PAULO HENRIQUE SILVA DE ARRUDA

COMPRESSÃO HORIZONTAL DE SEQUÊNCIAS GENÔMICAS SEM PERDA DE DADOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

CURITIBA
2021

PAULO HENRIQUE SILVA DE ARRUDA

COMPRESSÃO HORIZONTAL DE SEQUÊNCIAS GENÔMICAS SEM PERDA DE DADOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Edson Emílio Scalabrin

CURITIBA
2021

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Pamela Travassos de Freitas – CRB 9/1960

A779c 2021

Arruda, Paulo Henrique Silva de
Compressão horizontal de sequências genômicas sem perda de dados / Paulo Henrique Silva de Arruda ; orientador: Edson Emílio Scalabrin.– 2021.
59 f. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2021
Bibliografia: f. 56-59

1. Compressão de dados (Computação). 2. DNA. 3. Genoma humano. I. Scalabrin, Edson Emílio. II. Pontifícia Universidade Católica do Paraná. Pós-Graduação em Informática. III. Título.

CDD 22. ed. – 005.746

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

65-2021

DECLARAÇÃO

Declaro para os devidos fins que o aluno **PAULO HENRIQUE SILVA DE ARRUDA**, defendeu sua dissertação de Mestrado intitulada **"COMPRESSÃO HORIZONTAL DE SEQUÊNCIAS GENÔMICAS SEM PERDA DE DADOS"**, na área de concentração Ciência da Computação, no dia 14 de maio de 2021, no qual foi aprovado.

Declaro ainda que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade, firmo a presente declaração.

Curitiba, 06 de setembro de 2021.

Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática
Pontifícia Universidade Católica do Paraná

Agradecimentos

Agradeço em primeiro lugar a Deus que iluminou o meu caminho durante esta caminhada, ao meu pai Pedro, minha mãe Maria. Agradeço também as minhas tias, Lenir e Regina, por sempre me incentivarem a continuar no processo e pelas ajudas que deram no decorrer da minha vida, sou muito grato.

Também quero agradecer ao meu orientador, Prof. Edson, por todo apoio, incentivo e paciência que teve comigo durante este processo, sou muito grato por tudo que o senhor fez por mim, muito obrigado. Também agradeço à Pontifícia Universidade Católica do Paraná e seu corpo docente por tornar possível este projeto em parceria com a CAPES pelo suporte financeiro.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Dedico este trabalho ao meu Pai e minha Mãe.

Resumo

A leitura de dados genômicos foi possível por meio do esforço de pesquisa internacional, denominada Projeto Genoma Humano, iniciada em 1990 e concluída em abril de 2003, o primeiro sequenciamento do genoma humano custou mais de US $3 bilhões. Entretanto, como resultado, uma grande quantidade de dados genômicos é produzida, tornando o armazenamento e transferência um grande desafio em termos de volume de dados genéticos. Uma das alternativas viáveis para solucionar esse problema é a compressão de dados, que se mostra adequada para reduzir requisitos de armazenamento e transferência. Com o passar dos anos foram sugeridas vários algoritmos e ferramentas especializadas na compressão de dados genômicos, por exemplo, DELIMINATE, MFCOMPRESS e COMRAD. Classicamente, o aprendizado profundo mostrou um desempenho significativamente aprimorado em problemas complexos de classificação e regressão. Neste contexto, surgiu uma ferramenta com o foco em compressão de dados denominada DeepZip. O DeepDNA é uma proposta de arquitetura baseada no DeepZip. Neste contexto, o objetivo principal desta dissertação é apresentar uma nova abordagem para o processo de compressão de sequências genômicas, sem perda de dados, baseado em ferramentas de predição do próximo caractere de uma sequência, denominado DeepRLE. O resultado obtido foi uma taxa de economia de espaço de 94.85% e de 0.0515 bpb. Deve-se notar, entretanto, que nosso algoritmo se limita apenas a sequências mitocondriais.

Palavras-chave: compressão de dados, sem perda, DNA, sequência genômica, aprendizado profundo, LSTM, CNN.

Abstract

Reading genomic data was made possible through the international research effort, called the Human Genome Project, started in 1990 and completed in April 2003. The first sequencing of the human genome cost more than $3 billion. As result of such accomplishment, a large amount of genomic data is being produced, making storage and transfer a major challenge in terms of the volume of genetic data. One of the viable alternatives to solve this problem is data compression, which is adequate to reduce storage and transfer requirements. Over the years, several algorithms and specialized tools for compressing genomic data have been suggested, for example, DELIMINATE, MFCOMPRESS and COMRAD. Classically, deep learning has significantly improved performance on complex classification and regression problems. In this context, a tool focused on data compression called DeepZip emerged. DeepDNA is an architecture proposal based on DeepZip. In this context, the main objective of this dissertation is to present a new approach for the compression process of genomic sequences, without data loss, based on prediction tools to foresee the next character of a sequence, called DeepRLE. The result obtained was a space saving rate of 94.85% and 0.0515 bpb. It is important to highlight, however, that the algorithm is limited to mitochondrial sequences only.

Keywords: Compression lossless, DNA, compressing genomic data, Deep Learning, LSTM, CNN.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

A leitura de dados genômicos foi possível por meio do esforço de uma pesquisa internacional, denominada Projeto Genoma Humano (HGP - Human Genome Project), iniciada em 1990 e concluída em abril de 2003, o primeiro sequenciamento do genoma humano custou mais de US $3 bilhões (SCHUSTER, 2007). Desde então, a velocidade de sequenciamento cresceu exponencialmente, até excedeu a Lei de Moore (REUTER; SPACEK; SNYDER, 2015). Tal avanço dos pesquisadores, contribuiu para o desenvolvimento de tecnologias de sequenciamento genético (SCHUSTER, 2007), permitindo que genomas humanos possam ser sequenciados em poucas horas e com custo de menos de US $1.000. Entretanto, como resultado, uma grande quantidade de dados genômicos serão produzidos, tornando o armazenamento e transferência do grande volume de dados genéticos um problema.

Uma das alternativas viáveis para solucionar esse problema é a compressão de dados, que se mostra adequada para reduzir requisitos de armazenamento. Isso possibilita transferir e armazenar sequências genéticas de forma mais eficiente, ocupando menos espaço em memória e disco, e com um custo mais baixo. Com o passar dos anos foram sugeridas vários algoritmos e ferramentas especializadas na compressão de dados genômicos, onde são categorizadas em duas abordagens, sendo elas a abordagem vertical ou a abordagem horizontal, na abordagem vertical, utiliza-se de um DNA de referência para o processo de compressão, enquanto na compressão horizontal, utiliza-se de parte do mesmo DNA para o processo de compressão.

Em GRUMBACH; TAHI, (1993) foi apresentado a primeira aplicação com o propósito especial para a compressão de dados, denominada *Biocompress*, o *Biocompress* tenta encontrar as repetições exatas e palindrômicas na sequência de destino em uma janela de tamanho fixo anterior, e então, essas repetições exatas e palindrômicas são codificadas usando a posição e a duração de suas primeiras ocorrências. Em BIJI; NAIR, (2017) foi proposto um conjunto de sequências genômicas com intuito de avaliar o desempenho das seguintes ferramentas: DELIMINATE, MFCOMPRESS e COMRAD.

Em Martins et al. (2018), foi proposta uma nova perspectiva para a compressão de sequências genômicas, sem perdas, utilizando formatos de imagem, onde sequências de DNA armazenadas em um arquivo FASTA são convertidas em uma matriz de símbolos do alfabeto genômico {A, T, C e G}. A matriz então é codificada nos formatos de arquivo de imagem WEBP e FLIF para a avaliação do desempenho.

Nos últimos anos, o aprendizado profundo, do inglês Deep Learning, tem sido utilizado em diversos campos de pesquisa, como reconhecimento de fala, classificação de imagens, condução autônoma e processamento de linguagem natural. O aprendizado profundo mostrou um desempenho significativamente aprimorado em problemas complexos de classificação e regressão. Neste contexto, surgiu uma ferramenta com o foco em compressão de dados denominada DeepZip, (GOYAL et al., 2019). Ela utiliza método de aprendizado profundo para compressão de texto, onde é previsto o próximo caractere e/ou palavra com base em uma rede neural recorrente (LSTM/GRU) com uma única passagem pelos dados.

Já WANG et al., (2018) foi proposta uma nova arquitetura baseada no DeepZip, denominado DeepDNA. Ela tem por objetivo comprimir genomas mitocondriais humanos utilizando uma rede neural convolucional (CNN) e a rede de memória de curto e longo prazo (LSTM). A combinação de CNN e LSTM gerou bons resultados em tarefas de visão computacional como classificação de texto (ZHOU et al., 2015), legenda de imagem (VINYALS et al., 2015) e reconhecimento da fala (SAINATH et al., 2015). O modelo CNN é a primeira camada usada para capturar as características locais da sequência. A seguinte camada LSTM é usada para capturar a sequência de recursos de longo prazo, e uma camada totalmente achatada é conectada para reunir as informações de saída LSTM. Por último, uma camada final realiza uma transformação sigmóide não linear que serve como previsões probabilísticas do genoma base de sequência.

O objetivo principal desta dissertação é apresentar uma nova abordagem para o processo de compressão de sequências genômicas, sem perda de dados, baseado em ferramentas de predição do próximo caractere de uma sequência. Vale ressaltar que neste estudo, não será discutido o processo de descompressão das sequências genômicas com a abordagem proposta.

1.1 Objetivos

O objetivo principal desta dissertação, é apresentar uma nova abordagem para o processo de compressão de sequências genômicas, sem perda de dados, baseada em um método de predição do próximo caractere. Para que esse objetivo seja alcançado, um conjunto de objetivos específicos foram estabelecidos, a fim de proporcionar uma melhor compreensão do tema. Esses objetivos consistem em:

- identificar um conjunto de ferramentas de compressão, existentes na literatura, com foco na predição do próximo caractere;
- selecionar um conjunto de dados e aplicar testes de *performance* sobre os mesmos; e
- comparar a performance da compressão baseada em predição e compressão aritmética clássica.

1.2 Organização

Esta dissertação está organizada da seguinte forma: O Capítulo 2 apresenta a fundamentação teórica necessária para o entendimento sobre compressão de dados genéticos, aprendizagem profunda, onde escrever-se-á sobre CNN e LSTM, codificação aritmética e técnicas de transformações. O Capítulo 3 apresenta as seleções de métricas, *dataset* e a arquitetura do DeepRLE. O Capítulo 4 apresenta os experimentos realizados e os resultados obtidos e, por fim, no Capítulo 5 são apresentadas as conclusões da pesquisa.

2 Fundamentação Teórica

Esta seção tem por objetivo descrever os conceitos fundamentais para a compreensão desta dissertação. Assumindo que o crescimento dos dados genômicos é uma tendência cada vez mais presente, e que a compressão de dados é uma de suas soluções, na Seção 2.1, são abordados conceitos relacionados à compressão de dados, mais especificamente no que tange compressão de dados genômicos. Na Seção 2.2, é abordado o conceito de aprendizagem profunda, bem como alguns algoritmos de apoio. Na Seção 2.3, são abordados conceitos sobre a codificação aritmética. Por fim, na Seção 2.4, são abordadas as técnicas de transformações.

2.1 Compressão de dados genômicos

A compressão de dados é o processo de conversão de um fluxo de dados de entrada, o fluxo de origem ou os dados originais, em outro fluxo de dados que possui um tamanho menor. Esse processo pode ser aplicado a vários formatos de dados, como texto, imagens, áudios, entre outros, com objetivo de reduzir custos e aumentar a eficiência na manutenção de grandes volumes de dados (SALOMON, 2007).

Atualmente, existem dois tipos principais de compressão de dados: a compressão com perda de informação (do inglês, *lossy compression*) e compressão sem perda de informação (do inglês, *lossless compression*). Na compressão com perda, quando os dados passam pelo processo de compressão e descompressão, o resultado tende a diferir do arquivo original, devido à perda de parte dos dados no meio do processo. Em casos em que a perda de dados é pequena, o ser humano pode não conseguir perceber a diferença. Segundo SALOMON (2007), a compressão com perda tende a alcançar melhores taxas de compressão devido às perdas de informações. Por sua vez, a compressão sem perda preserva os dados de modo que, após o

processo de descompressão, o arquivo é idêntico ao arquivo de entrada da compressão, visto que eles exploram a redundância dos dados a serem comprimidos (SALOMON, 2007).

Durante o processo de compressão de dados, os algoritmos de compressão estão concentrados em realizar uma dentre duas ações (SALOMON, 2007). A primeira, a transformação dos dados, basicamente consiste em reduzir o número de símbolos únicos contidos em uma sequência genômica, a ser comprimido (na Seção 2.4 o tema será aprofundado). A segunda ação é a fase de codificação, que ocorre após a fase de transformação, se ela estiver presente no processo, que basicamente é a junção de um ou mais algoritmos no processo de reescrita dos dados de entrada em binário.

Com o crescimento dos dados genômicos, ferramentas especializadas em compressão de dados genômicos se fizeram necessárias. Como consequência de estudos realizados sobre o tema de compressão de sequências genômicas, algumas características foram notadas, sendo elas: o alfabeto de tamanho pequeno, as repetições e os palíndromos.

A Figura 1 ilustra outra característica que pode ser considerada. Uma sequência genômica possui dupla fita, ou seja, a adenina (A) complementa a timina (T), e a citosina (C) complementa a guanina (G).

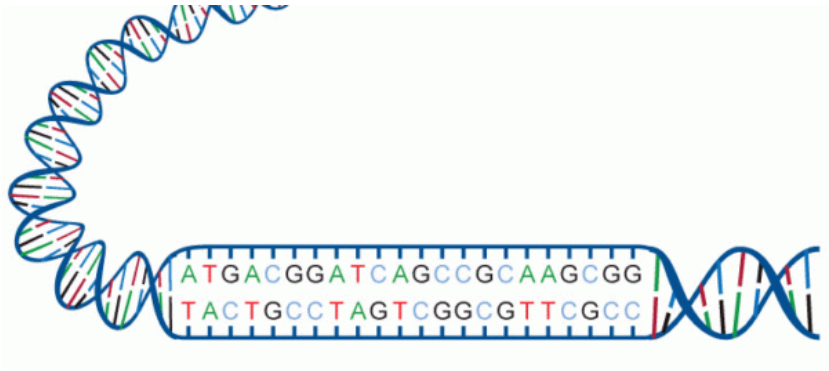

Figura 1: O DNA é composto por duplas fitas de quatro moléculas quimicamente distintas: adenina (A), guanina (G), citosina (C) e timina (T).

As sequências de DNA são armazenadas, no computador, como uma cadeia de caracteres, que podem ter formatos de entrada de sequências como: GenBank, (European

Molecular Biology Laboratory - EMBL), SwissProt, FASTA, XML entre outras. Neste trabalho vamos utilizar o formato FASTA. O formato FASTA é simples, pois a sequência não tem espaços, nem números demarcando as posições da sequência e basicamente consiste em 3 partes:

1) uma linha comentário identificada por “>” na primeira coluna seguida pelo nome e origem da sequência;

2) a própria sequência; e

3) um “*” opcional que pode indicar fim do programa. Além disso, ela é o formato mais conhecido e usado na Bioinformática (MOUNT, 2004). A Figura 2 é uma representação do arquivo no formato FASTA.

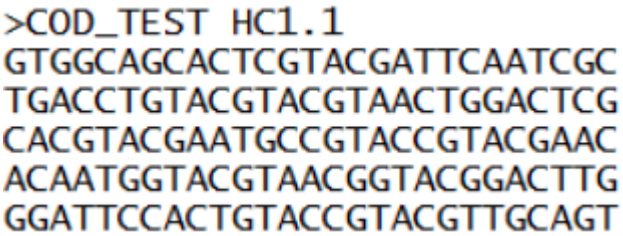

Figura 2: Exemplo de arquivo no formato FASTA. Da fonte (MOUNT, 2004)

Dadas essas características, as estratégias de compressão podem ser utilizadas para obter uma melhor taxa de compressão. A estratégia de compressão horizontal explora propriedades estruturais da própria sequência genômica, por exemplo, palíndromos, repetições e propriedades estatísticas. Assim, a transformação dos dados está baseada nas características contidas na própria sequência genômica. Desta forma, a compressão horizontal não depende de informações de outra sequência. A Figura 3 ilustra essa estratégia.

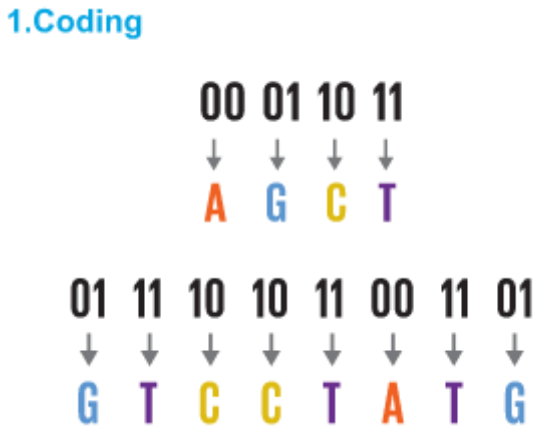

Figura 3: Exemplo da transformação dos dados, utilizando-se da estratégia horizontal, em que é definida que cada letra do alfabeto possui uma representação binária, por exemplo, A=00, G=01, C=10 e T=11 e para cada base é substituído pelo binário corres.

Atualmente, uma técnica muito interessante é usar um modelo de predição do próximo caractere de uma sequência. Nesta dissertação, nós avaliamos em particular o uso da aprendizagem profunda para predizer o próximo caractere de uma sequência genômica.

2.2 Aprendizagem profunda

As indústrias tradicionais, como agricultura, transporte, etc., e as indústrias modernas, como cultura, alimentação, turismo, etc., foram impulsionados em muitos aspectos na sociedade moderna devido à inteligência artificial (LIU et al., 2020). A Inteligência Artificial é uma área ampla na Ciência da Computação que faz com que os computadores desenvolvam a capacidade de compreender e aprender, agindo de forma semelhante à inteligência humana. É como se um *software* tentasse imitar o cérebro e seu processo de aprendizagem para agir por conta própria, de forma independente. Aqui, podemos entender "inteligência" como a capacidade de tomar a decisão certa com base em um conjunto de informações em uma variedade de ações possíveis. É também um conjunto de propriedades da mente, como capacidade de planejar, resolver problemas e raciocinar.

Observe que na Figura 4, temos o aprendizado de máquina, do inglês, *Machine Learning*, que está no contexto da inteligência artificial, *Machine Learning* é a tecnologia central da inteligência artificial, cujo objetivo é estudar os algoritmos que os sistemas de computação que utilizam para realizar tarefas, aprendendo a partir de dados, em vez de seguir instruções explícitas (LIU et al., 2020). Segundo LIU et al., (2020), apesar de suas aplicações extensas, as técnicas convencionais de aprendizado de máquina são limitadas em sua capacidade de processar dados naturais em formas brutas e aprender padrões intrincados em conjuntos de dados complexos (LIU et al., 2020).

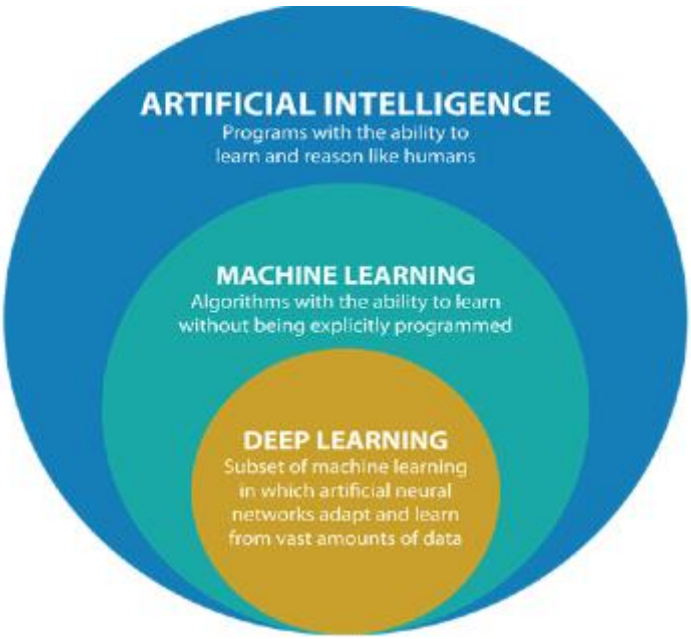

Figura 4: Inteligência artificial vs. aprendizado de máquina vs aprendizagem profunda. Da fonte: https://aakeria.com/blog/Ml_AI_DL_Blog.html

Os algoritmos de aprendizado de máquina são geralmente categorizados como aprendizado supervisionado, aprendizado não supervisionado e aprendizado semi-supervisionado. A forma mais comum de aprendizado de máquina é o aprendizado supervisionado, em que cada exemplo no conjunto de dados é rotulado. Espera-se que a máquina aprenda o mapeamento da entrada para a saída durante o processo de treinamento e consiga produzir previsões sensatas sobre novos dados. Por exemplo, um sistema de

aprendizado de máquina de classificação de imagens deve conseguir classificar uma imagem não vista em sua categoria após ser treinado em centenas de milhões de imagens rotuladas.

Na bioinformática, pode-se usar o aprendizado supervisionado para predizer os níveis de expressão do gene, a estrutura da população e assim por diante (KROGEL; SCHEFFER, 2004). Em contraste, no aprendizado não supervisionado, o conjunto de dados não têm rótulos. O algoritmo de aprendizagem deve agrupar adequadamente exemplos de dados aprendendo a função que minimiza o *gap* intragrupo e maximiza o *gap* intergrupo. Dois dos principais métodos usados na aprendizagem não supervisionada são análise de componente principal e análise de agrupamento (BOWDEN; MITCHELL; SARHADI, 1997), ambos amplamente utilizados em análise transcriptômica sobre conjuntos de dados RNA-Seq (KISELEV; ANDREWS; HEMBERG, 2019). Já o aprendizado semi-supervisionado fica entre supervisionada e não aprendizagem supervisionada, pois gera funções adequadas para aprender a partir de dados marcados e não marcados.

No contexto de aprendizado de máquina, cf. Figura 4, temos a aprendizagem profunda, sendo um ramo do aprendizado de máquina, baseado em redes neurais artificiais, que recentemente fez avanços na resolução de problemas que representam grandes esforços na comunidade da inteligência artificial por muitos anos (LECUN; BENGIO; HINTON, 2015). Uma rede neural imita o cérebro humano, então o aprendizado profundo também é uma espécie de imitação do cérebro humano. No aprendizado profundo, não precisamos programar tudo explicitamente.

O conceito de aprendizado profundo não é novo, já existe há alguns anos. Atualmente, ele ganhou destaque pelo fato de o poder de processamento ter aumentado exponencialmente e a geração de muitos dados. Podemos citar algumas diferenças entre aprendizado de máquina e aprendizado profundo, o aprendizado de máquina com um *Dataset* pequeno já vamos obter uma maior precisão nos resultados, ela depende de uma máquina de baixo custo e o tempo para treinamento é menor. Em contrapartida, no aprendizado profundo é necessário um *Dataset* grande, dependendo de máquinas de última geração para treinar e o tempo para treinamento é maior.

As vantagens da utilização de algoritmos de aprendizado profundo são:

1) melhor desempenho da classe em problemas;

2) reduz a necessidade de engenharia de características;

3) elimina custos desnecessários; e

4) identificar facilmente problemas difíceis.

E suas desvantagens são:

1) a grande quantidade de dados necessários;

2) poder computacional caro para treinar os modelos; e

3) sem base teórica forte (“Introduction to Deep Learning,” 2018).

2.2.1 Convolutional Neural Network

A Rede Neural de Convolução, do inglês *Convolutional Neural Network* (CNN), é um dos algoritmos de aprendizagem profunda associado ao Multi-Layer Perceptron (MLP) projetado para processar dados na forma de uma grade, um deles é uma imagem bidimensional. MLP é um dos métodos de classificação de imagens com resultados bastante significativos. Mas ele tem limitações, ou seja, só pode reconhecer objetos que estão em uma imagem, enquanto que objetos fora do centro da imagem não podem ser reconhecidos adequadamente. Portanto, a melhor solução para a deficiência de MLP é a CNN (SITORUS, 2020).

A CNN não só pode fazer um reconhecimento de objetos que estão no meio de uma imagem, mas também consegue reconhecer objetos no canto direito ou esquerdo da imagem. Basicamente, a CNN é um algoritmo que pode receber uma imagem de entrada, atribuir importância (pesos e vieses aprendidos) a vários aspectos/objetos na imagem e conseguir diferenciar uns dos outros. O pré-processamento exigido em uma CNN é muito menor em comparação com outros algoritmos de classificação. Enquanto que nos métodos primitivos, os filtros são feitos à mão, com treinamento suficiente, as CNNs conseguem aprender esses filtros/características.

As CNNs foram desenvolvidas e utilizadas pela primeira vez na década de 1980. O máximo que a CNN podia fazer naquela época era reconhecer dígitos manuscritos. Ele era utilizado principalmente nos setores postais para ler códigos postais. A arquitetura de uma CNN é análoga àquela do padrão de conectividade dos neurônios no cérebro humano, inspirada na organização do córtex visual. Os neurônios individuais respondem a estímulos apenas em uma região restrita do campo visual, conhecida como campo receptivo. Uma coleção de tais campos se sobrepõem para cobrir toda a área visual. Resumindo, o papel da CNN é reduzir as imagens a uma forma mais fácil de processar, sem perder recursos essenciais para obter uma boa previsão.

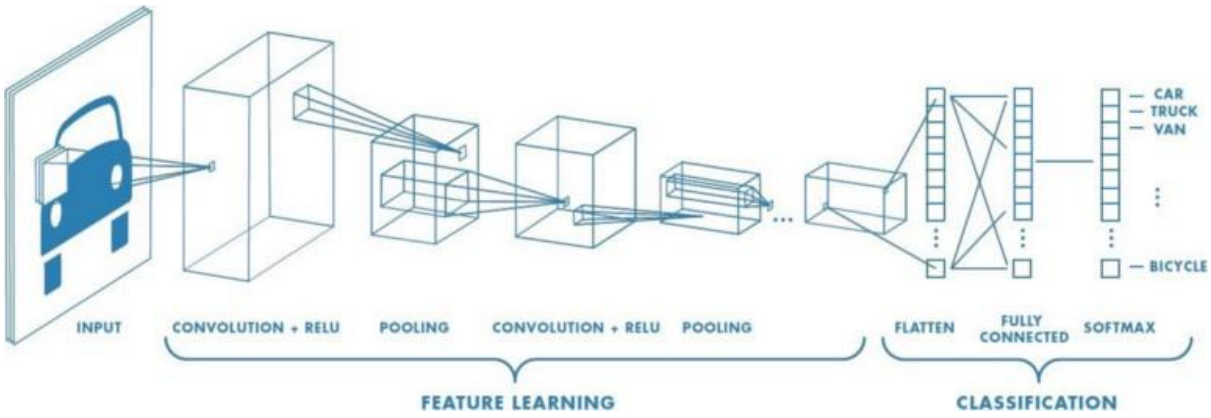

Figura 5: Tipos de camadas na rede neural de convolução. Da fonte (SITORUS, 2020)

A convolução tem três camadas em sua arquitetura, cf. Figura 5, sendo a camada convolucional, a camada de *pooling* e a camada totalmente conectada. Por essas três camadas da CNN, então essa rede é dividida em 2 seções principais que são: camada de extração de características (do inglês, *Feature Extraction Layer)*, e camada totalmente conectada (do inglês *Fully Connected Layer*) no processamento das imagens. A camada de extração de características consiste em camada convolucional e camada de *pooling* e a camada totalmente conectada consiste apenas na camada totalmente conectada (SITORUS, 2020).

2.2.2 Long Short-Term Memory Networks

As redes de memória de longo prazo (do inglês *Long Short-Term Memory* (LSTM))**,** são um tipo especial de RNN, capaz de aprender dependências de longo prazo. Elas foram introduzidas por HOCHREITER; SCHMIDHUBER, (1997), e foram refinadas e popularizadas por muitas pessoas em trabalhos subsequentes. Eles funcionam muito bem em uma grande variedade de problemas e amplamente utilizados neste momento.

Os LSTMs são projetados explicitamente para evitar o problema de dependência de longo prazo. Lembrar-se de informações por longos períodos é o comportamento padrão, não algo que eles lutam para aprender! Todas as redes neurais recorrentes têm a forma de uma cadeia de módulos repetidos de rede neural. Em RNNs padrão (cf. Figura 6), esse módulo de repetição tem uma estrutura muito simples, como uma única camada de *tanh*.

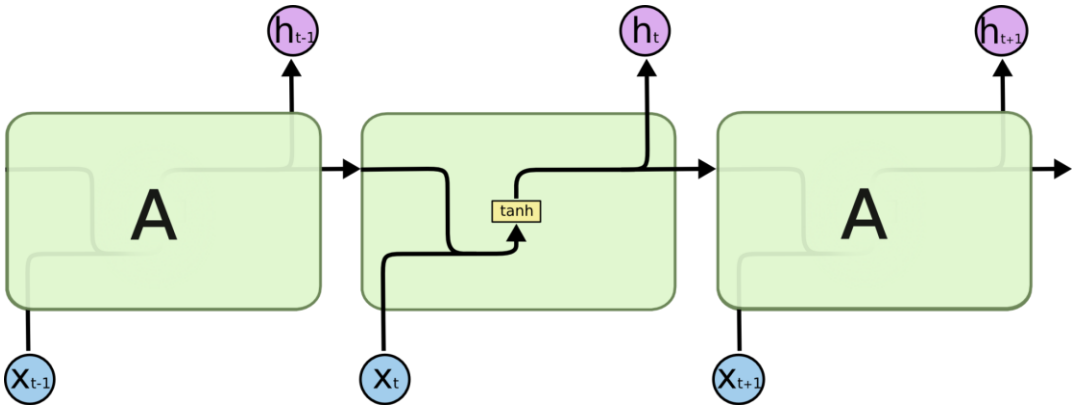

Figura 6: O módulo de repetição em um RNN padrão contém uma única camada. Da fonte: https://colah.github.io/posts/2015-08-Understanding-LSTMs

Os LSTMs também têm essa estrutura em cadeia, mas o módulo de repetição tem uma estrutura diferente. Em vez de ter uma única camada de rede neural, existem quatro, interagindo de uma maneira muito especial. Não há preocupação com os detalhes do que está acontecendo. As funções de transição LSTM são definidas da seguinte forma:

$$f_t = \delta(W_f \cdot [x_t, h_{t-1}] + b_t) \quad (2.1)$$

$$i_t = \delta(W_i \cdot [x_t, h_{t-1}] + b_i) \quad (2.2)$$

$$o_t = \delta(W_o \cdot [x_t, h_{t-1}] + b_o) \quad (2.3)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot tanh(W_c \cdot [x_t, h_{t-1}] + b_c) \quad (2.4)$$

$$h_t = o_t \odot tanh(c_t) \quad (2.5)$$

onde $f_t \in R^h$ é uma porta de esquecimento, sendo a função que controla até que ponto as informações da memória antiga celular vão ser jogado fora;$i_t \in R^h$ é uma porta de entrada que controla a quantidade de novas informações que serão armazenadas na célula de memória atual; e $o_t \in R^h$ é uma porta de saída que controla o que produzir com base na célula de memória. Através da combinação eficaz dessas três portas, o problema de gradiente é efetivamente resolvido e pode escavar dependência de longo prazo na sequência.

W_f, W_i, W_o e W_c representam matrizes de peso para a porta de esquecimento, a porta de entrada, a porta de saída e o estado da célula conexões. b_f, b_i, b_o e b_c denotam vetores de polarização para a porta de esquecimento, a porta de entrada, a porta de saída e o estado da célula conexões. x_t é a sequência de entrada de tempo atual e h_{t-1} é a saída do estado de tempo anterior. O símbolo δ é a função sigmóide logística que escala a saída para o intervalo [0 , 1], e é definido da seguinte forma:

$$Sigmoid(x) = \frac{1}{1+e^{-x}}. \quad (2.6)$$

A função *tanh* dimensiona a saída para o intervalo [- 1 , 1] e denota a função tangente hiperbólica

$$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \quad (2.7)$$

A operação $\odot$ denota multiplicação elemento a elemento como descrito na equação 2.4. Uma técnica padrão de compressão de caracteres está baseada na técnica de codificação aritmética. Essa técnica pode ser combinada com a predição do próximo caractere.

2.3 Codificação aritmética

As aplicações de compressão empregam uma ampla variedade de técnicas. Elas têm graus bastante diferentes de complexidade, mas compartilham alguns processos comuns. A Figura 7 mostra um diagrama com processos usados para compressão de dados. Esses processos dependem do tipo de dado. E os blocos (da Figura 7) podem estar em ordem diferente ou combinados.

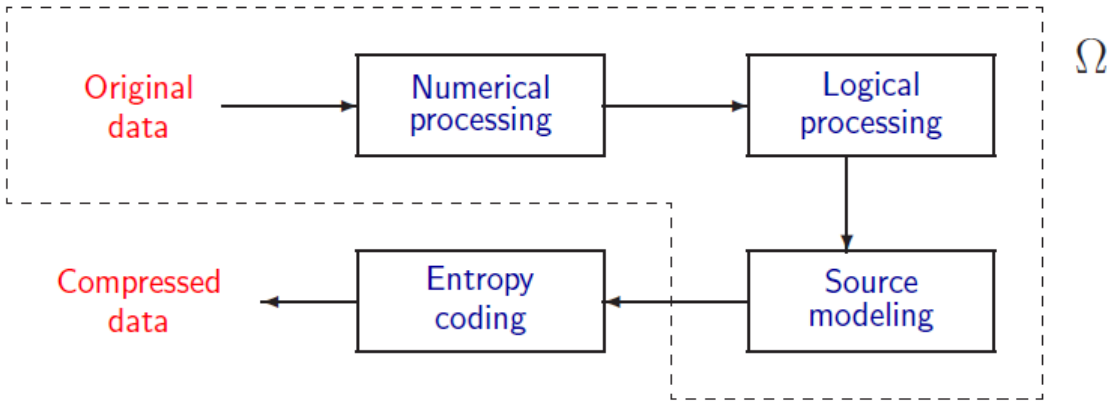

Figura 7: Sistema com processos típicos de compressão de dados. A codificação aritmética é normalmente o estágio final, e os outros estágios podem ser modelados como uma única fonte de dados Ω. Da fonte (SAYOOD, 2003)

O processamento numérico, como codificação preditiva e transformações lineares, é normalmente usado para sinais de forma de onda, como imagens e áudio (SAYOOD, 2003). O processamento lógico consiste em transformar os dados em um formulário mais adequado para

compressão, como *run-lengths*, *zero-trees*, *set-particitioning information*, e entradas de dicionário (SAYOOD, 2003). A próxima etapa, o modelo de origem, é usada para considerar as variações nas propriedades estatísticas dos dados. É responsável por reunir estatísticas e identificar contextos de dados que tornam os modelos de origem mais precisos e confiáveis (SAYOOD, 2003).

O que a maioria dos sistemas de compressão têm em comum é que o processo final é codificação de entropia (do inglês, *entropy coding*), processo de representar informações na forma mais compacta. Pode ser responsável por fazer a maior parte do trabalho de compressão, ou pode apenas complementar o que foi realizado nas etapas anteriores. Quando consideramos todos os diferentes métodos de codificação de entropia e suas possíveis aplicações de compressão, a codificação aritmética se destaca em eficácia produtividade e versatilidade, dado que ela consegue trabalhar de forma mais eficiente em um grande número de circunstâncias e propósitos. Entre seus recursos mais desejáveis, temos o seguinte.

- Quando aplicado a fontes independentes e distribuídas de forma idêntica (do inglês, *independent and identically distributed* (i.i.d.)), a compressão de cada símbolo é comprovadamente ótima.

- É eficaz em uma ampla gama de situações e taxas de compressão. A mesma aritmética de codificação pode codificar efetivamente todos os diversos dados criados pelos diferentes processos (cf. Figura 7), como parâmetros de modelagem, coeficientes de transformação, sinalização, etc.

- Simplifica a modelagem automática de fontes complexas, tornando quase ótimo ou significativo a compressão visivelmente melhorada para fontes que não são i.i.d.

- Seu processo principal é a aritmética, apoiada com eficiência crescente por todos os processadores de sinal digital ou de uso geral (CPUs, DSPs).

- É adequado para uso como uma compressão de caixa preta, do inglês *compression black-box*, por aqueles que não são especialistas em codificação ou não querem implementar o algoritmo de codificação por conta própria.

Mesmo com todas essas vantagens, a codificação aritmética não é tão popular e bem compreendida como outros métodos. Certos problemas práticos impediram sua adoção:

- A complexidade das operações aritméticas era excessiva para aplicativos de codificação.
- Patentes cobrem as implementações mais eficientes. *Royalties* e o medo de violação da patente desencorajou a codificação aritmética em produtos comerciais.
- Implementações eficientes eram difíceis de entender.

No entanto, esses problemas agora estão quase totalmente superados. Primeiro, com a eficiência relativa do computador, a aritmética melhorou significativamente e novas técnicas evitam as operações mais caras. Em segundo lugar, algumas das patentes expiraram ou tornaram-se obsoletas. Finalmente, não precisa se preocupar tanto com detalhes de redução de complexidade que obscurecem a inerente simplicidade do método. Os recursos computacionais atuais nos permitem implementar simples codificação aritmética eficiente e livre de *royalties* (SAYOOD, 2003).

2.3.1 Notação

Seja Ω uma fonte de dados que exibe símbolos s_k codificados como números inteiros com o conjunto $\{0,1,\ldots,M-1\}$, e seja $S = \{s_1, s_2, \ldots, s_N\}$uma sequência de N símbolos aleatórios colocados para fora por Ω (SAYOOD, 2003). Assume-se que os símbolos de origem são independentes e distribuído de forma idêntica, com probabilidade

$$p(m) = Prob\{s_k = m\}, m = 0,1,2,\ldots,M-1, k = 1,2,\ldots,N. \quad (2.8)$$

também assume-se que para todos os símbolos tem-se $\rho(m) \neq 0$ e define-se $c(m)$ como a distribuição cumulativa,

$$c(m) = \sum_{s=0}^{m-1} p(s), m = 0,1,\ldots,M. \quad (2.9)$$

Observe que $c(0) \equiv 0$, $c(M) \equiv 1$, e

$$p(m) = c(m+1) - c(m) \quad (2.10)$$

Usamos letras em negrito para representar os vetores com todos os valores $p(m)$ e $c(m)$, ou seja,

$$\mathbf{p} = [p(0)p(1)\ldots p(M-1)],$$
$$\mathbf{c} = [c(0)c(1)\ldots c(M-1)\ c(M)].$$

Assume-se que os dados compactados (saída do codificador) são salvos em um vetor (*buffer*) **d**. O alfabeto de saída possui *D*-símbolos, ou seja, cada elemento em **d** pertence ao conjunto $\{0,1,\ldots,D-1\}$.

Sob as suposições acima, um método de codificação ideal codifica cada símbolo s de Ω com um número médio de bits igual a

$$B(s) = -log_2 p(s) \ \ bits. \quad (2.11)$$

Por exemplo, a fonte de dados Ω pode ser um arquivo com texto em inglês: cada símbolo desta fonte é um *byte* único que representa um caractere. Este alfabeto de dados contém M = 256 símbolos e os números dos símbolos são definidos pelo padrão ASCII. As probabilidades dos símbolos podem ser estimadas reunindo estatísticas usando um grande número de texto em inglês. A Tabela 1 mostra alguns caracteres, seus valores de símbolo ASCII e suas probabilidades estimadas. Ela também mostra o número de bits necessários para o código de símbolo s em uma maneira ótima, $-log_2 p(s)$. A partir desses números, conclui-se que, se os símbolos do texto em inglês eram i.i.d., então a melhor taxa de compressão do texto possível seria de cerca de 2:1 (4 bits/símbolo). Métodos especializados de compressão de texto (ZIV; LEMPEL, 1977; ZIV; LEMPEL, 1978; BELL; CLEARY; WITTEN, 1990) podem resultar taxas de compressão significativamente melhores porque exploram a dependência estatística entre as letras.

Este primeiro exemplo mostra que nossas suposições iniciais sobre as fontes de dados são raramente encontradas em casos práticos. Normalmente, tem-se os seguintes problemas:

1. Os símbolos de origem não são distribuídos de forma idêntica.

2. Os símbolos na sequência de dados não são independentes (mesmo que não sejam correlacionados).

3. Pode-se apenas estimar os valores de probabilidade, a dependência estatística entre os símbolos e como eles mudam com o tempo.

No entanto, nas próximas seções, mostraremos que a generalização da codificação aritmética para fontes variáveis no tempo é direta e explicamos como abordar todas essas fontes práticas.

Tabela 1: Probabilidades estimadas de algumas letras e sinais de pontuação no inglês língua. Os símbolos são numerados conforme o padrão ASCII. Da fonte (SAYOOD, 2003)

Caractere	Símbolo ASCII	Probabilidade	Número ótimo de bits
	s	$p(s)$	$-log_2 p(s)$
Espaço	32	0,1524	2,714
,	44	0,0136	6,205
.	46	0,0056	7,492
A	65	0,0017	9,223
B	66	0,0009	10,065
C	67	0,0013	9,548
a	97	0,0595	4,071
b	98	0,0119	6,391
c	99	0,023	5,441
d	100	0,0338	4,887
e	101	0,1033	3,275
f	102	0,0227	5,464
t	116	0,0707	3,823
Z	122	0,0005	11,0669

2.3.2 Valores de código

A codificação aritmética difere de outros métodos de codificação para os quais conhece-se exatamente a relação entre os símbolos codificados e os bits reais gravados em um arquivo. Isto codifica um símbolo de dados por vez e atribui a cada símbolo um número de bits com um valor real (cf. última coluna da Tabela 1). Para descobrir como isso é possível, tem-se que compreender a representação do valor do código: mensagens codificadas mapeadas para números reais no intervalo [0, 1).

O valor do código v de uma sequência de dados compactada é o número real com dígitos fracionários, igual aos símbolos da sequência. Pode-se converter sequências em valores de código simplesmente adicionando "0." para o início de uma sequência codificada e, em seguida, interpretando o resultado como um número em notação base-D, onde D é o número de símbolos no alfabeto de sequência codificada. Para exemplo, se um método de codificação gera a sequência de bits 0011000101100, então tem-se:

$$\begin{aligned} \textit{Sequência de código} \quad & d = [0011000101100] \\ \textit{Valor do código} \quad & v = 0.0011000101100_2 = 0.19287109375 \end{aligned} \tag{2.12}$$

onde o subscrito "2" denota notação de base-2. Como de costume, omite-se o subscrito para notação decimal.

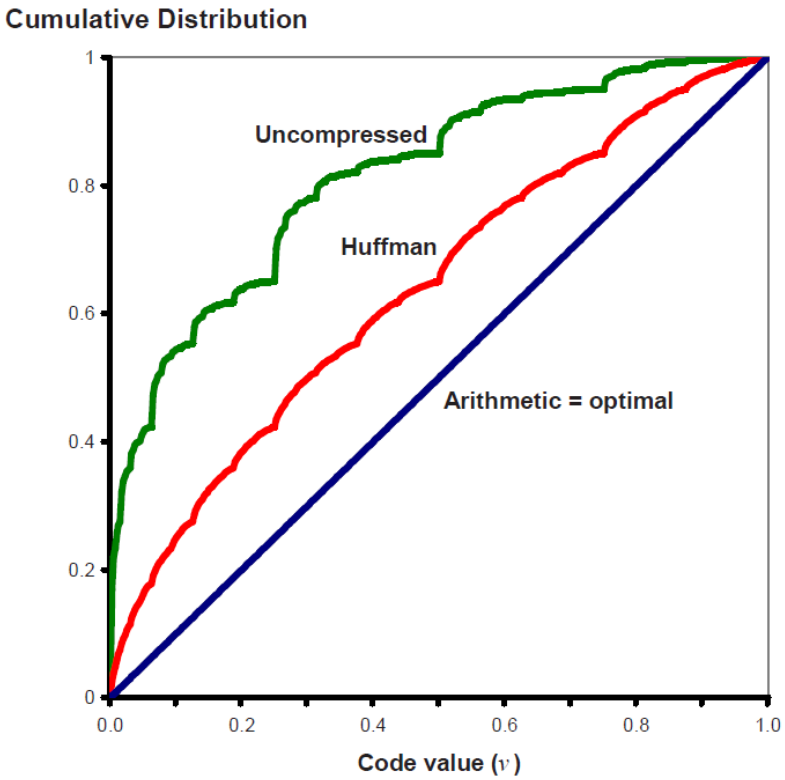

Figura 8: Distribuição cumulativa de valores de código gerados por diferentes métodos de codificação quando aplicado à fonte do Exemplo 2. Da fonte (SAYOOD, 2003)

Esta construção cria um mapeamento conveniente entre sequências infinitas de símbolos de um alfabeto de *D*-símbolos e números reais no intervalo [0, 1), onde qualquer sequência de dados pode ser representada por um número real e vice-versa. A representação do valor do código pode ser usada para qualquer sistema de codificação e fornece uma maneira universal de representar grandes quantidades de informações independentemente do conjunto de símbolos utilizados para codificação (binário, ternário, decimal, etc.). Por exemplo, na equação 2.5 vemos o mesmo código com representações de base-2 e base-10.

Pode-se avaliar a eficácia de qualquer método de compressão, analisando a distribuição dos valores de código que ele produz. Da teoria da informação de Shannon (SHANNON, 1948) sabe-se que, se um método de codificação é ótimo, então a distribuição cumulativa de seus valores de código tem que ser uma linha reta do ponto (0, 0) ao ponto (1, 1).

Por exemplo, assumindo-se que a fonte i.i.d. Ω tem quatro símbolos, e as probabilidades dos símbolos são $p = [0.68\ 0.2\ 0.1\ 0.05]$. Se codificarmos sequências de dados aleatórios de

fontes com dois bits por símbolos, os valores de códigos resultantes produzem uma distribuição conforme mostrado na Figura 8, sob o rótulo "descompactado". Observe como a distribuição é enviesada, indicando a possibilidade de compressão significativa.

As mesmas sequências podem ser codificadas com o código de Huffman para Ω, com um bit usado para o símbolo “0”, dois bits para o símbolo “1” e três bits para os símbolos “2” e “3”. A distribuição cumulativa do valor do código correspondente na Figura 8 mostra que há uma melhora substancial em relação ao caso não compactado, mas este método de codificação ainda não é o ideal. A terceira linha na Figura 8 mostra que as sequências compactadas com simulação de codificação aritmética produzem um valor de código distribuição que é praticamente idêntica à ótima.

A distribuição em linha reta significa que se um método de codificação é ideal, então não há nenhuma dependência estatística ou redundância deixada nas sequências compactadas e, consequentemente, seus valores de código são uniformemente distribuídos no intervalo [0, 1). Este fato é essencial para compreensão de como funciona a codificação aritmética. Além disso, os valores do código são partes integrantes dos procedimentos aritméticos de codificação/decodificação, com operações aritméticas aplicadas aos números reais que estão diretamente relacionados aos valores do código. Um comentário final sobre os valores do código: duas sequências diferentes infinitamente longas podem corresponder para o mesmo valor de código. Isso decorre do fato de que para qualquer $D > 1$, tem-se

$$\sum_{n=k}^{\infty} (D-1)D^{-n} = D^{1-k}. \qquad (2.13)$$

Por exemplo, se $D = 10$ e $k = 2$, então (2.6) é a igualdade $0.0999999\ldots = 0.1$. Este fato não tem significado prático importante para fins de codificação, mas precisa-se levá-lo em conta ao estudar algumas propriedades teóricas da codificação aritmética.

2.3.3 Processo de codificação

Nesta seção, apresentamos primeiro a notação e as equações que descrevem a codificação aritmética, seguido por um exemplo detalhado. O processo de codificação aritmética consiste em criar uma sequência de intervalos aninhados na forma $\Phi_k(S) = [\alpha_k, \beta_k), k = 0,1,\ldots,N,$

onde S é a sequência de dados de origem, α_ke $\beta_k$são números reais tais que $0 \leq \alpha_k \leq \alpha_{k+1}$, $\beta_{k+1} \leq \beta_k \leq 1$. Para uma maneira mais simples de descrever a codificação aritmética, representa-se os intervalos na forma $|\, b, l \,\rangle$, onde b é chamado base ou ponto inicial do intervalo, e l o comprimento do intervalo. A relação entre a notação de intervalo tradicional e a nova é

$$|\, b, l \,\rangle = [\alpha, \beta) \;\; if \; b = \alpha \;\; e \; l = \beta - \alpha \tag{2.14}$$

Os intervalos utilizados durante o processo de codificação aritmética são, nesta nova notação, definidos pelo conjunto de equações recursivas.

$$\Phi_0(S) = |\, b_0, l_0 \,\rangle = |\, 0, 1 \,\rangle \,, \tag{2.15}$$

$$\Phi_k(S) = |\, b_k, l_k \,\rangle = |\, b_{k-1} + c(s_k)\, l_{k-1}, p(s_k)\, l_{k-1} \rangle \,, k = 1,2,\ldots,N. \tag{2.16}$$

As propriedades dos intervalos garantem que $0 \leq b_k \leq b_{k+1} < 1$ e $0 < l_{k+1} < l_k \leq 1$. A Figura 9 mostra um sistema dinâmico, correspondente ao conjunto de equações recursivas (2.16). Em seguida, explicamos como escolher, no final do processo de codificação, um valor de código no final intervalo, ou seja, $\hat{v}(S) \in \Phi_N(S)$.

O processo de codificação definido por (2.15) e (2.16), também chamado *Elias coding*, foi descrito pela primeira vez em (JELINEK, 1968). A convenção de representar um intervalo

usando sua base e comprimento foi usada desde os primeiros artigos de codificação aritmética (RUBIN, 1979). Outros autores têm intervalos representados por seus pontos extremos, como $[base, base + comprimento)$, mas não há diferença matemática entre as duas notações.

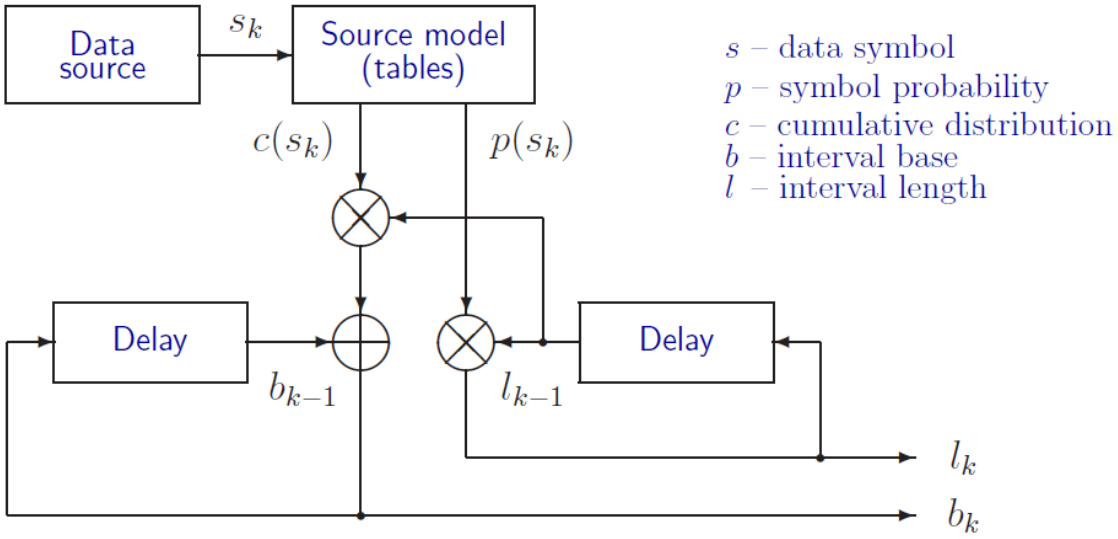

Figura 9: Sistema dinâmico para atualização de intervalos na codificação aritmética. Da fonte (SAYOOD, 2003).

Por exemplo, assume-se que a fonte Ω tem quatro símbolos ($M = 4$), as probabilidades e distribuição dos símbolos são $p = [0.2\ 0.5\ 0.2\ 0.1]$ e $c = [0\ 0.2\ 0.7\ 0.9\ 1]$, e a sequência de ($N = 6$) símbolos a serem codificados é $S = \{2,1,0,0,1,3\}$.

A Figura 10 mostra graficamente como o processo de codificação corresponde à seleção de intervalos na linha de números reais. Começa-se no topo da Figura 10, com o intervalo [0, 1), dividido em quatro subintervalos, cada um com comprimento igual à probabilidade dos símbolos de dados. Especificamente, o intervalo $[0, 0.2)$ corresponde a $s_1 = 0$, intervalo $[0.2, 0.7)$ corresponde a $s_1 = 1$, intervalo $[0.7, 0.9)$ corresponde a $s_1 = 2$, e, finalmente, o intervalo $[0.9, 1)$ corresponde a $s_1 = 3$. O próximo conjunto de dados aninhados permitidos subintervalos também têm comprimento proporcional à probabilidade dos símbolos, mas seus os comprimentos também são proporcionais ao comprimento do intervalo ao qual pertencem. Além disso, eles representam mais de um valor de símbolo. Por exemplo, intervalo $[0, 0.04)$

corresponde para $s_1 = 0, s_2 = 0,$ intervalo $[0.04, 0.14)$ corresponde a $s_1 = 0, s_2 = 1$ e assim por diante.

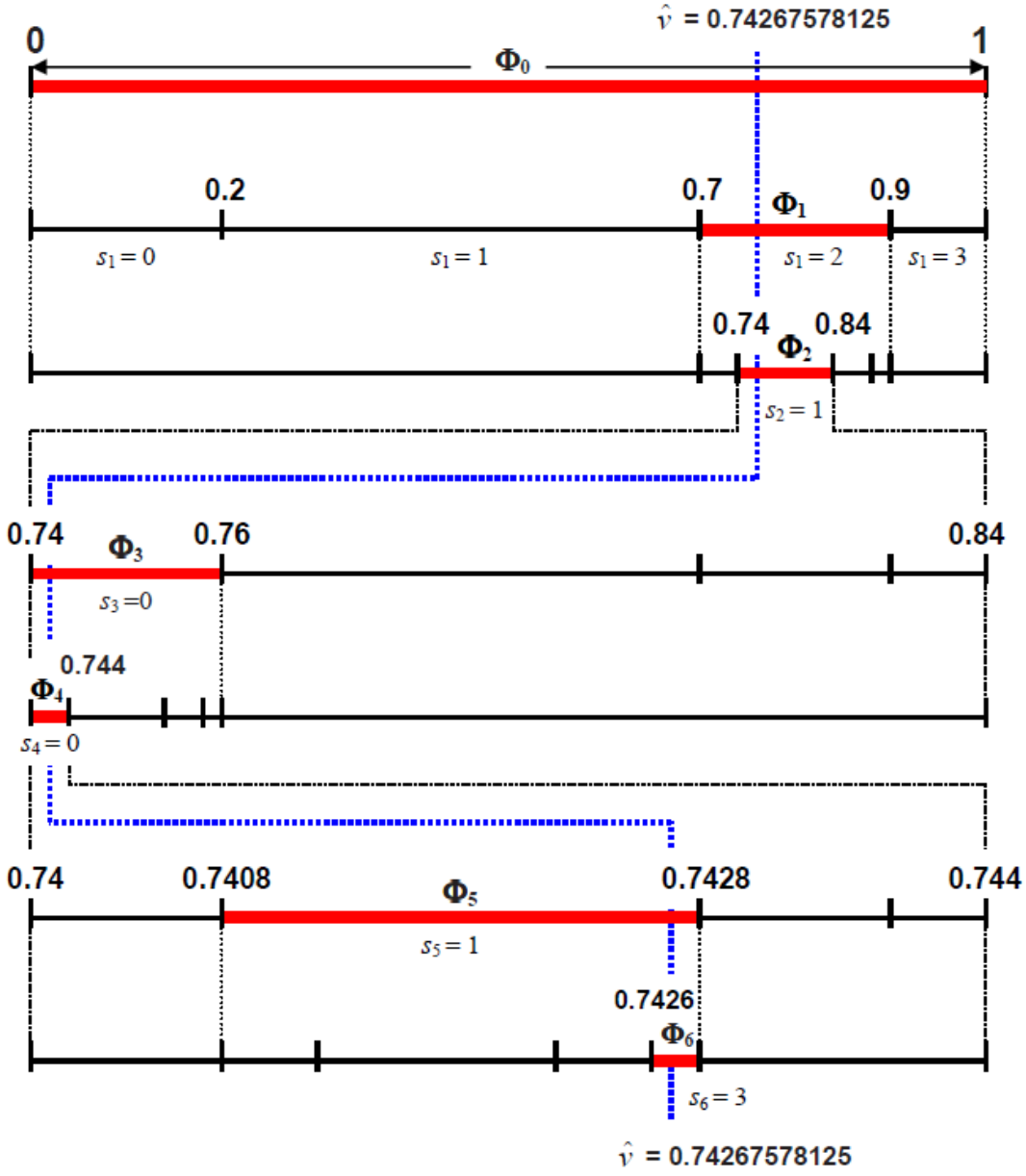

Figura 10: Representação gráfica do processo de codificação aritmética, o intervalo $\Phi_0 = [0, 1)$ *é dividido em intervalos alinhados conforme a probabilidade dos símbolos. Os intervalos selecionados, correspondentes à sequência de dados* $S = \{2,1,0,0,1,3\}$ *são indicados por linhas mais grossas. Da fonte (SAYOOD, 2003)*

Os comprimentos dos intervalos são reduzidos por fatores iguais às probabilidades dos símbolos para obter valores de código que são uniformemente distribuídos no intervalo [0, 1). Por exemplo, se 20% das sequências começam com o símbolo "0", então 20% dos valores do código devem estar no intervalo atribuído a essas sequências, o que só pode ser alcançado se atribuirmos ao primeiro símbolo "0" um intervalo com comprimento igual à sua probabilidade, *0.2*. O mesmo raciocínio se aplica à atribuição dos comprimentos de subintervalo: cada ocorrência do símbolo "0" deve resultar em uma redução do comprimento do intervalo para 20% do seu comprimento atual. Assim, após a codificação de vários símbolos, a distribuição dos valores do código deve ser uma boa aproximação de uma distribuição uniforme.

Tabela 2: Resultados de codificação e decodificação aritmética para os Exemplos 3 e 4. Os dois últimos as linhas mostram o que acontece quando a decodificação continua após o último símbolo. Da fonte (SAYOOD, 2003)

Interação	Símbolo de entrada	Intervalo base	Tamanho do intervalo	Valor atualizado do decodificador	Símbolo de saída
k	s_k	b_k	l_k	$\underline{v}_k = \frac{\hat{v} - b_{k-1}}{l_{k-1}}$	$\hat{s}_k$
0	-	0	1	-	2
1	2	0.7	0.2	0.74267578125	1
2	1	0.74	0.1	0.21337890625	0
3	0	0.74	0.02	0.0267578125	0
4	0	0.74	0.004	0.1337890625	0
5	1	0.7408	0.002	0.6689453125	1
6	3	0.7426	0.0002	0.937890625	3
7	-	-	-	0.37890625	1
8	-	-	-	0.3578125	1

As equações (2.8) e (2.9) fornecem as fórmulas para o cálculo sequencial dos intervalos. Aplicando-os ao nosso exemplo, obtém-se:

$$\Phi_0(S) = |0,1\rangle = [0,1),$$

$$\Phi_1(S) = |b_0 + c(2)l_0, p(2)l_0\rangle = |0 + 0.7 \times 1, 0.2 \times 1\rangle = [0.7, 0.9),$$

$$\Phi_2(S) = |b_1 + c(1)l_1, p(1)l_1\rangle = |0.7 + 0.2 \times 0.2, 0.5 \times 0.2\rangle = [0.74, 0.84),$$

.
.
.

$$\Phi_6(S) = |b_5 + c(3)l_5, p(3)l_5\rangle = |0.7426, 0.0002\rangle = [0.7426, 0.7428),$$

A lista com todos os intervalos do codificador é mostrada nas primeiras quatro colunas da Tabela 2. Visto que os intervalos rapidamente se tornam muito pequenos, na Figura 10 temos que ampliá-los graficamente (duas vezes) para podermos ver como o processo de codificação continua. Observe que mesmo que os intervalos sejam mostrados em diferentes ampliações, os valores dos intervalos não mudam, e o processo para subdividir os intervalos continuam exatamente da mesma maneira.

A tarefa final na codificação aritmética é definir um valor de código $\hat{v}(S)$ que irá representar sequência de dados S. Na próxima seção, será mostrada como o processo de decodificação funciona corretamente para qualquer valor de código $\hat{v} \in \Phi_N(S)$. No entanto, o valor do código não pode ser fornecido ao decodificador como um número real puro. Deve ser armazenado ou transmitido, usando uma representação de número convencional. Tem-se a liberdade de escolher qualquer valor no intervalo final, assim pode-se escolher valores com a representação mais curta. Por exemplo, no exemplo acima, a menor representação decimal vem

desde a escolha de $\hat{v} = 0.7427$, e o binário mais curto representação é obtida com $\hat{v} = 0.10111110001_2 = 0.74267578125$.

O processo para encontrar a melhor representação binária é bastante simples. A ideia principal é que, para intervalos relativamente grandes, pode-se encontrar o valor ideal testando algumas sequências binárias, e como os comprimentos dos intervalos são reduzidos pela metade, o número das sequências a serem testadas precisam dobrar, aumentando o número de bits em um. Assim, conforme o comprimento do intervalo l_N, pode-se usar as seguintes regras:

- Se $l_N \in [0.5, 1)$, em seguida, escolher valor de código $\hat{v} \in \{0, 0.5\} = \{0.0_2, 0.1_2\}$para uma representação de 1 bit.
- Se $l_N \in [0.25, 0.5)$, em seguida, escolher o valor $\hat{v} \in \{0, 0.25, 0.5, 0.75\} = \{0.00_2, 0.01_2, 0.10_2, 0.11_2\}$para uma representação de 2 bits.
- Se $l_N \in [0.125, 0.25)$, em seguida, escolher o valor $\hat{v} \in \{0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875\} = \{0.000_2, 0.001_2, 0.010_2,$

  $0.011_2, 0.100_2, 0.101_2, 0.110_2, 0.111_2\}$para uma representação de 3 bits.

Ao observar o padrão, conclui-se que o número mínimo de bits necessários para representar $\hat{v} \in \Phi_N(S)$ é

$$B_{min} = \lceil -log_2(l_N) \rceil \quad bits, \tag{2.17}$$

onde $\lceil x \rceil$ representa o menor inteiro maior que ou igual a x.

Pode-se testar essa conclusão observando os resultados na Tabela 2. O final o intervalo é $l_N = 0.0002$, e, assim, $B_{min} = \lceil -log_2(0.002) \rceil = 13\ bits$. Por exemplo, pode-se escolher $\hat{v} = 0.10111110001_2$, e requer apenas 11 bits.

A origem desta inconsistência é que pode-se escolher representações binárias com o número de bits fornecido por (10) e, em seguida, remover os zeros à direita. No entanto, com uma ótima codificação, o número médio de bits que podem ser salvos com este processo é apenas um bit, e por esse motivo é raramente aplicado na prática.

2.3.4 Processo de decodificação

A codificação aritmética, a sequência decodificada é determinada exclusivamente pelo valor de código $\hat{v}$ da sequência comprimida. Por esse motivo, representa-se a sequência decodificada como

$$\hat{S}(\hat{v}) = \{\hat{s}_1(\hat{v}), \hat{s}_2(\hat{v}), \ldots, \hat{s}_N(\hat{v})\}. \qquad (2.18)$$

Será agora o processo de descodificação por meio do qual qualquer valor de código $\hat{v} \in \Phi_N(S)$ pode ser utilizado para decodificar a sequência correta (isto é, $\hat{S}(\hat{v}) = S$. Para tal será apresentado o conjunto de equações recursivas que implementam a decodificação, seguido por um exemplo prático que fornece uma ideia intuitiva de como funciona o processo de decodificação e por que está correto.

O processo de decodificação recupera os símbolos de dados na mesma sequência em que estavam codificados. Formalmente, para encontrar a solução numérica, define-se uma sequência de código normalizado de valores $\{\underline{v}_1, \underline{v}_2, \ldots, \underline{v}_N\}$. Começando com $\underline{v}_1 = \hat{v}$, pode-se sequencialmente encontrar $\hat{s}_k$ de $\underline{v}_k$, e então computa-se $\hat{v}_{k+1}$ a partir de $\hat{s}_k$ e $\underline{v}_k$.

As fórmulas de recursão são:

$$\underline{v}_1 = \hat{v}, \tag{2.19}$$

$$\hat{s}_k(\hat{v}) = \{s: c(s) \leq \underline{v}_k < c(s+1)\}, \qquad k = 1,2,\ldots,N, \tag{2.20}$$

$$\underline{v}_{k+1} = \frac{\underline{v}_k - c(\hat{s}_k(\hat{v}))}{p(\hat{s}_k(\hat{v}))}, \qquad k = 1,2,\ldots,N-1. \tag{2.21}$$

Na Equação (2.20), os dois pontos significam "s que satisfaz as desigualdades."

Um método de decodificação matematicamente equivalente recupera a sequência de intervalos criados pelo codificador, e busca o valor correto $\hat{s}_k(\hat{v})$ em cada um destes intervalos. Está definido por

$$\Phi_0(\hat{S}) = |b_0, l_0\rangle = |0,1\rangle, \tag{2.22}$$

$$\hat{s}_k(\hat{v}) = \{s: c(s) \leq \frac{\hat{v} - b_{k-1}}{l_{k-1}} < c(s+1)\}, \qquad k = 1,2,\ldots,N, \tag{2.23}$$

$$\Phi_k(\hat{S}) = |b_k, l_k\rangle = |b_{k-1} + c\big(\hat{s}_k(\hat{v})\big) l_{k-1}, p\big(\hat{s}_k(\hat{v})\big) l_{k-1}\rangle, \qquad k = 1,2,\ldots,N. \tag{2.24}$$

a combinação de recursão (2.21) com recursão (2.24) produz

$$\underline{v_k} = \frac{\hat{v} - \sum_{i=1}^{k-1} c(\hat{s}_i) \prod_{j=1}^{i-1} p(\hat{s}_j)}{\prod_{i=1}^{k-1} p(\hat{s}_i)} = \frac{\hat{v} - b_{k-1}}{l_{k-1}}. \tag{2.25}$$

mostrando que (2.20) é equivalente a (2.23).

Será mostrada a aplicação do processo de decodificação aos dados obtidos na Seção 2.3.3 de codificação. Na Figura 10, pode-se ver graficamente o significado de $\hat{v}$: que é um valor que pertence a todos os intervalos aninhados criados durante a codificação. A linha pontilhada mostra que sua posição se move à medida que amplia-se os gráficos, mas o valor permanece o mesmo. Na Figura 10, pode-se começar a decodificação do primeiro intervalo de $\Phi_0(S) = [0,1)$: só se deve comparar $\hat{v}$ com a distribuição cumulativa c para encontrar o único valor possível de $\hat{s}_1$

$$\hat{s}_1(\hat{v}) = \{s: c(s) \leq \hat{v} = 0.74267578125 < c(s+1)\} = 2.$$

Pode-se usar o valor de $\hat{s}_1$ para descobrir o intervalo $\Phi_1(S)$, e usá-lo para determinar $\hat{s}_2$. Na verdade, pode-se "remover" o efeito de $\hat{s}_1$ em $\hat{v}$, definindo o valor do código normalizado

$$\underline{v}_2 = \frac{\hat{v} - c(\hat{s}_1)}{p(\hat{s}_1)} = 0.21337890625.$$

Observe que, em geral, $\underline{v}_2 \in [0,1)$, ou seja, é um valor normalizado para o intervalo inicial. Neste intervalo, pode-se usar o mesmo processo para encontrar

$$\hat{s}_2(\hat{v}) = \{s: c(s) \leq \underline{v}_2 = 0.21337890625 < c(s+1)\} = 1.$$

As últimas colunas da Tabela 2 mostram como o processo continua, e os valores calculados durante a decodificação. Poder-se-ia dizer que o processo continua até que $\hat{s}_6$ seja decodificado. No entanto, como pode o descodificador, tendo apenas o valor inicial do código $\hat{v}$, saber que é hora de parar de decodificar? A resposta é simples: não pode. Deve-se adicionar

duas linhas extras na Tabela 2 para mostrar que o processo de decodificação pode continuar normalmente após o último símbolo ser codificado.

É importante entender que a codificação aritmética mapeia intervalos para conjuntos de sequências. Cada número real em um intervalo corresponde a uma sequência infinita. Assim, as sequências correspondendo a $\Phi_6(S) = [0.7426, 0.7428)$ são todos aqueles que começam como $\{2,1,0,0,1,3,\ldots\}$. O valor de código $\hat{v} = 0.74267578125$ corresponde a uma dessas sequências infinitas, e a decodificação do processo pode continuar para sempre decodificando essa sequência particular.

Existem duas maneiras práticas de informar que a decodificação deve parar:

1. Forneça o número de símbolos de dados (N) no início do arquivo compactado.
2. Usar um símbolo especial como "fim da mensagem", codificado apenas no final dos dados sequência, e atribuir a este símbolo o menor valor de probabilidade permitido pelo codificador/decodificador.

Como explicado acima, o procedimento de decodificação sempre produzirá dados da sequência decodificados. No entanto, como se sabe qual é a sequência certa? Isso pode ser inferido do fato de que se S e S' são sequências com N símbolos, então

$$S \neq S' \Leftrightarrow \Phi_N(S) \cap \Phi_N(S') = \emptyset. \qquad (2.26)$$

Isso garante que sequências diferentes não possam produzir o mesmo valor de código.

2.3.5 Otimização da codificação aritmética

A teoria da informação nos mostra que o número médio de bits necessários para codificar cada símbolo de uma fonte estacionária e sem memória Ω não pode ser menor que sua entropia $H(\Omega)$, definida por:

$$H(\Omega) = -\sum_{m=0}^{M-1} p(m) log_2 p(m) \quad bits/simbolo. \tag{2.27}$$

Viu-se que o processo de codificação aritmética gera valores de código que são uniformemente distribuídos no intervalo $[0,1)$. Esta é uma condição necessária para a otimização, mas não suficiente. No intervalo $\Phi_N(S)$, pode-se escolher valores que requerem um valor arbitrariamente grande de número de bits a serem representados, ou escolher valores de código que podem ser representados com o número mínimo de bits, dado pela equação (2.17). Será mostrado na sequência que a última escolha satisfaz a condição suficiente para a otimização.

Para começar, deve-se considerar que há alguma sobrecarga em um arquivo compactado, que pode incluir:

- *Bits* extra necessário para salvar $\hat{v}$ com um número inteiro de *bytes*.
- Um número fixo ou variável de *bits* que representa o número de símbolos codificados.
- Informações sobre as probabilidades (p ou c)

Assumindo que a sobrecarga total é um número positivo σ $bits$, conclui-se de (2.17)

que o número de *bits* por símbolo usado para codificar uma sequência S deve ser limitado por:

$$B_s \leq \frac{\sigma - log_2(l_N)}{N} \ bits/simbolo. \tag{2.28}$$

Segue de (2.16) que

$$l_N = \prod_{k=1}^{N} p(s_k), \tag{2.29}$$

e assim

$$B_s \leq \frac{\sigma - \sum_{k=1}^{N} log_2 p(s_k)}{N} \ bits/simbolo. \tag{2.30}$$

Definindo $E\{.\}$ como o operador de valor esperado, o número esperado de *bits* por símbolo é

$$\underline{B} = E\{B_s\} \leq \frac{\sigma - \sum_{k=1}^{N} E\{log_2 p(s_k)\}}{N} = \frac{\sigma - \sum_{k=1}^{N} \sum_{m=0}^{M-1} p(m) log_2 p(m)}{N} \leq H(\Omega) + \frac{\sigma}{N} \tag{2.31}$$

Uma vez que o número médio de bits por símbolo não pode ser menor que a entropia, tem-se

$$H(\Omega) \leq \underline{B} \leq H(\Omega) + \frac{\sigma}{N}, \tag{2.32}$$

e segue-se que

$$\lim_{N\to\infty}\{\underline{B}\} = H(\Omega), \qquad (2.33)$$

o que significa que a codificação aritmética de fato atinge o desempenho de compressão ideal.

Neste ponto, pode-se perguntar porque a codificação aritmética cria intervalos, em vez de código único valores. A resposta está no fato de que a codificação aritmética é ideal não apenas para binários, mas sim para qualquer alfabeto de saída. No intervalo final, encontram-se os diferentes códigos valores ideais para cada alfabeto de saída. Aqui está um exemplo de uso com não binários saídas.

Pode-se ter uma fase inicial no processo que consiste na transformação do dado para posteriormente modificá-lo.

2.4 Técnicas de transformações

A transformação de dados, também chamado processamento inicial ou de mapeamento de dados, ocorre antes da fase de codificação. Pode-se aplicar várias técnicas de transformação, sozinhas ou mesmo encadeadas, onde isso resulta, dependendo da técnica que se aplicar, em um alfabeto menor ou até mesmo em um alfabeto maior, com quantidade menor de caracteres. Segundo MCANLIS; HAECKY (2016), a técnica de transformação omite dados ou palavras, que geralmente são informações repetidas ou supérfluas. A Figura 3 ilustra esse processo, em que a sequência genética é transformada em um código binário, onde A=00, G=01, C=10 e T=11.

2.4.1 Naïve-bit encoding

Segundo GRUMBACH; TAHI, (1993), a Codificação *Naïve-bit* ou "codificação binária ingênua", é uma técnica que tem como abordagem explorar a codificação de comprimento fixo de dois ou mais símbolos em um único *byte*. Essa transformação, ilustrada na Figura 2, consiste em aplicar uma simples substituição dos quatro caracteres do alfabeto do DNA {A, C, G, T} por dois *bits* cada, por exemplo, A=00, C=01, G=10 e T=11.

2.4.2 Hexadecimal

Dada uma sequência genômica, a mesma passa pelo processo de transformação binária (*Naïve-bit*). Após isso, é criado uma biblioteca (ilustrada na Tabela 3), em que na primeira linha tem-se os binários e na segunda linha da tabela o seu correspondente em hexadecimal. O algoritmo vai percorrendo a sequência a cada 4 elementos, verificando o seu correspondente na Tabela 1 e realizando a substituição. No fim do processo, obtém-se a correspondente a sequência hexadecimal. A Figura 11 ilustra as etapas da transformação Hexadecimal.

Tabela 3: Biblioteca utilizada para substituir os binários pelo seu hexadecimal correspondente.

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

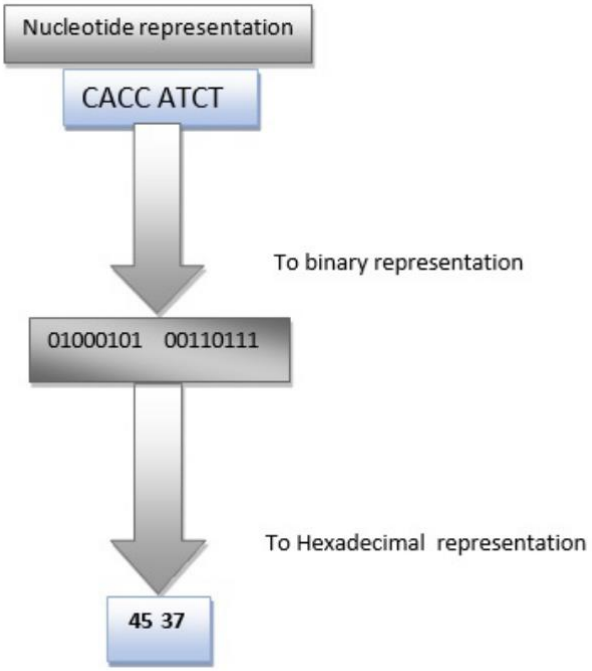

Figura 11: Processo de Transformação de uma sequência genética para um código Hexadecimal. Da fonte (Saada et al., 2015)

2.4.3 Move-to-Front

O algoritmo *Move-to-Front* (MTF) converte os dados em uma sequência de inteiros, com a expectativa de que os valores dos inteiros sejam pequenos e possam ser efetivamente transformados, usando um algoritmo de codificação estatística como Huffman ou de codificação aritmética. O método MTF opera da seguinte forma: primeiro é inicializada uma lista *E* com cada letra do alfabeto. Em seguida, as letras são lidas uma de cada vez, e para cada caractere *A* lido, é anotado o índice de *A* na lista *E* e *A* é movido para a primeira posição da lista *E*. Assim, a lista MTF vai ter diferentes permutações das letras do alfabeto à medida que S é processado e obtém uma sequência de índices. A propriedade mais importante dessa técnica é que os símbolos usados recentemente estão próximos do início da lista. Símbolos iguais aparecerão frequentemente próximos uns dos outros nos dados de saída, portanto, esses símbolos serão convertidos em inteiros pequenos (BENTLEY et al., 1986).

3 Materiais e Métodos

Nesta seção, serão descritos os materiais e métodos utilizados para a realização deste trabalho, cujo objetivo é apresentar uma nova abordagem para o processo de compressão horizontal de sequências genômicas, sem perda de dados, baseada na predição do próximo caractere de uma sequência genômica. Nesta linha, iniciar-se-á a apresentação da arquitetura de um modelo de predição baseado em aprendizagem profunda.

3.1 Arquitetura DeepRLE

A Figura 12 apresenta a arquitetura *DeepRLE*, a arquitetura está dividida em três etapas, a primeira etapa é de aprendizagem profunda, onde treinamos o modelo de predição do próximo caractere da sequência (seção 3.1.1), a segunda parte dessa arquitetura seria o processo de compressão ou descompressão da sequência, onde aplicamos um algoritmo de verificação para checar se a predição foi correta (seção 3.1.2 e 3.1.3) e a última etapa é o processo de transformação (seção 3.1.4)

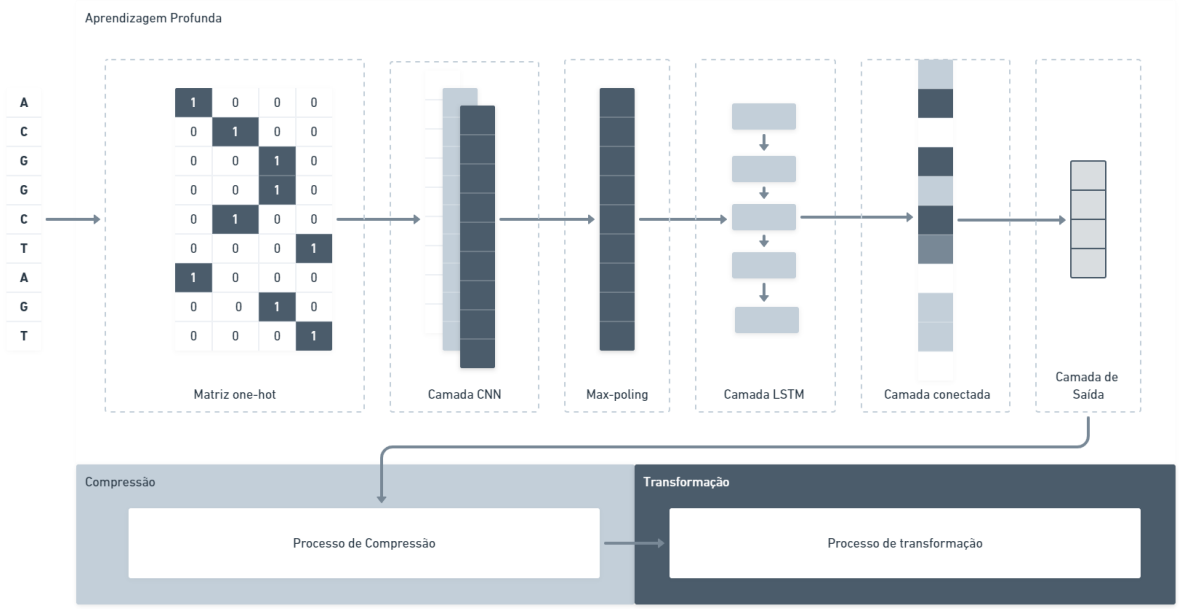

Figura 12: Arquitetura da abordagem DeepRLE para predição da próxima base de uma sequência.

3.1.1 Aprendizagem profunda

Nesta etapa da arquitetura DeepRLE criamos a arquitetura utilizando algoritmos de aprendizagem profunda para predição do próximo caractere da sequência, ela foi baseada na arquitetura *DeepDNA* proposto por WANG et al. (2018), A Figura 12 ilustra as camadas:

1. As bases da sequência de entrada são convertidas em uma matriz one-hot.
2. Algoritmo CNN verifica os recursos locais da sequência (seção 2.2.1 e seção 3.1.1);
3. A camada de max-pooling extrai recursos da sequência
4. A camada de LSTM captura a dependência de longo prazo da sequência (seção 2.2.2)
5. A camada conectada integra informações da camada anterior;
6. A camada de saída sigmóide calcula a probabilidade das saídas para cada base da sequência de forma individual;

A camada da rede neural convolucional (CNN), rede de memória de curto e longo prazo (LSTM) e camada totalmente conectada executam uma transformação pela função de ativação linear retificada antes da saída da camada anterior.

Basicamente, a CNN envolve um conjunto de filtros, todos deslizando sobre a sequência do genoma e detectando recursos em posições diferentes. Seja $x_i \in R^4$ o vetor one-hot em 4- dimensões para a i-ésima base em uma sequência. As representações do vetor one-hot para diferentes bases são definidas na Figura 13.

$$A : \{1, 0, 0, 0\}$$
$$C : \{0, 1, 0, 0\}$$
$$G : \{0, 0, 1, 0\}$$
$$T : \{0, 0, 0, 1\}.$$

Figura 13: Representação do vetor one-hot para as bases ACTG.

Seja $x \in R^{N\times 4}$ a sequência de entrada do genoma, onde N é o comprimento da sequência. O vetor de filtro convolucional é definido como $\omega_j \in R^{k\times 4}$, onde k é o tamanho da janela do filtro. A saída para vetores de filtro é $o \in R^{(N-k+1)\times m}$, onde m é o número de vetores de filtro. Cada elemento $o_{i,j}$ do mapa de recursos para o vetor de filtro, ω_j, é produzido da seguinte forma:

$$o_{i,j} = \delta(\omega_j \odot [x_i, x_{i+1,}\ldots, x_{i+k-1}] + b_j) \qquad (3.1)$$

onde $b_j \in R$ é um valor compartilhado para o viés correspondente para filtrar o vetor ω_j, e $\odot$ é chamado de operação de convolução definido como:

$$\omega_j \odot [x_i, x_{i+1,}\ldots, x_{i+k-1}] = \sum_{n=0}^{k-1} \omega_{nj} x_{i+n.} \qquad (3.2)$$

Escolhemos o ReLU [19] como a ativação neural não linear função δ, que dimensiona o valor de saída negativo para 0.

$$ReLU(x) = \begin{cases} x \ \ if \ x \geq 0 \\ 0 \ \ if \ x < 0 \end{cases} \tag{3.3}$$

Uma operação de max-pooling é aplicada para extração de recursos após a saída da camada CNN para selecionar o recurso mais importante, e isso permite o aprendizado de recursos de sequência em um nível superior à escala espacial na próxima camada. Cada unidade na camada max-pooling resume um valor máximo em um tamanho fixo da região de recurso.

Uma camada de eliminação é conectada para reduzir o sobreajuste e fornece uma maneira de combinar de forma eficiente diferentes arquiteturas de rede neural existentes no momento do treinamento. O termo "Dropout" refere-se a unidades removidas temporariamente (ocultas e visíveis) em uma rede neural, eliminando todas as suas conexões de saída. A escolha de quais unidades descartar é aleatório com uma probabilidade fixa, p, independente das unidades.

3.1.2 Algoritmo de compressão

Nesta seção será descrito o algoritmo para compressão das sequências utilizando um modelo de aprendizagem profunda. Na Figura 14 é apresentado o algoritmo para compressão da sequência, onde o algoritmo recebe como argumento uma sequência que deverá ser comprimida e o modelo de aprendizagem profunda para predição do próximo base da sequência. Após receber esses argumentos, a sequência é percorrida, onde o *loop* começa após a base de índice 64, no *loop* é capturado o *label* (linha 5 da Figura 14) que é o valor correto da predição e as 64 bases anteriores (linha 6 da Figura 14) a variável *label*, após isso é executado o modelo para realizar a predição das 64 bases (linha 7 da Figura 15). A função predict_next_char (cf. Figura 15) é executada e recebe como argumento as 64 bases, o valor

correto do próximo caractere das 64 bases e o modelo que foi treinando para predição do próximo caractere. A função retornará *True* caso o valor retornado pelo modelo de predição acerte o valor do próximo caractere das 64 bases.

```
1  def compress(sequence, model):
2      sequence_model = ''
3      aux = ''
4      for i in range(0, len(sequence) - 64):
5          label = sequence[i+64]
6          sequence_64 = sequence[i:i+64]
7
8          result_prediction = pnc.prediction_next_char(sequence_64, label=label, model=model)
9
10         if (result_prediction == True):
11             aux = aux + '1'
12         else:
13             aux = aux + '00' + character_to_binary(result_prediction)
14
15     sequence_model = aux
16     return sequence_model
17
```

Figura 14: Algoritmo para compressão a sequência utilizando aprendizagem profunda.

Essa verificação é realizado na linha 18 da Figura 15, onde é verificado se a base retornada pelo modelo é igual ao valor do *label*, continuando o processo de compressão. Após receber o retorno da função predict_next_char, é verificado se a predição acertou, caso tenha acertado o valor ‘1’ é concatenado a variável aux (linha 11 da Figura 14), caso contrário é concatenado o valor de ‘00’ com o binário correspondente a base seguindo o seguinte critério: o valor do binário correspondente a base A é igual a ‘00’, a base C é igual a ‘01’, a base T igual a ‘10’ e a base G igual a ‘11’. Após ter percorrido toda a sequência é retornado uma *string* contendo apenas os caracteres ‘0’ e/ou ‘1’.

3.1.3 Algoritmo de descompressão

Nesta seção será descrito o algoritmo para descompressão de sequências mitocondriais utilizando um modelo de aprendizagem profunda. Na Figura 16 é apresentado o processo de descompressão, em que ele recebe como argumento a sequência retornada no processo de compressão, as 64 bases da sequência original e o modelo de Deep Leaning para predição do próximo base da sequência. A primeira etapa é atribuir as 64 bases da sequência inicial a variável *sequence_origin* (linha 2 da Figura 16), e após isso percorre-se a sequência retornada no processo de compressão. Caso o valor do índice da vez seja igual a '1' é executado o modelo para realizar a predição das 64 bases (linha 7 da Figura 15). A função predict_next_char (cf. Figura 15) recebe como argumento as últimas 64 bases da sequência original. Ou seja, a variável *sequence_origin* também recebe um argumento para dizer que se está no processo de descompressão e por último deve ser passado o modelo que foi treinando para predição do próximo caractere. E como retorno da função predict_next_char tem-se o caractere correto da sequência (linha 16 da Figura 15). Após isso, no processo de descompressão concatena-se esse retorno a sequência original (linha 9 da Figura 16).

```
def prediction_next_char(sequence_64, label=None, decode=False):

    X = one_hot_encoder(sequence_64, ALLOWED_SYMBOLS)

    y_pred = model.predict(X[np.newaxis,...])

    y_pred = np.squeeze(y_pred)

    y_pred_max = y_pred.max(axis=-1,keepdims=1) == y_pred

    y_pred_index = np.argmax(y_pred_max)

    next_char_pred = ALLOWED_SYMBOLS[y_pred_index]

    if decode == True:
      return next_char_pred

    if label == next_char_pred :
      return True

    return label

```

Figura 15: Algoritmo para fazer predição do próximo caractere.

Caso o valor do índice da vez seja diferente de ‘1’ é capturado o valor dos próximos 4 caracteres da sequência do processo de compressão. Continua-se seguindo o critério para converter os 4 caracteres em base: caso esse valor seja igual a ‘0000’ a base correspondente é ‘A’, caso seja ‘0001’ o valor correspondente é igual a ‘C’, caso seja ‘0010’ o valor correspondente é ‘T’ e caso seja ‘0011’ o valor correspondente é ‘G’. Após isso, o valor é concatenado à sequência original. Após ter percorrido toda a sequência retornada pelo modelo é obtido uma string contendo apenas os caracteres ‘A’, ‘C’, ‘T’ e/ou ‘G’, e essa sequência será igual à sequência original informada no processo de compressão.

```
def decompress(sequence_output_model, sixty_four_sequence_start, model):
    sequence_origin = sixty_four_sequence_start
    i = 0
    j = 0
    while (j < len(sequence_output_model)):
        bit_sequence_model = sequence_output_model[j]
        if (bit_sequence_model == '1'):
            r = pnc.prediction_next_char(sequence_origin[i:], decode=True, model=model)
            sequence_origin = sequence_origin + r
            j = j + 1
        else:
            sequence_origin = sequence_origin + code_table_binary_to_actg(sequence_output_model[j:j+4])
            j = j + 4
        i = i + 1
    return sequence_origin

```

Figura 16: Algoritmo para descompressão a sequência utilizando aprendizagem profunda.

3.1.4 Algoritmo para transformação

Na seção 2.4 foram apresentadas algumas técnicas de transformações e o porquê da sua utilização. Deve-se notar que a sequência retornada após o processo de compressão possui apenas dois caracteres, ‘0’ e/ou ‘1’, e que o modelo de predição em sua maioria sempre acerta o próximo caractere, ou seja, tem-se uma grande quantidade de caracteres ‘1’ em sequência.

Na literatura tem-se o algoritmo de codificação *run-length* (RLE). O RLE armazena sequências longas de valores repetidos como um único valor, por exemplo, a ocorrência ‘111111111’ é possível substituir sua representação pelo par (9, 1), ou seja, tem-se 9 caracteres ‘1’ em sequência e se inicia na posição 1.

```python
def rle_encode(sequence, tamanho_binary):
    cont = 2
    lastSymbol = sequence[0]
    newSequence = ''
    for i in range(1, len(sequence)):
        if lastSymbol != sequence[i] or cont == 2 ** tamanho_binary:
            newSequence = newSequence + lastSymbol + format(cont-1, 'b').zfill(tamanho_binary)
            cont = 1

        cont = cont + 1
        lastSymbol = sequence[i]
    newSequence = newSequence + lastSymbol + format(cont-1, 'b').zfill(tamanho_binary)
    return newSequence


```

Figura 17: Algoritmo para realizar a transformação baseado no algoritmo RLE

```python
def rle_decode(sequence, tamanho_binary):
    MAX_BASE_COMMIT = tamanho_binary + 1
    newSequence = ''
    sequence_split = [sequence[i:i+MAX_BASE_COMMIT] for i in range(0, len(sequence), MAX_BASE_COMMIT)]
    for sequence_split_item in sequence_split:
        if len(sequence_split_item) > 1:
            repeatSequence = sequence_split_item[0] * int(sequence_split_item[1:], 2)
            newSequence = newSequence + repeatSequence
    return newSequence

```

Figura 18: Algoritmo para reverter a transformação baseado no algoritmo RLE

Pensando em nosso cenário de reduzir o máximo possível o número de caracteres, adaptou-se o algoritmo do RLE para o problema em questão. Na Figura 17 apresenta-se o algoritmo, em que recebe como argumento a sequência retornada no processo de compressão (Figura 14) e a base elevado a 2 (tamanho_binary) que representará a quantidade de caracteres repetidos em sequências que representa em um outro valor, por exemplo, se o for informado a base sendo igual a 5 o valor correspondente de bits será 32 bits, ou seja, para toda sequência repetida de tamanho 32 representa-se em um outro valor.

Para cada vez que é encontrado uma quantidade de caracteres repetidos em sequência do tamanho especificado no argumento (linha 6 da Figura 17), transforma-se o tamanho da sequência repetido para um binário e substitui-se a sequência repetida pelo valor binário do tamanho da sequência (linha 7 da Figura 17). Desta forma, reduz-se consideravelmente a quantidade de caracteres repetidos em sequência da sequência retornada pelo processo de compressão (Figura 14).

Na Figura 18 apresenta-se o algoritmo para reverter a transformação. Esse algoritmo recebe como argumento a sequência retornada pelo processo de transformação (Figura 17) e a base elevada a 2 (tamanho_binary) que representará a quantidade de caracteres repetidos em sequências que será representada em outro valor. Como retorno tem-se a sequência do processo de compressão (Figura 14) antes do processo de transformação.

3.2 Seleção das Métricas

Com intuito de descobrir o quanto pode-se economizar de espaço após o processo de compressão utilizando a abordagem de predição do próximo caractere, realizou-se o cálculo de economia de espaço (Equação 3.4).

$$Economia\ Espaço\ = \left(1\ -\ \frac{CFS}{UFS}\right) \times 100 \qquad (3.4)$$

onde UFS é o tamanho do arquivo não compactado e CFS é o tamanho do arquivo compactado.

Além disso, outra métrica utilizada neste trabalho é a taxa de compressão alcançada (CA), dada pela Equação 3.5, na qual tem-se o tamanho do arquivo após a compressão, medido em *bits*, dividido pela quantidade total dos símbolos que representam as bases nitrogenadas presentes na sequência genômica utilizada no processo.

$$CA = \frac{Tamanho\ arquivo\ após\ compressão}{quantidade\ de\ símbolos\ nucleotídeos\ presentes\ na\ sequência} \quad (3.5)$$

3.3 Seleção do dataset

Para realização dos experimentos, utilizou-se o dataset disponibilizado por WANG et al., (2018), onde foram baixadas do banco de dados MITOMAP (http://www.mitomap.org) 1000 sequências humanas mitocondriais completas. O DNA mitocondrial (mtDNA) é formado por uma fita dupla de DNA em formato de hélice, e contém 37 genes, formados a partir de 16.569 pares de bases.

A escolha deste *dataset* nos permite comparar os resultados com WANG et al., (2018). Experimentou-se o nosso método deepRLE em 1000 sequências de genoma mitocondrial humano completo, dividido aleatoriamente em três conjuntos de dados: conjunto de treinamento 70%, conjunto de validação 20% e conjunto de teste 10%.

3.4 Seleção das Ferramentas

Para realização dos experimentos foi executada sobre uma máquina com sistema operacional Windows 10 Pro, com 12GB de memória RAM e um processador Intel(R) Core(TM) i5-7300HQ CPU @2.50 GHz 2.50GHz.

4 Experimentos e Análise dos Resultados

Nesta seção apresentam-se os testes obtidos no desenvolvimento da pesquisa com o propósito de avaliar a sua viabilidade, bem como realizar a respectiva análise e comparação dos resultados obtidos. Para realização deste experimento foi treinado um modelo de aprendizagem profundo apresentado na seção 3.1.1, os parâmetros detalhados de nossa arquitetura de modelo são definidos da seguinte forma:

- Camada convolucional (1024 filtros, tamanho da janela: 24, tamanho do passo: 1)
- Camada de max-pooling (tamanho da janela: 3, tamanho do passo: 1)
- Camada de exclusão (probabilidade: 0,1)
- Camada de rede de memória de curto e longo prazo (LSTM) (256 neurônios LSTM)
- Camada de *Dropout* (probabilidade: 0,2)
- Camada totalmente conectada (1024 neurônios)
- Camada de saída sigmóide (4 neurônios)

Todos os pesos são inicializados aleatoriamente de $unif\,(-0.05,0.05)$, e todos os vieses são inicialmente definidos como 0. Foi utilizado treinamento em *mini lote* e comprimento de sequência de entrada com tamanho 64 para minimizar a entropia cruzada binária multitarefa média no conjunto de dados de treinamento. O *dropout* é avaliado no final de cada período de treinamento para monitorar a convergência. Levou-se 10 épocas para treinar totalmente, e cada época de treinamento levou cerca de 1h. O *dataset* foi dividido aleatoriamente em três conjuntos de dados: conjunto de treinamento 70%, conjunto de validação 20% e conjunto de teste 10%. O modelo de predição do próximo caractere teve uma acurácia de 0.9981 e perda de 0.0094.

No experimento foram utilizadas apenas sequências humanas mitocondriais e não considerou-se o cabeçalho do arquivo FASTA. Nesta etapa de experimentação foi utilizado o conjunto de teste que possui 100 sequências selecionadas de forma aleatória do *dataset*.

Tabela 4: Resultado dos experimentos após o processo de compressão utilizando o DeepRLE.

Quantidade de caracteres repetidos (bits)	Média de caracteres após compressão	Máximo	Mínimo	Economia de Espaço (%)
8	9642	10012	9508	41.79
16	5776	6255	5600	65.13
32	3526	4116	3306	78.71
64	2227	2912	1967	86.55
128	1495	2312	1184	90.97
256	1101	2034	747	93.34
512	916	2040	510	94.70
1024	858	2156	396	94.85
2048	854	2340	348	94.84
4096	902	2535	325	94.55
8192	969	2730	350	94.14

A Tabela 4 apresenta os resultados obtidos no processo de compressão utilizando a arquitetura DeepRLE. Na primeira coluna tem-se a quantidade de caracteres repetidos em sequência. Na segunda coluna tem-se a média de caracteres após o processo de compressão. A terceira coluna especifica o valor máximo obtido por cada quantidade de caracteres repetidos do conjunto de teste. A quarta coluna especifica o valor mínimo obtido por cada quantidade de caracteres repetidos do conjunto de teste e a quinta coluna especifica a porcentagem de economia de espaço após o processo de compressão.

Percebeu-se que a quantidade de caracteres repetidos que obteve o melhor resultado depois da transformação, considerando a economia de espaço, foi com 1024 bits, que obteve, 94.85% de economia, seguido por 2048 bits, que obteve 94.84% de economia. Na sequência, tem-se 512 bits, que obteve 94.70% de economia de espaço.

A Figura 19 apresenta um gráfico que mostra a relação entre a quantidade de caracteres repetidos em sequência com a economia de espaço, em que percebe-se que quanto mais bits são utilizados para a quantidade de caracteres em sequências, melhores são os resultados. Entretanto, percebe-se que a partir de 4096 bits a economia de espaço resultante não é satisfatório.

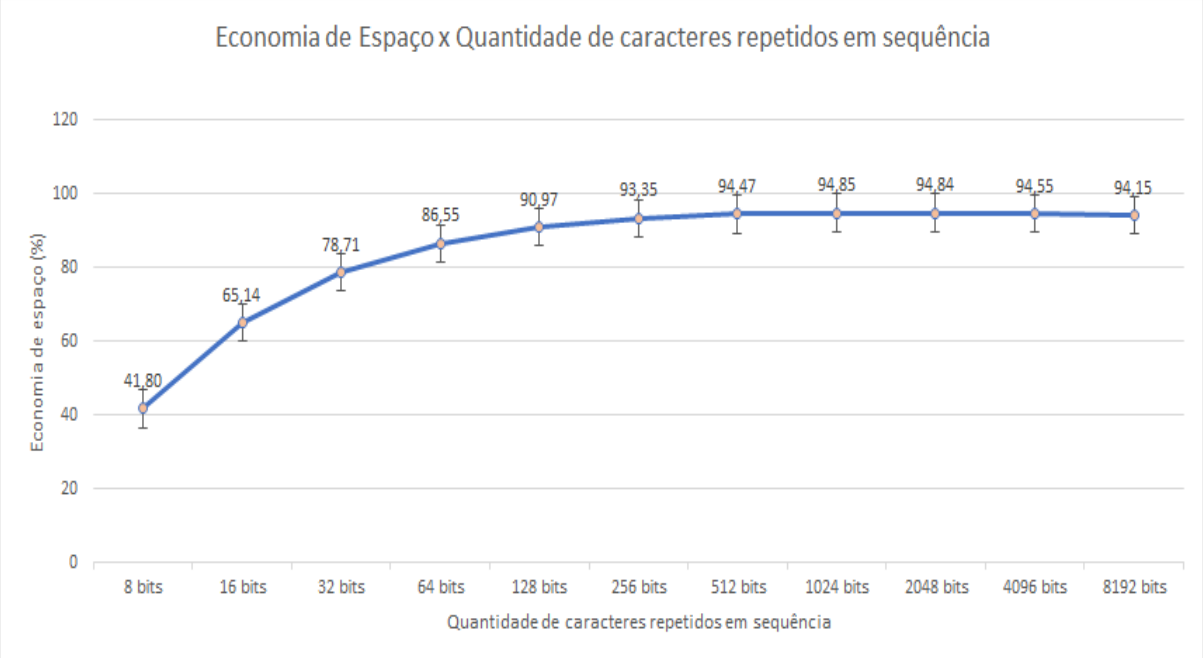

Figura 19: Resultados obtidos no experimento aplicando a equação de economia de espaço em relação à quantidade de caracteres repetidos em sequência.

Percebe-se também que no gráfico da Figura 20 acontece o mesmo cenário que no gráfico da Figura 19. Porém, neste gráfico quanto menor for resultado, melhor é a taxa de compressão.

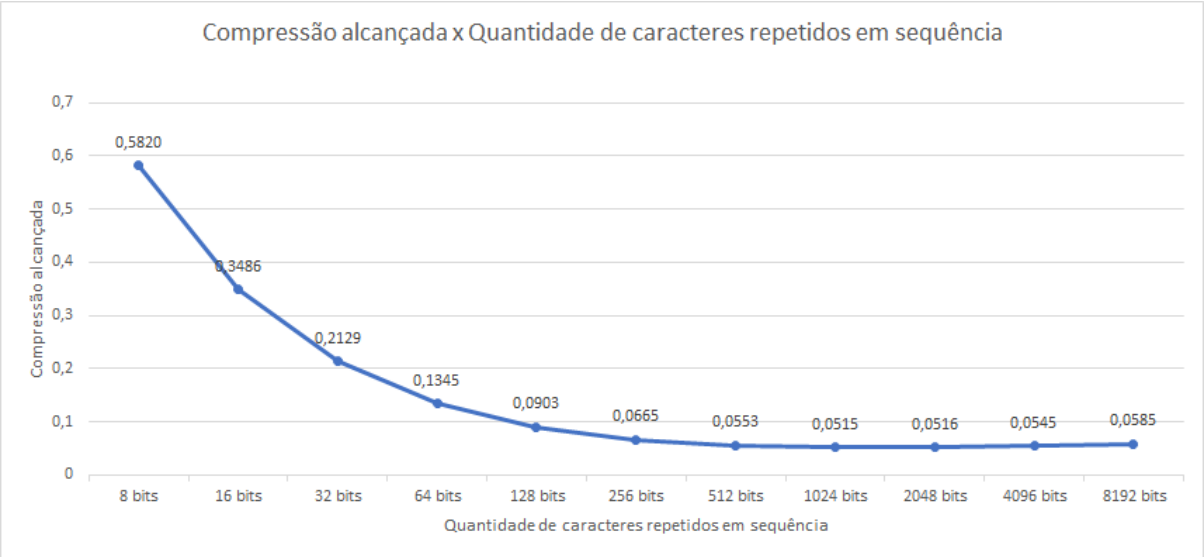

Figura 20: Resultados obtidos no experimento aplicando a equação de compressão alcançada em relação à quantidade de caracteres repetidos em sequência.

Já no gráfico da Figura 21 tem-se a relação entre a quantidade média de caracteres com a quantidade de caracteres repetidos em sequência. Esse gráfico tem o mesmo comportamento dos outros dois anteriores.

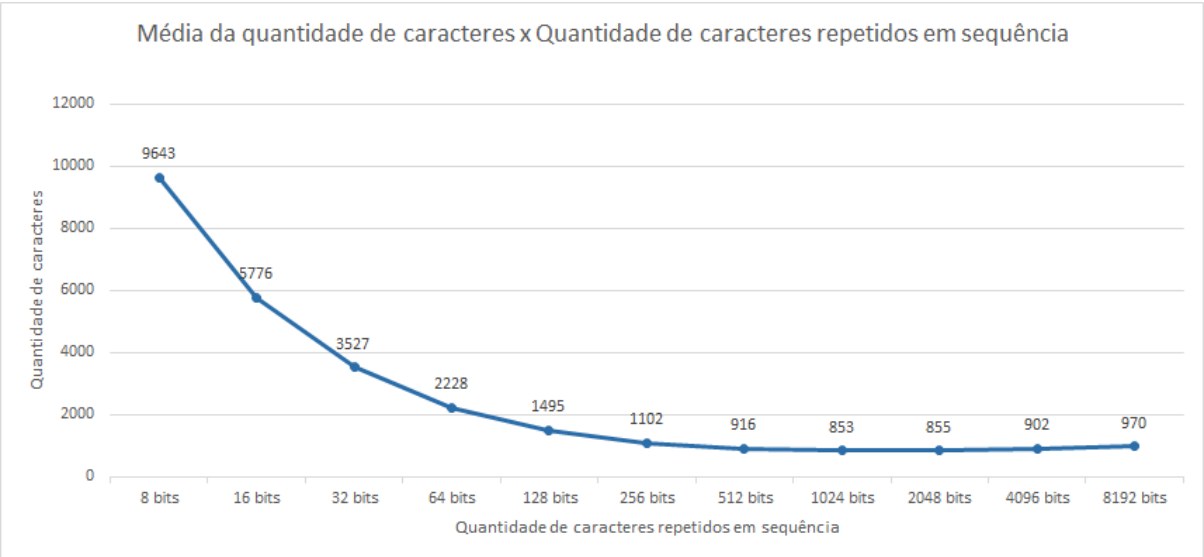

Figura 21: Resultados obtidos no experimento, média da quantidade de caracteres em relação à quantidade de caracteres repetidos em sequência.

Para comparar os resultados obtidos em nossa pesquisa com o estudo de WANG et al. (2018), considerou-se o melhor resultado que se obtive, que foi utilizando a 1024 bits para realizar as transformações, em que se obtive a média de 0.0515 bpb (bits por base), e 94.85% de economia de espaço. Em WANG et. al (2018) o resultado foi de 0.0336 bpb, um resultado superior ao do nosso estudo. Entretanto, ficou alguns pontos a serem respondidos, primeiro foi a métrica utilizada para chegar nesse resultado, uma vez que, no artigo não apresenta essa informação.

Por se trabalhar com sequências consideravelmente pequenas, ou seja, cada sequência do mitocondrial humano tem em média 16.569 pares de bases, tem-se um limite de quantidade de caracteres repetidos em sequência. Deve-se notar que a partir de 8192 bits os resultados não são bons, pois em teoria o tamanho da quantidade de caracteres repetidos é maior que a quantidade de caracteres da sequência original. Essa mesma abordagem de compressão do DeepRLE pode ser aplicada em outras sequências, por exemplo, o Chromosome 22. Entretanto, para que o algoritmo consiga uma boa economia de espaço é preciso que o modelo de tenha uma boa acurácia, para que quando é passado esse modelo para o algoritmo de compressão do DeepRLE, ele consiga retornar vários caracteres ‘1’ em sequência, permitindo que o processo de transformação possa utilizar mais caracteres repetidos em sequência, ou seja, acima de 8192 bits e ainda obter um resultado satisfatório, que consequentemente indicará ganho na economia de espaço.

5 Conclusão

Este estudo teve como objetivo apresentar uma nova abordagem para o processo de compressão de sequências genômicas, sem perda de dados, baseada em um método de predição do próximo caractere. Obteve-se resultados próximos do esperado em comparação a outras ferramentas que utilizam o método de predição do próximo caractere. Entretanto, em comparação com ferramentas tradicionais de compressão os resultados são excelentes.

Considerando o *dataset* utilizado neste estudo, pode-se concluir que para as sequências de mitocôndrias humanas é melhor utilizar 1024 bits para o processo de transformação, visto que acima disso, os resultados de economia de espaço tendem a aumentar. Porém, como propostas futuras, pode-se adaptar o valor da quantidade de caracteres repetidos em sequência conforme a sequência original. Dessa forma, sempre ter-se-á o melhor resultado para a sequência em questão no processo de transformação.

Em relação à comparação ao estudo de WANG et. al (2018), obteve-se um resultado inferior. Porém, isso se deu em função da forma como é aplicada a métrica de bits por base, onde em seu estudo não apresenta o cálculo, e foram feitos testes locais com o algoritmo proposto por eles. Entretanto, os resultados não foram idênticos ao do artigo.

A abordagem DeepRLE pode ser utilizada com outros modelos de predição do próximo caractere. Entretanto, é preciso ter-se uma atenção especial sobre a acurácia que o modelo produzirá, visto que, se a acurácia for baixa, teoricamente ter-se-á uma sequência onde os caracteres ‘1’ serão distribuídos de forma aleatória sem apresentar uma grande quantidade repetidas em sequência, fazendo com que o processo de transformação não obtenha um resultado satisfatório para um ganho considerável de espaço.

Como trabalhos futuros, pretende-se:

a) criar outros modelos para a compressão, utilizando a abordagem DeepRLE;

b) aplicar outras formas de transformação com intuito de obter melhores resultados na economia de espaço.

6 Referências Bibliográficas

AI Based Software Development Company In Navi Mumbai. Disponível em: <https://aakeria.com/blog/Ml_AI_DL_Blog.html>. Acesso em: 10 ago. 2021.

B. a. J. Z. Saada, "**Vertical DNA sequences compression algorithm based on hexadecimal representation,**" Proceedings of the World Congress on Engineering and Computer Science. Vol. 2, 2015

BELL, T. C.; CLEARY, J. G.; WITTEN, I. H. **Text Compression**. [s.l.] Englewood Cliffs, N.J. : Prentice Hall, 1990.

BENTLEY, J. L. et al. A locally adaptive data compression scheme. **Communications of the ACM**, v. 29, n. 4, p. 320–330, abr. 1986.

BIJI, C. L.; NAIR, A. S. Benchmark Dataset for Whole Genome Sequence Compression. **IEEE/ACM Transactions on Computational Biology and Bioinformatics**, v. 14, n. 6, p. 1228–1236, 1 nov. 2017.

BOWDEN, R.; MITCHELL, T. A.; SARHADI, M. Cluster based nonlinear principle component analysis. **Electronics Letters**, v. 33, n. 22, p. 1858, 1997.

GOYAL, M. et al. **DeepZip: Lossless Data Compression Using Recurrent Neural Networks**. 2019 Data Compression Conference (DCC). **Anais**...IEEE, mar. 2019Disponível em: <http://dx.doi.org/10.1109/dcc.2019.00087>. Acesso em: 10 ago. 2021

GRUMBACH, S.; TAHI, F. **Compression of DNA sequences**. [Proceedings] DCC `93: Data Compression Conference. **Anais**...IEEE Comput. Soc. Press, [s.d.] Disponível em: <http://dx.doi.org/10.1109/dcc.1993.253115>. Acesso em: 10 ago. 2021

HOCHREITER, S.; SCHMIDHUBER, J. Long Short-Term Memory. **Neural Computation**, v. 9, n. 8, p. 1735–1780, 1 nov. 1997.

Introduction to Deep Learning. Disponível em: <https://www.geeksforgeeks.org/introduction-deep-learning/>. Acesso em: 10 ago. 2021.
JELINEK, F. **Probabilistic Information Theory: Discrete and Memoryless Models**. [s.l: s.n.].

KISELEV, V. Y.; ANDREWS, T. S.; HEMBERG, M. Challenges in unsupervised clustering of single-cell RNA-seq data. **Nature Reviews Genetics**, v. 20, n. 5, p. 273–282, 7 jan. 2019.

KROGEL, M.-A.; SCHEFFER, T. Multi-Relational Learning, Text Mining, and Semi-Supervised Learning for Functional Genomics. **Machine Learning**, v. 57, n. 1/2, p. 61–81, out. 2004.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, v. 521, n. 7553, p. 436–444, 27 maio 2015.

LIU, J. et al. Application of deep learning in genomics. **Science China Life Sciences**, v. 63, n. 12, p. 1860–1878, 10 out. 2020.

MARTINS, J. V. et al. **An Approach to Compression of Genomic Data Based on Image File Format**. 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC). **Anais**...IEEE, out. 2018Disponível em: <http://dx.doi.org/10.1109/smc.2018.00554>. Acesso em: 10 ago. 2021

MCANLIS, C.; HAECKY, A. **Understanding Compression: Data Compression for Modern Developers**. [s.l.] “O’Reilly Media, Inc.,” 2016.

MOUNT, D. W. **Bioinformatics: Sequence and Genome Analysis**. [s.l.] CSHL Press, 2004.

REUTER, J. A.; SPACEK, D. V.; SNYDER, M. P. High-Throughput Sequencing Technologies. **Molecular Cell**, v. 58, n. 4, p. 586–597, maio 2015.

RUBIN, F. Arithmetic stream coding using fixed precision registers. **IEEE Transactions on Information Theory**, v. 25, n. 6, p. 672–675, nov. 1979.

SAINATH, T. N. et al. **Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks**. 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). **Anais**...IEEE, abr. 2015Disponível em: <http://dx.doi.org/10.1109/icassp.2015.7178838>. Acesso em: 10 ago. 2021

SALOMON, D. **Data Compression: The Complete Reference**. [s.l.] Springer Science & Business Media, 2007.

SAYOOD, K. **Lossless Compression Handbook**. [s.l: s.n.].

SCHUSTER, S. C. Next-generation sequencing transforms today's biology. **Nature Methods**, v. 5, n. 1, p. 16–18, 19 dez. 2007.

SHANNON, C. E. A Mathematical Theory of Communication. **Bell System Technical Journal**, v. 27, n. 4, p. 623–656, out. 1948.

SITORUS, I. An Introduction to Convolution Neural Network (CNN) for A Beginner. **Easyread**, 31 mar. 2020.

Understanding LSTM Networks -- colah's blog. Disponível em: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Acesso em: 25 ago. 2021.

VINYALS, O. et al. **Show and tell: A neural image caption generator**. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). **Anais**...IEEE, jun. 2015Disponível em: <http://dx.doi.org/10.1109/cvpr.2015.7298935>. Acesso em: 10 ago. 2021

WANG, R. et al. **DeepDNA: a hybrid convolutional and recurrent neural network for compressing human mitochondrial genomes**. 2018 IEEE International Conference on

Bioinformatics and Biomedicine (BIBM). **Anais**...IEEE, dez. 2018Disponível em: <http://dx.doi.org/10.1109/bibm.2018.8621140>. Acesso em: 10 ago. 2021

ZHOU, C. et al. **A C-LSTM Neural Network for Text Classification**. Disponível em: <https://arxiv.org/abs/1511.08630>.

ZIV, J.; LEMPEL, A. A universal algorithm for sequential data compression. **IEEE Transactions on Information Theory**, v. 23, n. 3, p. 337–343, maio 1977.

ZIV, J.; LEMPEL, A. Compression of individual sequences via variable-rate coding. **IEEE Transactions on Information Theory**, v. 24, n. 5, p. 530–536, set. 1978.