uma tarefa complexa e desafiadora, mas pode ser simplificada pelo uso da biblioteca TensorFlow. A construção da função de perda, do otimizador e da função de precisão será realizada usando, respectivamente, as classes `CategoricalCrossentropy`, `Adam`, e `CategoricalAccuracy`.

Além disso, é necessário configurar alguns passos adicionais, como especificar o diretó-

rio onde os pesos do modelo serão armazenados e o critério para salvar esses pesos. Como discutido anteriormente, o critério escolhido é a época com a maior precisão no conjunto de validação. O gerenciamento desse processo é realizado pela classe `ModelCheckpoint`, com base nos critérios previamente definidos. Durante o processo, também verificamos se há algum arquivo de pesos previamente salvos. Se esses pesos já existirem, eles são carregados para que o modelo possa continuar a partir do ponto de parada anterior.

Adicionalmente, para permitir uma inspeção mais prática dos resultados após cada época, foi criada a classe personalizada `DisplayOutputs`, baseada na classe `Callback`. O objetivo dessa classe é exibir, ao final de cada época, a aplicação do modelo a um único lote do conjunto de teste. A implementação da classe está detalhada no programa A.6.

Programa A.6 Inspeção parcial de resultado

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

class DisplayOutputs(keras.callbacks.Callback):
    def __init__(self, batch, commands):
        self.batch = batch
        self.commands = commands

    def on_epoch_end(self, epoch, logs=None):
        audio = self.batch[0]
        target_label = self.batch[1].numpy()
        batch_size = tf.shape(audio)[0]
        predicted_label = self.model.predict(audio)
        for i in range(batch_size):
            command_target = self.commands[np.argmax(target_label[i,
             ↪  :])]
            command_predicted =
             ↪  self.commands[np.argmax(predicted_label[i, :])]
            print(f"Target: {command_target}\nPrediction:
             ↪  {command_predicted}\n")
```

Assim, a cada época de treinamento, os resultados do treinamento e a comparação entre a predição em um lote do conjunto de teste e a representação real são exibidos no terminal, como mostrado no seguinte exemplo:

```
Epoch 1/400
1/1 [==============================] - 0s 240ms/step
Target: go
Prediction: go

Target: yes
Prediction: yes
```

```
Target: no
Prediction: no

Target: yes
Prediction: yes

200/200 [==============================] - 22s 97ms/step - loss: 0.8589 -
categorical_accuracy: 0.7142 - val_loss: 0.8429 - val_categorical_accuracy: 0.7287
```

Com todas as definições estruturais estabelecidas, passamos então à configuração dos hiperparâmetros, que incluem a quantidade de épocas, o tamanho dos lotes, as dimensões do modelo, o número de *heads* e a taxa de *dropout*. Essas definições são baseadas em todas as especificações necessárias. A implementação geral da fase de treinamento desse trabalho está feita na classe `Train` e pode ser visualizada no programa (A.7).

Programa A.7 Treinamento do modelo

```
import os
from LoadData import LoadData
import matplotlib.pyplot as plt
from keras.optimizers import Adam
from DisplayOutputs import DisplayOutputs
from keras.callbacks import ModelCheckpoint
from EncoderOnlyModel import EncoderOnlyModel
from keras.metrics import CategoricalAccuracy
from keras.losses import CategoricalCrossentropy

batch_size, test_batch_size = 32, 4
epochs, num_heads, d_model, dff, dropout_rate = 400, 2, 128, 512, 0.1

class Train():
    def __init__(self):
        self.load_data = LoadData()

    def train_model(self):
        train = self.load_data.get_data_train(batch_size=batch_size)
        validation =
        ↪ self.load_data.get_data_validation(batch_size=batch_size)
        test =
        ↪ self.load_data.get_data_test(batch_size=test_batch_size)
        batch_test = next(iter(test))

        model = EncoderOnlyModel(num_heads=num_heads, d_model=d_model,
        ↪ dff=dff,
        ↪ target_vocab_size=len(self.load_data.get_commands()),
        ↪ rate=dropout_rate)
```

cont ⟶

⟶ *cont*

```
25
26          checkpoint_path = "files/checkpoints"
27          os.makedirs(checkpoint_path, exist_ok=True)
28          checkpoints = os.listdir(checkpoint_path)
29          checkpoint = checkpoints[0] if len(checkpoints) > 0 else None
30
31          model.build((None, 124, 129))
32          if checkpoint:
33              model.load_weights(os.path.join(checkpoint_path,
                ↪  checkpoint))
34
35          model.compile(optimizer=Adam(), loss=CategoricalCrossentropy(),
            ↪  metrics=[CategoricalAccuracy()])
36
37          display_cb = DisplayOutputs(batch=batch_test,
            ↪  commands=self.load_data.get_commands())
38          ckpt_cb =
            ↪  ModelCheckpoint(filepath=f'{checkpoint_path}/model.h5',
            ↪  save_best_only=True, save_weights_only=True,
            ↪  monitor='val_categorical_accuracy', mode='max',
            ↪  save_freq='epoch')
39
40          history = model.fit(train, validation_data=validation,
            ↪  callbacks=[display_cb, ckpt_cb], epochs=epochs)
41
42          plt.figure()
43          plt.plot(history.history['loss'], label='Train Loss')
44          plt.title("Treinamento")
45          plt.xlabel('Épocas')
46          plt.ylabel('Perda')
47          plt.savefig(f'files/report/train_loss.png', format='png',
            ↪  bbox_inches='tight', pad_inches=0)
48          plt.clf()
49          plt.plot(history.history['val_categorical_accuracy'],
            ↪  label='Validation Accuracy')
50          plt.title("Validação")
51          plt.xlabel('Épocas')
52          plt.ylabel('Precisão')
53          plt.savefig(f'files/report/validation_accuracy.png',
            ↪  format='png', bbox_inches='tight', pad_inches=0)
54          plt.clf()
55
56  Train().train_model()
```

Normalmente, a definição dos valores dos hiperparâmetros é baseada em observações, ou seja, avaliando os efeitos que as alterações têm nas métricas durante a fase de treinamento. No início dos treinamentos deste trabalho, os resultados já eram significativamente satisfatórios. No entanto, foram realizados alguns testes que indicaram melhorias ao aumentar o número de épocas e *heads*, e ao reduzir o tamanho dos lotes. No entanto, é importante notar que, devido à complexidade do problema, não é garantido que aumentar ou diminuir ainda mais esses parâmetros resultará em um desempenho melhor. Em alguns casos, pode até tornar o treinamento computacionalmente inviável.

O treinamento do modelo com os parâmetros e conjuntos de dados mencionados demandou aproximadamente duas horas. No entanto, esse tempo pode variar dependendo das capacidades do computador utilizado para a execução do processo. As especificações técnicas do computador em questão são as seguintes:

Processador: AMD Ryzen 5 5500 3.6GHz

Memória RAM: 16 GB DDR4

Placa de Vídeo: NVIDIA GeForce GTX 1660Ti 6 GB

A.3 Geração de resultados

Para a análise dos resultados, também é utilizada a biblioteca Tensorflow e também Matplotlib, utilizadas para carregar os pesos do modelo treinado previamente. Em seguida, as previsões e os valores de precisão são calculados, juntamente com a matriz de confusão dos conjuntos de dados. Os resultados obtidos são salvos em arquivos. Nesse trabalho, esse papel é feito pela classe `Report`, cujo código pode ser visualizado no programa (A.8).

Programa A.8 Treinamento do modelo

```
import os
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from LoadData import LoadData
from keras.optimizers import Adam
from sklearn.metrics import confusion_matrix
from EncoderOnlyModel import EncoderOnlyModel
from keras.metrics import CategoricalAccuracy
from keras.losses import CategoricalCrossentropy

len_train, len_validation, len_test = 6400, 800, 800
num_heads, d_model, dff, dropout_rate = 2, 128, 512, 0.1

class Report():
    def __init__(self):
```

cont ⟶

⟶ *cont*

```
17          self.load_data = LoadData()
18
19      def generate_report(self):
20          train_input, train_output =
             ↪  self.load_data.get_data_train(len_train).as_numpy_iterator().next()
21          validation_input, validation_output =
             ↪  self.load_data.get_data_validation(
22              len_validation).as_numpy_iterator().next()
23          test_input, test_output =
             ↪  self.load_data.get_data_test(len_test).as_numpy_iterator().next()
24
25          model = EncoderOnlyModel(num_heads=num_heads, d_model=d_model,
             ↪  dff=dff,target_vocab_size=len(self.load_data.get_commands()),
             ↪  rate=dropout_rate)
26
27          model_path = "files/checkpoints"
28          model_weights_file = os.listdir(model_path)[0]
29          model.build((None, None, 129))
30          model.load_weights(os.path.join(model_path,
             ↪  model_weights_file))
31          model.compile(optimizer=Adam(), loss=CategoricalCrossentropy(),
             ↪  metrics=[CategoricalAccuracy()])
32
33          train_precision = model.evaluate(x=train_input,
             ↪  y=train_output)[1]
34          validation_precision = model.evaluate(x=validation_input,
             ↪  y=validation_output)[1]
35          test_precision = model.evaluate(x=test_input,
             ↪  y=test_output)[1]
36
37          train_predictions = np.argmax(model.predict(train_input),
             ↪  axis=1)
38          validation_predictions =
             ↪  np.argmax(model.predict(validation_input), axis=1)
39          test_predictions = np.argmax(model.predict(test_input),
             ↪  axis=1)
40
41          train_confusion =
             ↪  np.array(confusion_matrix(np.argmax(train_output, axis=1),
             ↪  train_predictions))
42          validation_confusion =
             ↪  np.array(confusion_matrix(np.argmax(validation_output,
             ↪  axis=1), validation_predictions))
```

cont ⟶

⟶ *cont*

```
43          test_confusion =
             ↪  np.array(confusion_matrix(np.argmax(test_output, axis=1),
             ↪  test_predictions))
44
45          fig = plt.figure()
46          report_path = "files/report"
47          os.makedirs(report_path, exist_ok=True)
48          report = f';Treinamento;Validação;Teste;Geral\nPrecisão;' \
49                   f'{np.mean(train_precision):.4f};' \
50                   f'{np.mean(validation_precision):.4f};' \
51                   f'{np.mean(test_precision):.4f}'
52          with open(os.path.join(report_path, "report.csv"), 'w') as
             ↪  file:
53              file.write(report)
54
55          self.save_confusion(tittle="Treinamento",
             ↪  confusion=train_confusion)
56          self.save_confusion(tittle="Validação",
             ↪  confusion=validation_confusion)
57          self.save_confusion(tittle="Teste", confusion=test_confusion)
58
59      def save_confusion(self, tittle, confusion):
60          sns.heatmap(confusion, annot=True, fmt='d', cmap='YlGnBu',
             ↪  cbar=False)
61          plt.title(tittle)
62          plt.xlabel(f'Valores Previstos')
63          plt.ylabel(f'Valores Reais')
64          class_names = self.load_data.get_commands().values()
65          tick_marks = np.arange(len(class_names))
66          plt.xticks(tick_marks + 0.5, class_names)
67          plt.yticks(tick_marks + 0.5, class_names)
68
             ↪  plt.savefig(f'files/report/confusion-{str(tittle).lower()}.png',
             ↪  format='png', bbox_inches='tight',
69                         pad_inches=0)
70          plt.clf()
71
72  Report().generate_report()
```

Referências

[Furui 2005] Sadaoki Furui. “50 years of progress in speech and speaker recognition”. *SPECOM 2005, Patras* (2005), pp. 1–9 (citado na pg. 1).

[Peixoto *et al.* 2015] João Pedro Peixoto, Raiane Borges Souza Peixoto, Vinicius Pinheiro do Nascimento, Leandro Marques Samyn e Carlos Eduardo Pantoja. “Desenvolvimento de uma cadeira de rodas acionada por comandos de voz”. *Anais do Computer on the Beach* (2015), pp. 557–558 (citado na pg. 1).

[Shafran *et al.* 2003] Izhak Shafran, Michael Riley e Mehryar Mohri. “Voice signatures”. In: *2003 IEEE workshop on automatic speech recognition and understanding (IEEE Cat. No. 03EX721)*. IEEE. 2003, pp. 31–36 (citado na pg. 3).

[Sivanandam e Paulraj 2009] S Sivanandam e M Paulraj. *Introduction to artificial neural networks*. Vikas Publishing House, 2009 (citado na pg. 9).

[Vaswani *et al.* 2017] Ashish Vaswani *et al.* “Attention is all you need”. *Advances in neural information processing systems* 30 (2017) (citado na pg. 9).