Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise de modelos de rede neuronal para previsão de próximo pixel de imagem binarizada

João G. Bispo

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Ricardo L. de Queiroz

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise de modelos de rede neuronal para previsão de próximo pixel de imagem binarizada

João G. Bispo

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Ricardo L. de Queiroz (Orientador)
CIC/UnB

Prof. Dr. Eduardo Peixoto Fernandes da Silva
EnE/UnB

Prof. Dr. Diogo Caetano Garcia
CIC/UnB

Prof. Dr. João Luiz Azevedo de Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 17 de julho de 2023

Dedicatória

Eu dedico esse trabalho a minha família que tanto me apoiou durante toda minha jornada acadêmica. Particularmente dedico aos meus pais que nunca pouparam esforços para que eu pudesse chegar onde estou na academia e na vida. Além de sempre me encorajarem, diretamente e pelo exemplo, a procurar a excelência acadêmica que me proporcionou tantas oportunidades, acreditando no poder da educação de moldar a vida do ser humano. Em especial dedico ao meu pai, minha mãe, a Clara, minha vó Marluce, meu tio Pedro, minha vó Ilta, meu vô Antenor e meu vô Iran.

Em segundo eu dedico esse trabalho aos meus amigos mais próximos de dentro e de fora do curso que tanto me ajudaram a manter a cabeça no lugar e lembrar o que é mais importante durante a graduação. A vocês: Duda, Vinão, Ale, Mendel, Palhares, Estevan, Raposo, Keidi, Igor e Eduardo.

E por fim aquela que já é família e também minha melhor amiga, obrigado por tudo que você fez e faz por mim, meu amor, te dedico esse projeto, Isabella.

Agradecimentos

Obrigado primeiramente ao meu orientador, Dr. Ricardo de Queiroz, por acreditar em mim e me guiar durante essa última etapa na graduação. Foi um privilégio participar das reuniões do grupo de 3D e ter a chance de aprender tanto com você e todos os membros do grupo.

Obrigado ao Lucas que me inspirou, guiou e ajudou desde o começo do projeto até a linha final com orientações, informações e até uma seção de "debbuging".

Obrigado a Isabella, minha namorada, que ficou do meu lado durante toda a confecção desse documento, revisando cada paragrafo, cada frase, cada vírgula.

Obrigado Éder e Alice, meus pais, por me apoiarem, principalmente durante as últimas semanas de correria.

E obrigado ao meu primo, Gabriel por me ajudar com minha procura por fontes, e pelas dicas de como melhorar o documento.

Resumo

Nesse estudo vários modelos de redes neuronais foram treinados para prever o próximo pixel em uma imagem binarizada. Modelo de rede neuronal é um algoritmo que tenta simular o comportamento de um cérebro biológico para processar dados e resolver problemas de forma inteligente. Modelos para previsão de pixels de imagens são uteis para a codificação aritmética, um algoritmo que diminui o tamanho dessas imagens para que elas possam ser transmitidas pela web mais rápido. O estudo deverá vir a ajudar profissionais trabalhando com redes neuronais e codificação aritmética a gerar melhores modelos. Particularmente o estudo chega a conclusão que uma taxa de aprendizado de 0.01 é ideal para esse tipo de problema. Para obter resultados que possam complementar os encontrados nesse projeto, outros estudos deveriam treinar e analisar modelos com taxa de aprendizado variável, para ver qual o impacto dessa técnica nos resultados.

Palavras-chave: compressão de dados, inteligencia artificial, rede neural, rede neuronal, classificação

Abstract

In this project several neural network models were trained to predict the next pixel in a binarized image. Neural network models are a technique that tries to simulate the way a biological brain processes information with the objective of creating a "smart" structure capable of learning and solving problems. The problem of predicting the next pixel in an image arises from aritmetic coding, an algorithm that shortens an image's size, so it can be transfered through the web faster. After training such models it was observed that a learning rate of 0.01 resulted in better models overall. Such result should lead to an easier and cheaper time training similar models to solve this particular problem in the future. To extend this project, other studys should try to train models with a scheduler in order to introduce some variance in the learning rate.

Keywords: data compression, artificial inteligence, neural networks, classificacion

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capítulo apresenta a teoria por trás do problema que foi escolhido, com o objetivo de esclarecer a motivação e relevância do projeto desenvolvido.

Na área da compressão de dados, existe uma técnica chamada codificação aritmética que precisa de probabilidades baseadas em contextos para funcionar de forma eficiente. Para gerar essas probabilidades podem ser usadas tabelas de consulta com essa informação, mas para contextos muito grandes essa técnica se torna inviável. Esse projeto vai explorar uma outra forma de gerar essas probabilidades que se mantém eficiente mesmo com contextos muito grandes. Essa forma é modelando o contexto com redes neuronais. Redes neuronais são algoritmos que usam aprendizado de máquina, uma sub-área da inteligencia artificial, para identificar padrões em conjuntos de dados. A Figura 1.1 ilustra esses conceitos. Nas próximas seções será esclarecida a base teórica dos principais tópicos relacionados ao problema para que o leitor possa entendê-lo.

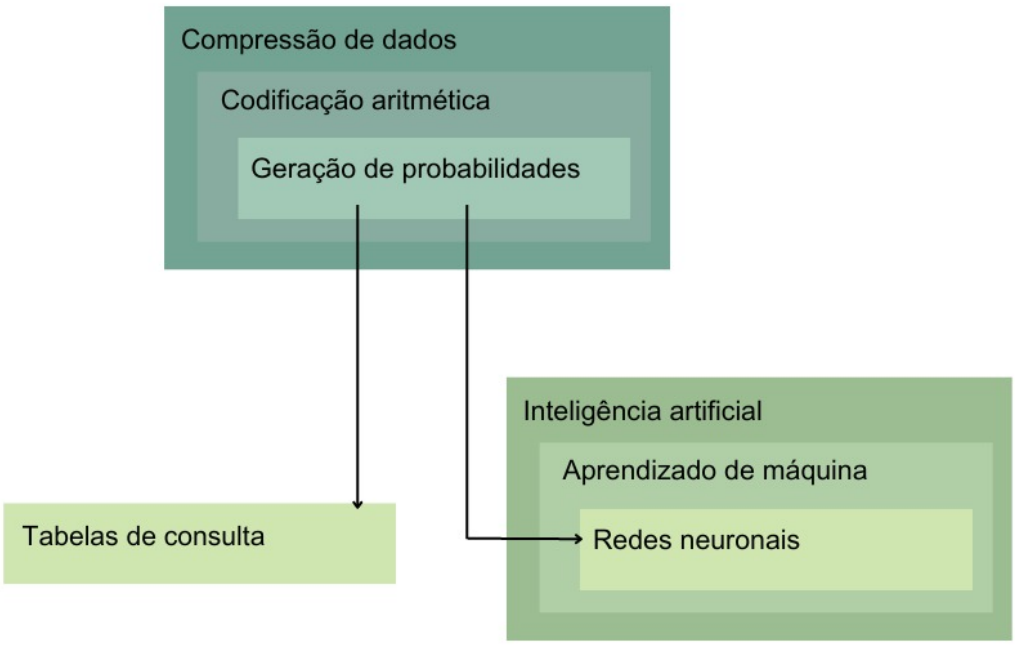

Figura 1.1: Resumo visual dos conceitos teóricos do problema.

1.1 Compressão de dados [1]

Compressão de dados é a redução de um conjunto de dados, comumente um arquivo, para transmissão ou armazenamento. Geralmente esse conjunto de dados é denominado mensagem. Na compressão de dados existem dois tipos de técnicas:

(i) **Com perda:** técnicas com perda identificam informação menos importante e a eliminam da mensagem.

(ii) **Sem perda:** técnicas sem perda eliminam redundâncias estatísticas, assim elas conseguem diminuir os bits da mensagens sem perder nenhuma informação.

1.1.1 Codificação Aritmética

No mundo da compressão de dados, uma das técnicas sem perda utilizadas é a codificação aritmética (CA). Na CA, é dividido um intervalo de [0,1) entre os possíveis símbolos do alfabeto da mensagem original, proporcionalmente à probabilidade de cada símbolo. Em seguida é lido o primeiro símbolo da mensagem e esse processo é repetido, usando o intervalo dado a esse símbolo no lugar do intervalo de [0,1). Dá-se continuidade a esse processo até o fim da mensagem quando é obtido um intervalo final bem pequeno. Então é encontrado o número binário, fracionário que contém o menor número de dígitos e está nesse intervalo, esse número é a mensagem comprimida. [7]

Um exemplo seria tentar comprimir a mensagem ACC, cujo alfabeto é A, B, C e as probabilidades são indicadas pela Tabela 1.1.

Tabela 1.1: Tabela de probabilidade de símbolos.

símbolo	**Probabilidade**
A	50%
B	30%
C	20%

O intervalo inicial é [0,1), dividindo proporcionalmente se obtém os intervalos indicados na Tabela 1.2

Tabela 1.2: Tabela de intervalos para cada símbolo na 1ª iteração.

símbolo	**Intervalo**
A	[0,0.5)
B	[0.5,0.8)
C	[0.8,1)

Como o primeiro símbolo é A, o novo intervalo é [0,0.5), então o intervalo é redividido e o resultado está indicado na Tabela 1.3

Tabela 1.3: Tabela de intervalos para cada símbolo na 2ª iteração.

símbolo	**Intervalo**
A	[0,0.25)
B	[0.25,0.4)
C	[0.4,0.5)

Como o segundo símbolo é C, o novo intervalo fica [0.4,0.5), então esse intervalo é dividido uma última vez gerando a Tabela 1.4

Tabela 1.4: Tabela de intervalos para cada símbolo na 3ª iteração.

símbolo	**Intervalo**
A	[0.4,0.45)
B	[0.45,0.48)
C	[0.48,0.5)

Como o último símbolo lido é C, o intervalo final é [0.48,0.5). Agora encontra-se o número binário, fracionário com o menor número de dígitos que está nesse intervalo. Esse número é: 0.011111, que pode ser representado como 011111, então ACC pode ser comprimido para 011111 usando codificação aritmética.

1.1.2 Contexto e imagens binarizadas

Na *CA*, para calcular qual a probabilidade de que o próximo símbolo a ser lido seja um símbolo específico, usam-se os n símbolos já lidos mais próximos dele. Isso é chamado de contexto, e o tamanho desse contexto vai ser referido daqui para frente como n.

Um dos tipos de dados que podem ser comprimidos usando-se a *CA*, são imagens binarizadas (imagens preto e branco, cujos pixels são todos representados por 0 ou 1). Nesse caso, o alfabeto tem apenas dois símbolos, 0 e 1, e o contexto é formado pelos n pixels já lidos mais próximos ao pixel a ser lido.

Nas Figuras 1.2 a 1.5 são ilustrados contextos de imagens com tamanhos 4, 10, 32 e 100 respectivamente.

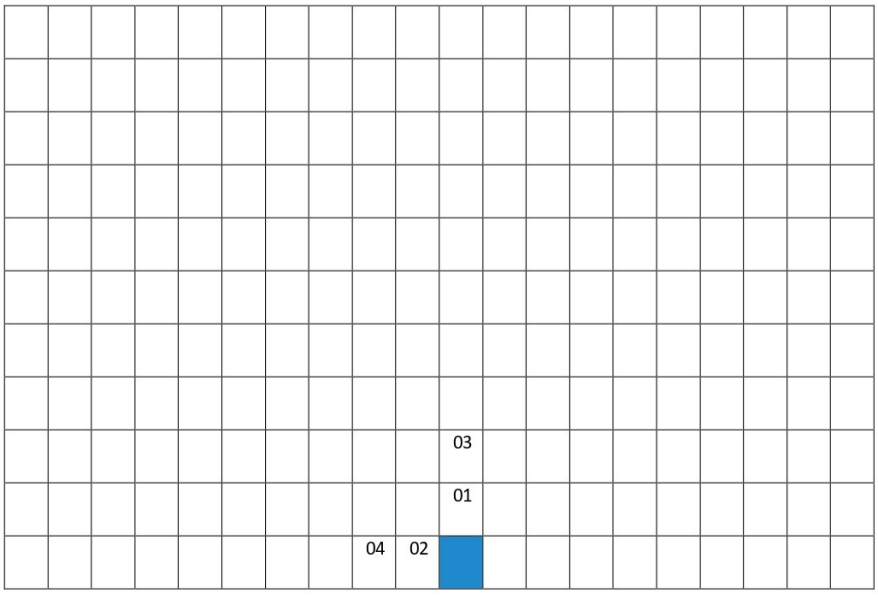

Figura 1.2: Exemplo de contexto do pixel colorido, com tamanho 4.

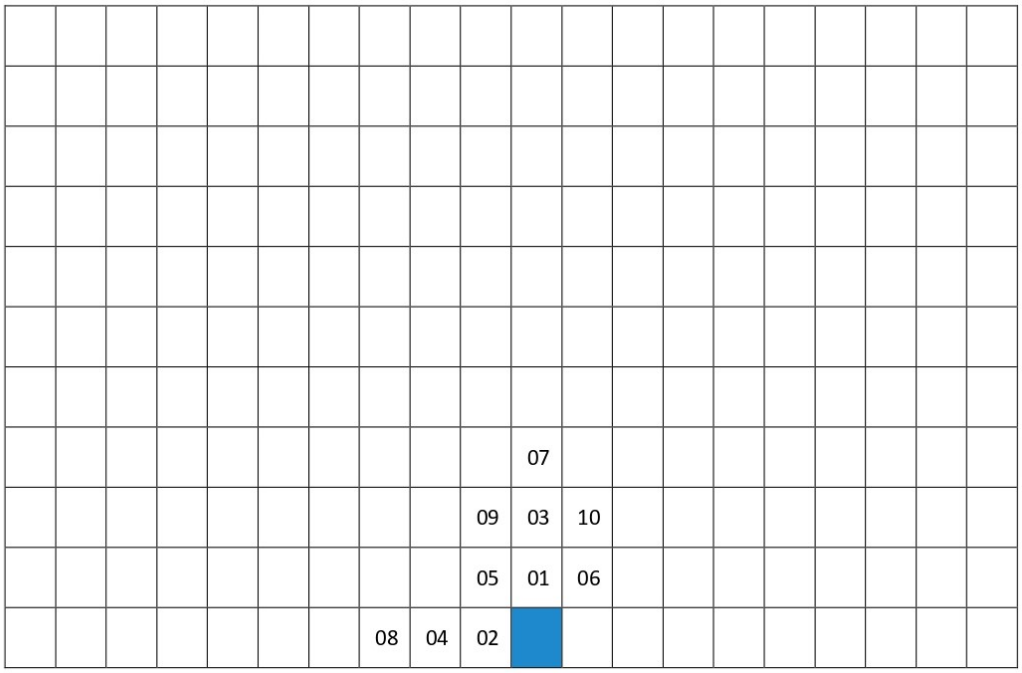

Figura 1.3: Exemplo de contexto do pixel colorido, com tamanho 10.

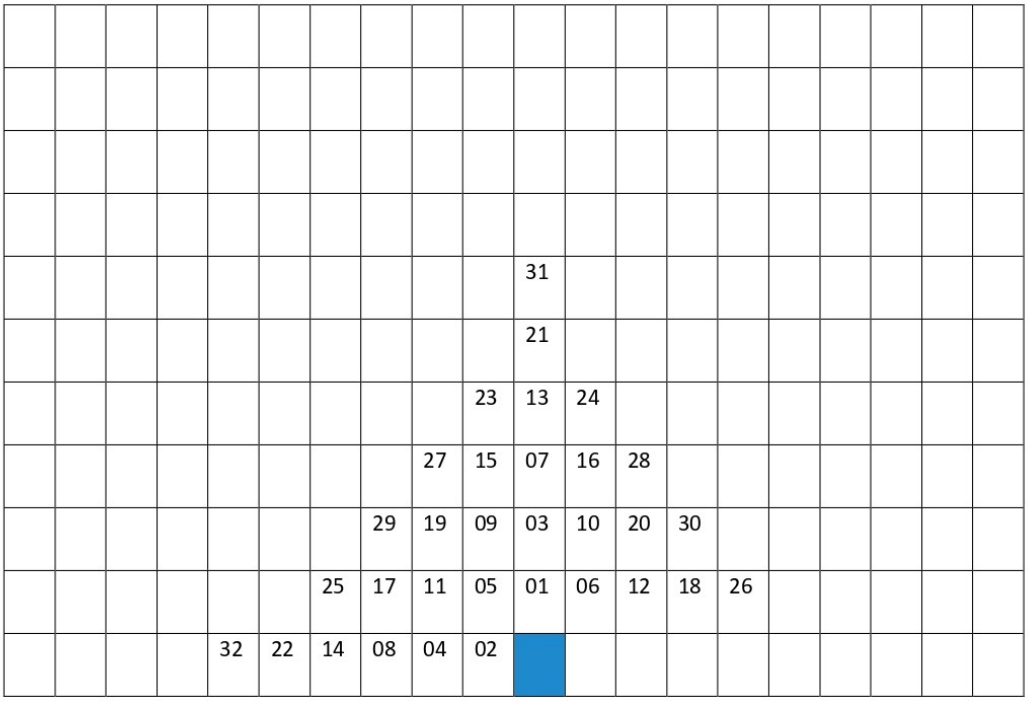

Figura 1.4: Exemplo de contexto do pixel colorido, com tamanho 32.

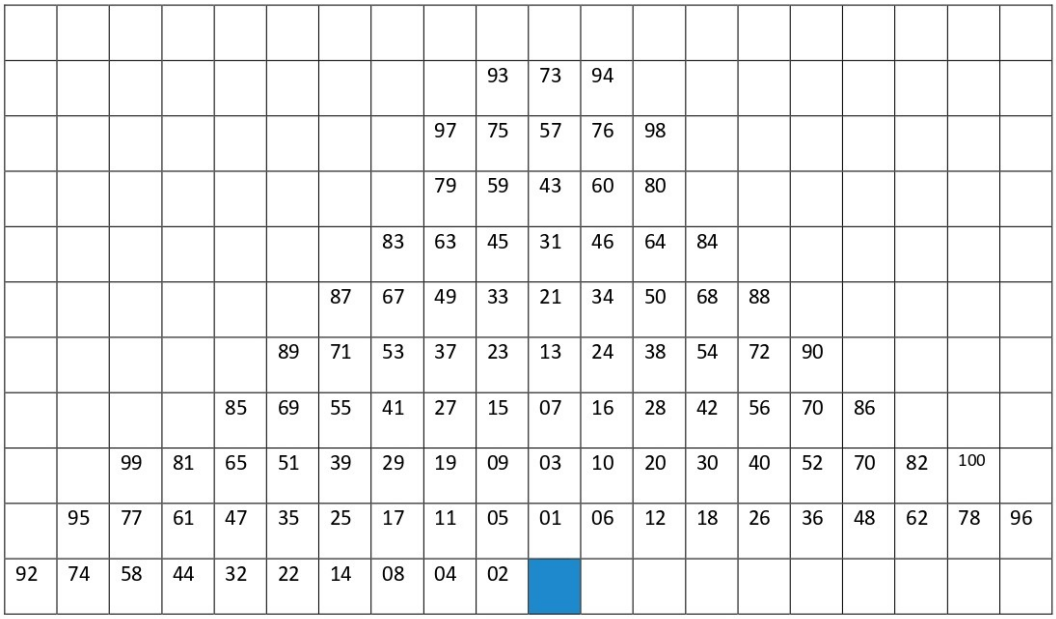

Figura 1.5: Exemplo de contexto do pixel colorido, com tamanho 100.

1.1.3 Tabelas de consulta [2]

A *CA* depende de boas estimativas de probabilidade para conseguir atingir bons níveis de compressão.

O método clássico para se obter as probabilidades de cada símbolo, baseado em seu contexto, são as tabelas de consulta, também chamadas de LUT's. Para construir essas tabelas contam-se os símbolos e os contextos em várias mensagens para calcular as probabilidades desejadas. Em seguida, guardam-se essas informações na tabela. Essas tabelas tem sempre 2^n linhas. Para n pequenos essa técnica é excelente e por isso é a usada quando se quer gerar probabilidades para *CA* com contextos pequenos. A Figura 1.6 ilustra uma dessas tabelas com n igual a 3.

Col	context	probability
0	000	13%
1	001	18%
2	010	18%
3	011	10%
4	100	3%
5	101	7%
6	110	17%
7	111	14%

Figura 1.6: Exemplo de tabela de consulta de $n = 3$, com probabilidades aleatórias.

O problema dessa técnica é que essa tabela cresce exponencialmente com n, logo, valores muito grandes de n são inviáveis. Mesmo para valores médios de n tem-se o problema de diluição do contexto. Esse problema é quando aparecem contextos na mensagem a ser codificada que não apareceram nas mensagens usadas para construir a tabela. Essa falta de garantia de boas estimativas para probabilidades especificas vai se agravando com o crescimento de n. A Figura 1.7 ilustra uma tabela de consulta com tamanho do contexto igual a 5. Quando comparada com a tabela da Figura 1.6, pode ser observado como é drástico o crescimento das mesmas.

Col	context	probability
0	00000	2%
1	00001	3%
2	00010	5%
3	00011	2%
4	00100	5%
5	00101	6%
6	00110	1%
7	00111	1%
8	01000	1%
9	01001	2%
10	01010	2%
11	01011	4%
12	01100	1%
13	01101	3%
14	01110	5%
15	01111	3%
16	10000	3%
17	10001	1%
18	10010	5%
19	10011	1%
20	10100	3%
21	10101	6%
22	10110	3%
23	10111	5%
24	11000	1%
25	11001	3%
26	11010	5%
27	11011	5%
28	11100	2%
29	11101	3%
30	11110	4%
31	11111	4%

Figura 1.7: Exemplo de tabela de consulta de $n = 5$, com probabilidades aleatórias.

As Figuras 1.2 a 1.5 acima serão usadas para explanar o desempenho das LUT's em relação ao tamanho do contexto. Na Figura 1.2 a tabela de consulta teria 16 entradas e seria uma perfeita alternativa para gerar as probabilidades desse pixel. Semelhantemente na Figura 1.3 a LUT teria 1024 linhas, seria bem maior, mas ainda seria uma alternativa excelente em termos de performance. Já na Figura 1.4 a tabela de consulta teria mais de 4 bilhões de linhas. Nessas condições, ainda seria possível usar esta nessas condições, mas a mesma gastaria muita memória e sofreria com uma forte diluição de contexto, o que na prática torna essa opção inviável em termos de performance. Por fim na Figura 1.5 a LUT precisaria de um número de linhas na ordem de 10^{30}. A construção de uma tabela dessa grandeza em memória seria inviável na maioria das máquinas de hoje.

Para solucionar essa limitação usa-se outra técnica que não necessita das probabilidades exatas de todos os símbolos para todos os contextos do tamanho n escolhido. Essa técnica consiste em treinar uma rede neuronal, para que essa consiga prever a probabilidade do próximo símbolo, dado seu contexto.

1.2 Inteligência artificial [3]

Inteligência artificial, no âmbito da ciência da computação, se refere a tentativa de fazer com que computadores consigam processar problemas de forma inteligente como humanos ou animais processam. Infelizmente não existe uma definição formal para o que isso significa, então o campo da inteligência artificial reúne uma série de técnicas que, de alguma forma, tentam imitar um aspecto da inteligência biológica.

1.2.1 Aprendizado de máquina

Aprendizado de máquina é uma área dentro da inteligência artificial onde, para resolver um problema, ao invés de se pensar em um algoritmo que consiga resolver esse problema e implementá-lo, cria-se um algoritmo que tenta aprender a solução do problema. Esse tipo de técnica pode encontrar soluções para problemas mesmo quando os programadores que implementaram o algoritmo não sabem como resolver esse problema, ou quando se entende pouco sobre o problema em si. Também é comum que técnicas de aprendizado de máquina sejam mais eficientes em determinado problema do que outros algoritmos, em determinados contextos.

1.2.2 Redes neuronais [4]

Redes neuronais são uma técnica do aprendizado de máquina, onde tenta-se aproximar uma função de saída desejada, desconhecida, a partir de pontos conhecidos da mesma.

Para isso essa técnica usa uma rede dividida em camadas, onde cada camada tem neurônios que se ligam a outros neurônios na camada seguinte. Existem três tipos de camada:

(i) a camada de entrada, que é sempre a primeira e recebe o vetor de entrada

(ii) a camada de saída, que é sempre a última e devolve o vetor de saída

(iii) as camadas ocultas, que são camadas intermediárias que processam a entrada para que essa gere a saída correta

Nesse tipo de rede os neurônios tem um valor interno chamado de bias. Quando os dados fluem na rede passando pelos neurônios, estes somam todos os sinais que recebem ao seu respectivo bias e multiplicam esse novo valor pelo peso associado a cada ligação com os neurônios da camada da frente, gerando assim um novo sinal para cada ligação [8]. Claro que, para que a saída esteja correta, o importante é que os pesos das arestas e os bias dos neurônios estejam corretos. Para encontrar os valores corretos, a rede passa por um treinamento. A forma geral de uma arquitetura de uma dessas redes está ilustrada na Figura 1.8

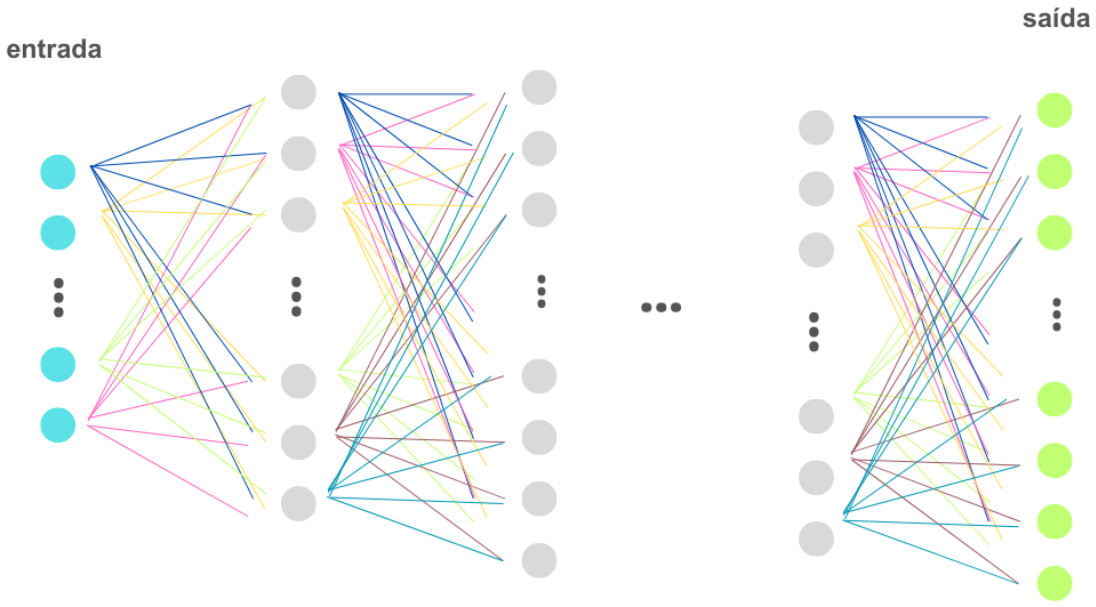

Figura 1.8: Exemplo de arquitetura de uma rede neuronal.

Nesse treinamento a rede é alimentada com diversas entradas para as quais se sabe a saída correta, então, a rede processa essas entradas e devolve uma saída. Essa saída é processada junto com a saída desejada para que se calcule uma perda, que é proporcional a diferença entre as saídas. Essa perda então é utilizada para modificar os pesos das ligações dos neurônios e suas biases, por meio de um algoritmo, denominado gradiente descendente

estocástico. Ao fim de várias iterações desse processo, a rede consegue aproximar a função desejada. Cada vez que esse processo usa todas as entradas disponíveis, esse conjunto de iterações é chamado de uma epoch.

1.2.3 Gradientes e mínimos locais [5]

O gradiente de uma função é uma função vetorial. As coordenadas do vetor gerado pelo gradiente são definidas como a derivada da função por cada variável. Por exemplo, se:

$$f(x, y) = x^2 + y^4 = 0$$

o gradiente de $f(x, y)$ será:

$$\nabla f(x, y) = (\frac{d(x^2 + y^4)}{dx}, \frac{d(x^2 + y^4)}{dy}) = (2x, 4y^3)$$

O gradiente sempre aponta para a direção do maior aumento de valor da função. Isso é ilustrado para uma função qualquer na Figura 1.9

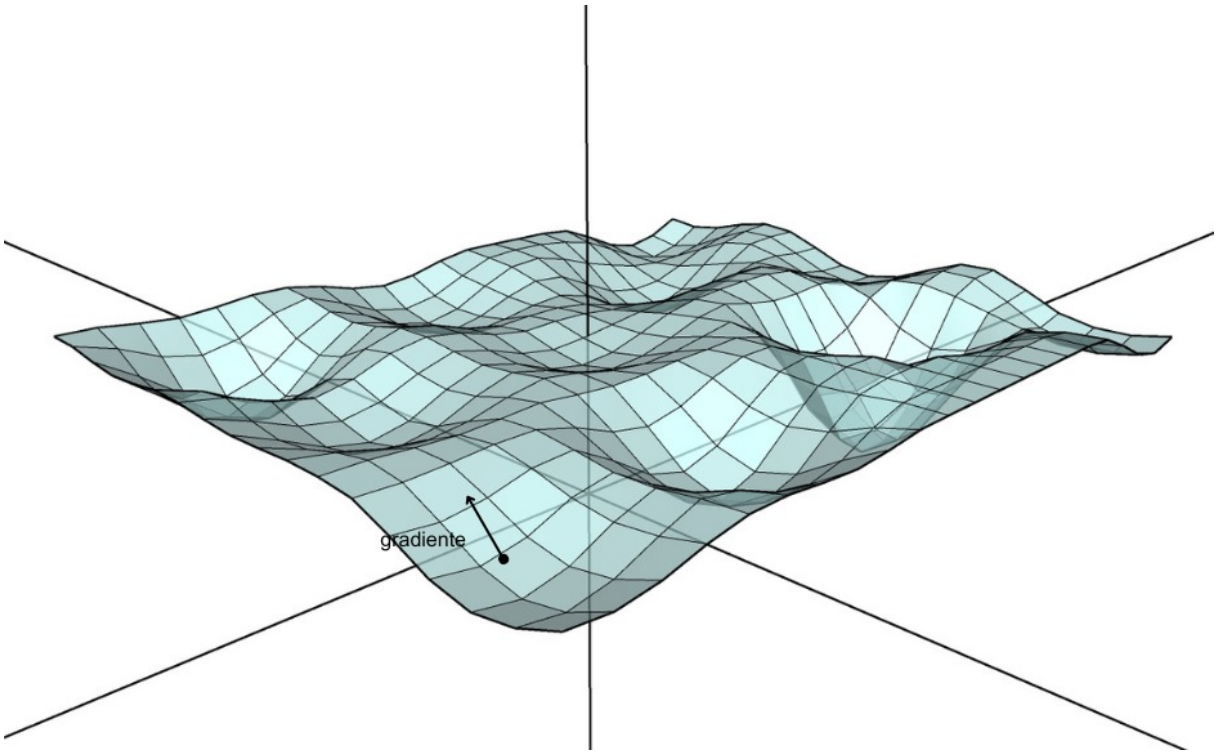

Figura 1.9: Gradiente de função 3D qualquer.

Graças a essa propriedade do gradiente, é possível inverté-lo para obter a direção do mínimo local. Dessa forma, com o gradiente, é possível encontrar um mínimo local de qualquer função.

1.2.4 Retropropagação de erro [4]

Retropropagação de erro é uma técnica matemática que se utiliza da regra da cadeia do cálculo para calcular de forma eficiente o gradiente de uma função em função das variáveis de um grafo. Usando o gradiente da saída do grafo e a regra da cadeia, é possível calcular o gradiente em termos das variáveis da última camada do grafo, e com o gradiente em função das variáveis de uma camada e a regra da cadeia, é sempre possível calcular o gradiente em relação às variáveis da camada anterior.

1.2.5 Gradiente descendente estocástico [6]

Gradiente descendente estocástico é um algoritmo que calcula uma aproximação do gradiente de uma função em relação a um conjunto de entradas, para iterativamente chegar mais perto de um mínimo local da função. Para isso é necessário que se defina um tamanho para o passo que as entradas da função devem tomar em direção ao mínimo em cada iteração, assim, o algoritmo pode tomar um desses passos na direção do gradiente em sentido contrário ao do mesmo, se aproximando do mínimo local. A escolha desse tamanho de passo não é trivial, e altera muito a eficiência do algoritmo. Isso se dá porque se esse passo é muito pequeno, o algoritmo precisa de muitas iterações para finalmente chegar no mínimo, porém um passo muito grande pode passar direto pelo mínimo, o que também pode fazer com que o algoritmo demore muitas iterações, ou até mesmo que o algoritmo fique preso, sempre passando pelo mínimo sem nunca chegar mais perto dele.

1.2.6 Retropropagação de erro e gradiente descendente estocástico aplicados em redes neuronais

Para treinar redes neuronais é muito comum que se use o algoritmo do gradiente descendente estocástico com o auxílio da retropropagação de erro, para adaptá-lo a estrutura de grafo das redes, assim é possível encontrar um mínimo local em funções de erro de redes neuronais. Nesse contexto o tamanho do passo do gradiente descendente estocástico, é chamado de taxa de aprendizado.

1.2.7 Complexidade

Há várias maneiras de se calcular uma complexidade para uma rede neuronal, a escolhida foi:

$Complexidade = (NE + 1) * N1 + (N1 + 1) * N2 + (N2 + 1) * NS$, onde:

$NE =$ neurônios na camada de entrada

$N1 =$ neurônios na 1^{a} camada oculta

$N2 =$ neurônios na 2^{a} camada oculta

$NS =$ neurônios na camada de saída

1.2.8 Redes neuronais aplicadas para *CA*

O problema de encontrar a probabilidade do próximo símbolo ser 1 ou 0, baseado no contexto dele, pode ser entendido como um problema de classificação binária de redes neuronais. Nesse caso o grau de certeza que a rede tem de que um contexto pertence a uma classe é entendido como a probabilidade de que o símbolo lido seja correspondente a essa classe. O contexto então seria a entrada e a rede teria n neurônios na camada de entrada para ler o contexto e um neurônio na camada de saída para receber a probabilidade. A arquitetura da rede está ilustrada na Figura 1.10

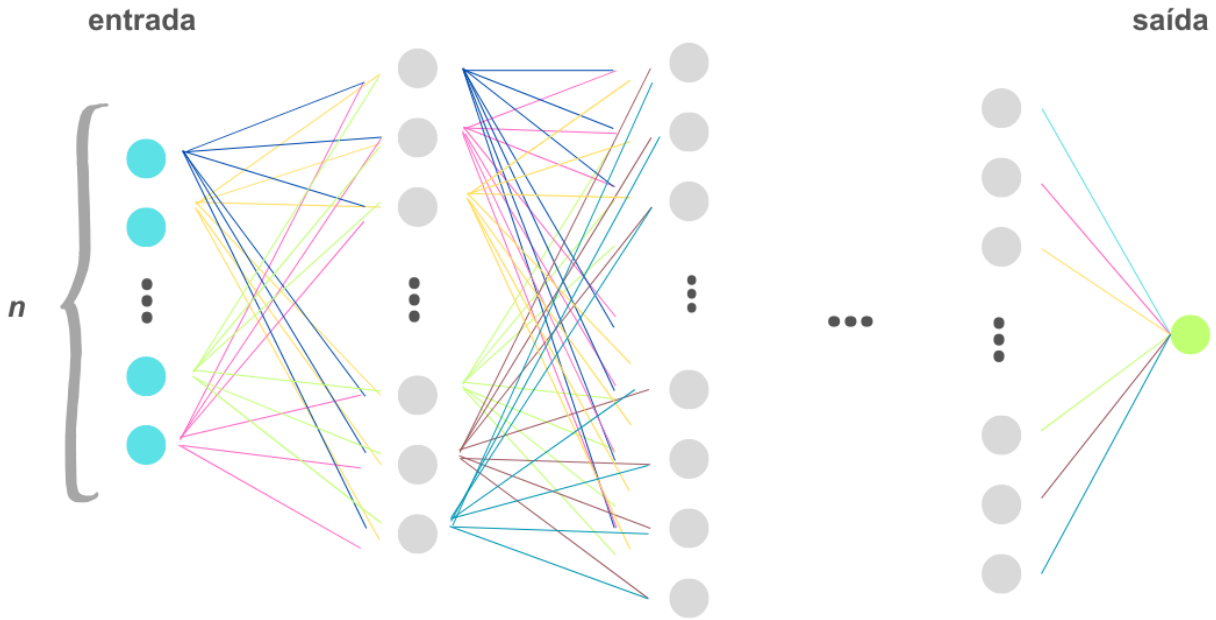

Figura 1.10: Arquitetura da rede.

Usando essa técnica, é viável gerar probabilidades com contextos muito grandes, comparados com o máximo das LUT's. As redes neuronais evitam o problema da diluição de contexto, pois encontram padrões nas entradas usadas nos treinamentos que podem ser

aplicados a entradas nunca antes vistas. Por esse motivo, sempre se tem uma estimativa baseada em todos os contextos treinados para qualquer entrada, seja esta nova ou velha, que seja avaliada. Revisitando nossos exemplos de contexto das Figuras 1.2 a 1.5, para o contexto na Figura 1.2, uma LUT teria um desempenho provavelmente melhor que o de uma rede neuronal. No contexto da Figura 1.3 a rede provavelmente já teria uma pequena vantagem competitiva em relação a LUT por causa da diluição de contexto. No caso da Figura 1.4 a rede teria um desempenho já bastante superior ao da tabela de consulta, tanto em termos de uso de memória, quanto em termos de performance, mais uma vez, devido à diluição de contexto. Mesmo na Figura 1.5 ainda seria viável treinar uma rede para um contexto desse tamanho, coisa que não é verdade para uma LUT, como foi visto na seção 1.1.3 (Tabelas de consulta). Essas afirmações foram extrapoladas dos resultados obtidos no paper Adaptive Context Modeling for Arithmetic Coding Using Perceptrons [2].

1.3 Objetivo

Na *CA* existe o problema de gerar boas estimativas para tamanhos de contexto grandes. Uma das soluções para esse problema é treinar modelos de redes neuronais para prever a probabilidade do próximo símbolo baseado em seu contexto.

Conhecendo esse problema foi escolhido fazer uma análise de diferentes modelos de redes neuronais, comparando o bits/amostragem como medida de qualidade dos resultados e complexidade como medida de custo, com o objetivo de descobrir quais são os modelos mais eficientes. Essa análise é relevante, pois a complexidade do modelo é diretamente relacionada ao tempo para fazer uma inferência e a quantidade de memória usada pelo modelo.

A eficiência de modelos de redes neuronais aplicadas a compressão de dados é especialmente importante, porque, diferente de outras aplicações de redes neuronais, não faz sentido rodar o modelo remotamente visto que uma das duas principais razões para compressão é acelerar a transmissão de dados via rede. Se for preciso mandar a mensagem sem compressão para usar o modelo, esse será inútil. Logo surge a necessidade de modelos leves que possam ser usados nos mais diversos aparelhos, com as mais diversas limitações de recursos, como computadores antigos, celulares ou até mesmo micro-controladores, entre outros.

Capítulo 2

Procedimento

Este capítulo descreve a confecção do projeto, baseado no artigo Adaptive Context Modeling for Arithmetic Coding Using Perceptrons [2].

2.1 Imagens

Para esse projeto foram usadas 20 imagens binarizadas, para treinamento e avaliação dos modelos. Um exemplo de imagem está representado na Figura 2.1

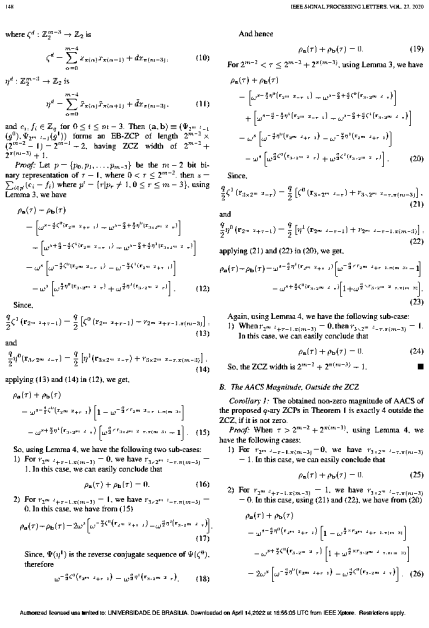

Figura 2.1: Exemplo de imagem usada para treinar/avaliar a rede.

Essas imagens foram as mesmas usadas no paper Adaptive Context Modeling for Arithmetic Coding Using Perceptrons [2].

2.2 Dependências importantes

Como linguagem foi escolhido o python, por ser uma das linguagens com mais suporte para desenvolvimento de redes neuronais hoje. Para leitura das imagens em formato png, foi usada a biblioteca pillow. Para o treinamento e criação dos modelos foi escolhido a biblioteca pytorch por ser uma das bibliotecas de python mais usadas e reconhecidas para essa tarefa. E para facilitar a manipulação de listas numéricas foi usado numpy.

2.3 Gerando os contextos

O primeiro desafio do projeto foi criar uma rotina que conseguisse gerar os contextos para cada pixel de uma imagem. Nesse projeto foram usadas duas rotinas diferentes para essa finalidade: uma externa, chamada de causal_context_2d [9], disponibilizada para ser usada nesse projeto, e uma desenvolvida durante o estudo, chamada de PrepareData.

A rotina PrepareData gera uma lista com as coordenadas do contexto do primeiro pixel, como ilustrado na Figura 2.2.

```
context_cordinate_list = [(-1,0)]

radius = -1
i = 1
while (i < n):
    if( (context_cordinate_list[-1][1] < 0)
        and (context_cordinate_list[-1][0] < 0) ):

        context_cordinate_list += [(context_cordinate_list[-1][0], -context_cordinate_list[-1][1])]

    elif( (context_cordinate_list[-1][0] < context_cordinate_list[-1][1])
        and (context_cordinate_list[-1][1] < 1) ):

        context_cordinate_list += [(context_cordinate_list[-1][1], context_cordinate_list[-1][0])]

    elif( (context_cordinate_list[-1][0] < -context_cordinate_list[-1][1])
        and (context_cordinate_list[-1][1] > 0) ):

        context_cordinate_list += [(-context_cordinate_list[-1][1], context_cordinate_list[-1][0])]

    elif( abs(context_cordinate_list[-1][1]) - abs(context_cordinate_list[-1][0]) > 1 ):
        context_cordinate_list += [(-abs(context_cordinate_list[-1][1]) + 1, context_cordinate_list[-1][0] - 1)]
    else:
        radius -= 1
        context_cordinate_list += [(radius, 0)]
    i += 1
```

Figura 2.2: Código responsável pela geração da lista de coordenadas do contexto do primeiro pixel.

A rotina percorre os pixels das imagens e para cada pixel, ela soma a coordenada do mesmo aos elementos da lista para gerar uma nova lista que possua as coordenadas do contexto desse pixel específico. Com essa última lista e os valores dos pixels já lidos anteriormente na imagem, a rotina consegue gerar o contexto desse pixel. Se um pixel

do contexto está fora da imagem, a rotina sempre coloca o seu valor como 1 (branco). O código para essa parte final da função está detalhado na Figura 2.3.

```
image_list = os.listdir(data_dir)

context_matrix = []
result_vector = []

images_size = 0
number_of_zeros = 0
for image_dir in image_list:
    image = np.array(img.open(data_dir + image_dir))

    height,width = image.shape
    print('image width: ', width)

    size = height * width

    i = 0
    j = 0

    for iterator in range(size):
        context_matrix += [[]]
        for point in context_cordinate_list:
            point = sum_coordinates(point, (i,j))
            if (point[1] > width) or (point[0] < 0) or (point[1] < 0):
                context_matrix[-1] += [1]
            else:
                context_matrix[-1] += [result_vector[images_size + (point[0]*width) + point[1]][0]]

        # print((i,j))
        #result_vector += [[image[i][j]]]
        if(image[i][j] == 255):
            result_vector += [[1]]
        elif(image[i][j] == 0):
            if(number_of_zeros < 10):
                print('coordinates: [', i, '][', j, ']')
            number_of_zeros += 1
            result_vector += [[0]]
        else:
            print("Error, image has value diferente from 0 and 255")
            exit(-1)

        j += 1
        if j >= width:
            i += 1
            j = 0

    images_size += width*height
```

Figura 2.3: Código responsável pela geração dos contextos de todos os pixels de todas as imagens.

2.4 Gerando modelos

Para gerar os modelos, foram escritas uma função para cada. Essas funções detalham a arquitetura de cada modelo, especificando a quantidade de neurônios em cada camada,

bem como aplicando um filtro entre as camadas ocultas que filtra valores negativos. Também detalham uma função sigmoide, mais especificamente uma função logística, no fim do modelo que serve para garantir que a saída da rede estará entre os valores 0 e 1. Em seguida, a função define se o aparelho do modelo será uma *CPU* ou *GPU* (todos os modelos usaram *GPU*, para acelerar o treinamento do modelo). É definida a função de perda, como uma binary cross entropy loss. O motivo da escolha dessa função é explicado na seção 2.8 (Análise dos modelos). O próximo passo é definir o otimizador como um gradiente descendente estocástico com a taxa de aprendizado necessária. Finalmente, move-se o modelo para o aparelho correto, salva-se os pesos atuais como melhores pesos, atribui-se 1.0 como valor da melhor perda, 0 como número de epochs treinadas, 32 como tamanho do contexto (n) e salva-se o modelo em um arquivo para treinamento posterior.

A Figura 2.4 contém um exemplo de uma dessas funções geradoras de contexto.

```
def CreateModel_N_32_640_320_LR_01(modelPath):
    model = nn.Sequential(
        nn.Linear(32, 640),
        nn.ReLU(),
        nn.Linear(640, 320),
        nn.ReLU(),
        nn.Linear(320, 1),
        nn.Sigmoid()
    )

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    loss_function = nn.BCELoss()

    optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

    best_model_wts = copy.deepcopy(model.state_dict())

    model.to(device)

    checkpoint = {
    'epoch': 0,
    'model': model,
    'optimizer': optimizer,
    'loss_function': loss_function,
    'device': device,
    'N': 32,
    'best_model_wts': best_model_wts,
    'best_loss': 1.0}

    torch.save(checkpoint, modelPath + "N32_640_320_LR01_model_no_train")
```

Figura 2.4: Uma das funções usadas para gerar um modelo.

2.5 Arquivos de modelo

Para que os modelos possam ser treinados é preciso salvar mais que só o modelo em si. Portanto sempre que se refere a salvar um modelo, ou ler um modelo de um arquivo, na verdade além do modelo, também são salvos: a função de perda, a melhor perda, os pesos que geraram a melhor perda, o valor de n, o aparelho que o modelo deve rodar (CPU ou GPU), o otimizador usado para a retropropagação de erro e o número de epochs que o modelo já treinou.

2.6 Função de treinamento

A Figura 2.5 é um fluxograma completo da função de treinamento.

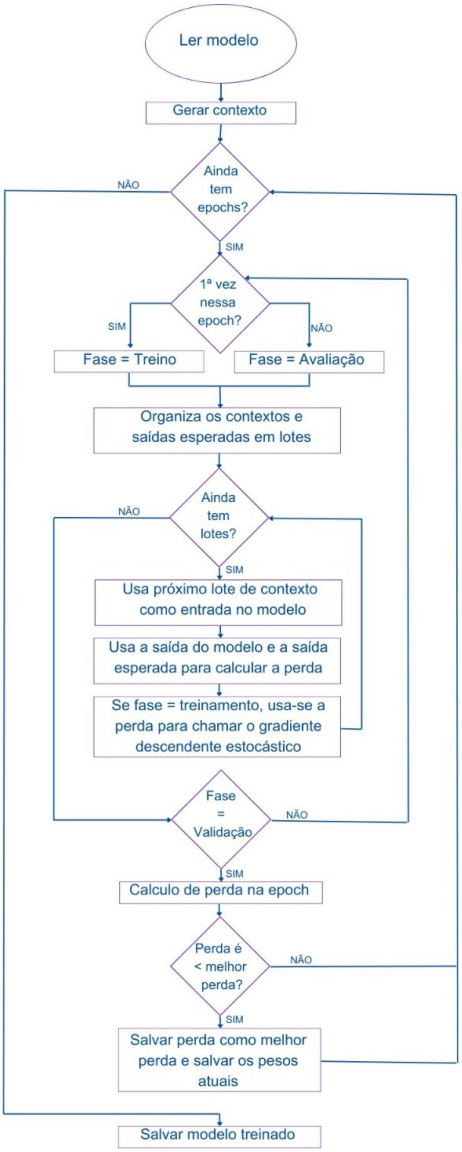

Figura 2.5: Fluxograma da função de treinamento.

A função de treinamento, desenvolvida para esse projeto, lê um arquivo com um modelo e treina esse arquivo por um número de epochs.

```
checkpoint = torch.load(modelDir)

if(restart_epoch_count):
    epoch = 0
else:
    epoch = checkpoint['epoch']

model = checkpoint['model']
optimizer = checkpoint['optimizer']
loss_function = checkpoint['loss_function']
device = checkpoint['device']
N = checkpoint['N']
best_model_wts = checkpoint['best_model_wts']
best_loss = checkpoint['best_loss']
```

Figura 2.6: Parte da função de treinamento responsável pela leitura do modelo.

Em cada epoch a função passa por uma fase de treinamento e uma de validação. Nessas fases, imagens são usadas para geração de contextos e vetores de resultado, usando a rotina de geração de contextos. As diferentes fases usam 10 imagens distintas para a geração dessas estruturas.

```
context_matrix_train, output_vector_train = PrepareData.PrepareData(N, "Data/SPL2020_train/")
len_matrix_train = len(np.array(context_matrix_train))
context_train = torch.tensor(np.array(context_matrix_train))
output_train = torch.tensor(np.array(output_vector_train))
dataset_train = torch.utils.data.TensorDataset(context_train, output_train)
#dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=batch_size, shuffle=shuffle)


context_matrix_val, output_vector_val = PrepareData.PrepareData(N, "Data/SPL2020_valid/")
len_matrix_val = len(np.array(context_matrix_val))
context_val = torch.tensor(np.array(context_matrix_val))
output_val = torch.tensor(np.array(output_vector_val))
dataset_val = torch.utils.data.TensorDataset(context_val, output_val)
#dataloader_val = torch.utils.data.DataLoader(dataset_val, batch_size=batch_size, shuffle=shuffle)

since = time.time()

while epoch < num_epochs:
    print(f'Epoch {epoch}/{num_epochs - 1}')

    for phase in ['train', 'val']:
        if phase == 'train':
            model.train()  # Set model to training mode
        else:
            model.eval()   # Set model to evaluate mode

        running_loss = 0.0


        if phase == 'train':
            len_matrix = len_matrix_train
            dataloader = torch.utils.data.DataLoader(dataset_train, batch_size=batch_size, shuffle=shuffle)
        else:
            len_matrix = len_matrix_val
            dataloader = torch.utils.data.DataLoader(dataset_val, batch_size=batch_size, shuffle=shuffle)
```

Figura 2.7: Parte da função de treinamento responsável pela diferenciação das fases, e construção dos contextos corretos baseados na fase.

A rotina usa esses contextos como entradas para o modelo e usa os vetores de resultado como saídas esperadas para o cálculo da perda, que foi feito usando uma função de cálculo de perda de entropia cruzada do próprio pytorch. Depois de calculada a perda, essa é usada para o gradiente descendente estocástico na fase de treinamento e para avaliação do modelo na fase de validação. A função lembra dos pesos que geraram a menor perda na fase de validação para que esses possam ser carregados para uso, após o treinamento.

```
batch_num = len(dataloader)

batch = 0
for inputs, labels in dataloader:
    inputs = inputs.float()
    labels = labels.float()
    inputs = inputs.to(device)
    labels = labels.to(device)

    # zero the parameter gradients
    optimizer.zero_grad()

    with torch.set_grad_enabled(phase == 'train'):
        outputs = model(inputs)
        _, preds = torch.max(outputs, 0)
        loss = loss_function(outputs, labels)

        # backward + optimize only if in training phase
        if phase == 'train':
            loss.backward()
            optimizer.step()

    # statistics
    running_loss += loss.item() * inputs.size(0)
    batch += 1

epoch_loss = running_loss / len_matrix

print(f'{phase} Loss: {epoch_loss:.4f}')

# deep copy the model
if phase == 'val' and epoch_loss < best_loss:
    best_loss = epoch_loss
    best_model_wts = copy.deepcopy(model.state_dict())
```

Figura 2.8: Parte da função de treinamento responsável por usar o modelo, gerar uma perda, usar o otimizador e lembrar da melhor perda.

No fim das epochs a rotina salva o modelo em um arquivo antes de terminar, para que o modelo possa ser usado, ou treinado novamente.

```
new_checkpoint = {
'epoch': epoch,
'model': model,
'optimizer': optimizer,
'loss_function': loss_function,
'device': device,
'N': N,
'best_model_wts': best_model_wts,
'best_loss': best_loss}

torch.save(new_checkpoint, modelPath + "N" + str(N) + "_shuffle_" + ("true" if shuffle else "false") +
    "_epochs" + str(epoch) + "_model_trained_date_" + str(time.time()))
```

Figura 2.9: Parte da função de treinamento responsável por salvar o modelo no fim do treinamento.

Cada fase das epochs também é dividida em lotes, cada lote tem 1024 entradas e saídas esperadas que são processadas ao mesmo tempo. Na fase de treinamento o gradiente descendente estocástico é chamado ao fim de cada lote. Isso pode ser observado nas Figuras 2.7 a 2.8. Após todos os lotes, a perda de cada fase é calculada como a soma das perdas de cada lote dividida pelo número de entradas da fase completa. Esse processamento em lotes é feito para acelerar o treinamento dos modelos, se aproveitando das capacidades de paralelização da *GPU* usada.

Todos os modelos foram treinados na mesma máquina e na mesma *GPU*. Essa decisão foi tomada para que esses pudessem ser todos treinados localmente em uma máquina privada, garantindo assim mais controle da máquina que rodava os modelos.

2.7 Treinamento de diversos modelos

Com a função de treinamento pronta, foram treinados vários modelos, tentando obter resultados melhores e entender quais configurações resultavam em modelos mais eficientes. Na tentativa de fazer isso foram gerados modelos que treinaram com e sem aleatorizar a ordem das entradas/saídas e com diferentes taxas de aprendizado. Todos os modelos sempre tiveram $n = 32$ e eram perceptrons multicamadas, o que quer dizer que todo neurônio na camada i está ligado a todos os neurônios na camada i+1, para qualquer camada i do modelo, com a exceção da última. Esse tipo de rede foi escolhida por ser muito flexível, e, portanto, poder ser usada em uma variedade maior de cenários.

Quase todos os modelos foram treinados até que a perda parasse de diminuir. Alguns demoraram cerca de 300 epochs para que isso ocorresse, outros mais de 1000. As únicas excessões são alguns modelos com taxa de aprendizado de 0.0001, esses foram todos treinados por mais de 1000 epochs e não mais que 1600. Apesar disso, os modelos não tinham boas perdas e, além disso, os modelos com essa taxa de aprendizado que foram treinados até pararem de melhorar a perda, ainda assim, tinham as perdas mais altas de todo o estudo. Por esses dois motivos, alguns desses treinamentos foram interrompidos prematuramente, pois não parecia fazer sentido gastar tanto tempo com esses modelos.

Na busca de melhores resultados para modelos grandes, foi criada uma nova rotina de treinamento que usava a rotina causal_context_2d [9], e nela foram treinados mais alguns modelos.

2.8 Análise dos modelos

Para analisar os modelos foi preciso calcular o bits/amostragem de cada modelo. Essa medida é proporcional a perda de entropia cruzada e pode ser obtida, nesse caso, dividindo essa perda pelo logaritmo natural de dois [2]. Em seguida, esses resultados foram plotados em um gráfico, para que se pudesse dar início à análise.

Capítulo 3

Resultados e Análise

Neste capítulo seram apresentados os resultados obtidos e as análises realizadas.

3.1 Apresentação dos resultados

O gráfico na Figura 3.1 mostra os resultados obtidos pelos diferentes modelos. Nesse gráfico, assim como nos próximos, a complexidade está em escala logarítmica para facilitar a visualização.

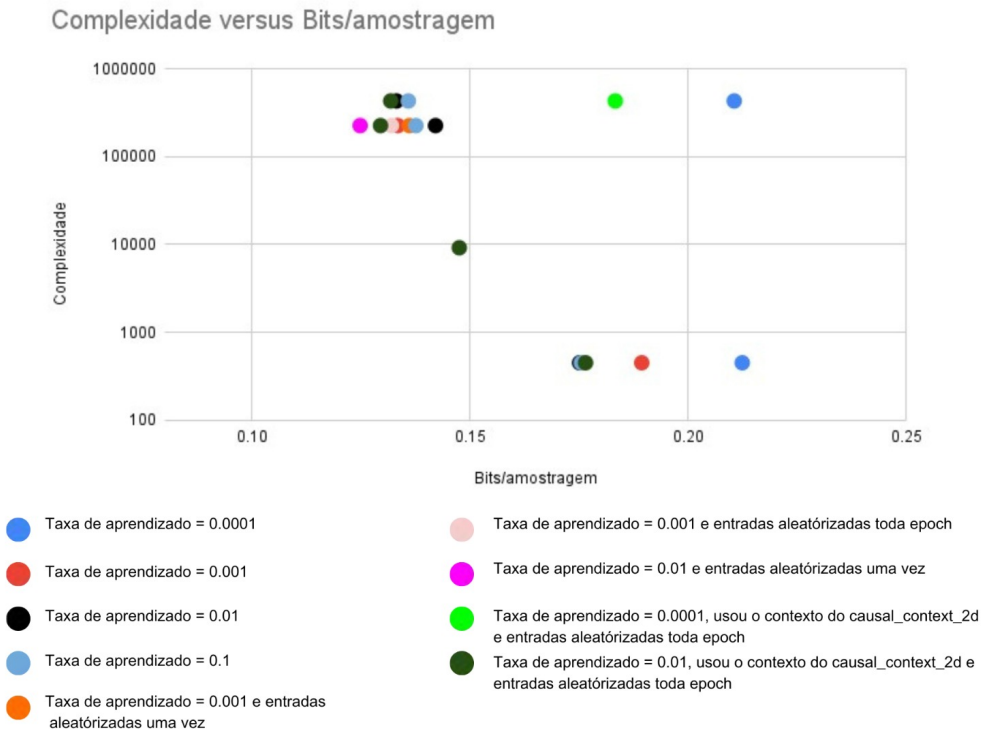

Figura 3.1: Gráfico dos resultados dos modelos.

Para cada complexidade usada há uma só arquitetura de modelos. As arquiteturas estão apresentadas na Tabela 3.1. A coluna arquitetura contém o número de neurônios por camada, em uma ordem que vai da camada de entrada para a de saída.

Tabela 3.1: Tabela de arquiteturas dos modelos por complexidade.

Complexidade	**Arquitetura**
432,001	32; 640; 640; 1
226,561	32; 640; 320; 1
9,201	32; 80; 80; 1
451	32; 10; 10; 1

3.2 Melhores modelos por complexidade

Fica claro olhando a Figura 3.1 , que os melhores modelos obtidos tinham uma taxa de aprendizagem de 0.01. Interessantemente parece que os modelos treinados usando a rotina de geração de contexto causal_context_2d [9] tem resultados parecidos com os modelos treinados com a rotina PrepareData, indicando que a última foi realmente bem construída no final das contas. Além disso, a aleatorização das entradas não parece ter um efeito claro nos resultados dos modelos, o que faz sentido, já que os modelos não tem memória. Colocando só os melhores modelos de cada complexidade no gráfico obtemos a Figura 3.2

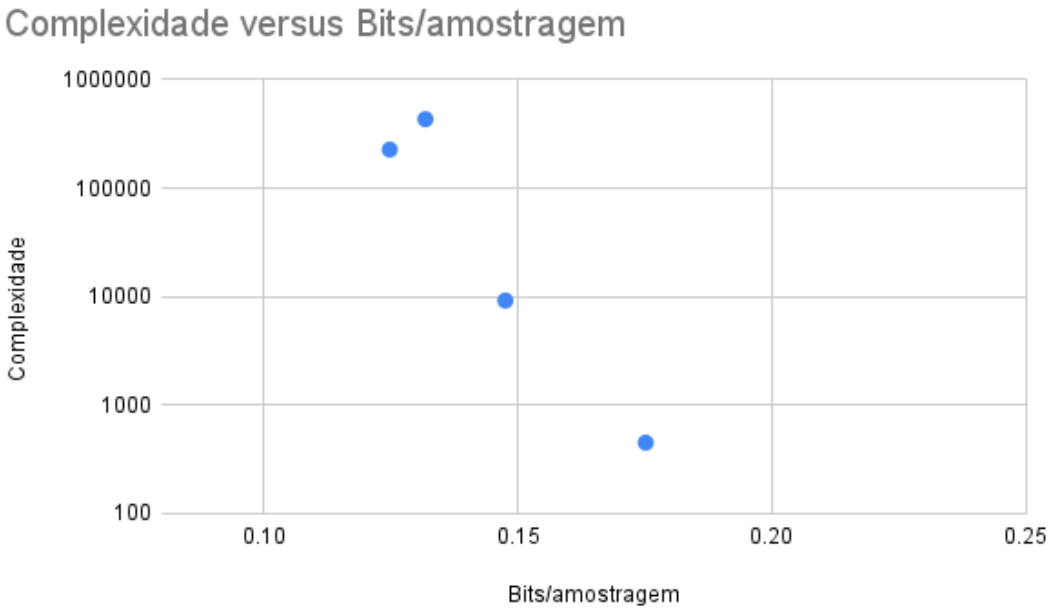

Figura 3.2: Gráfico dos resultados dos melhores modelos.

Nesse gráfico é possível aproximar uma curva que passe pelos pontos. Isso foi feito utilizando o excel e o resultado está apresentado na Figura 3.3

3.3 Curva dos melhores modelos

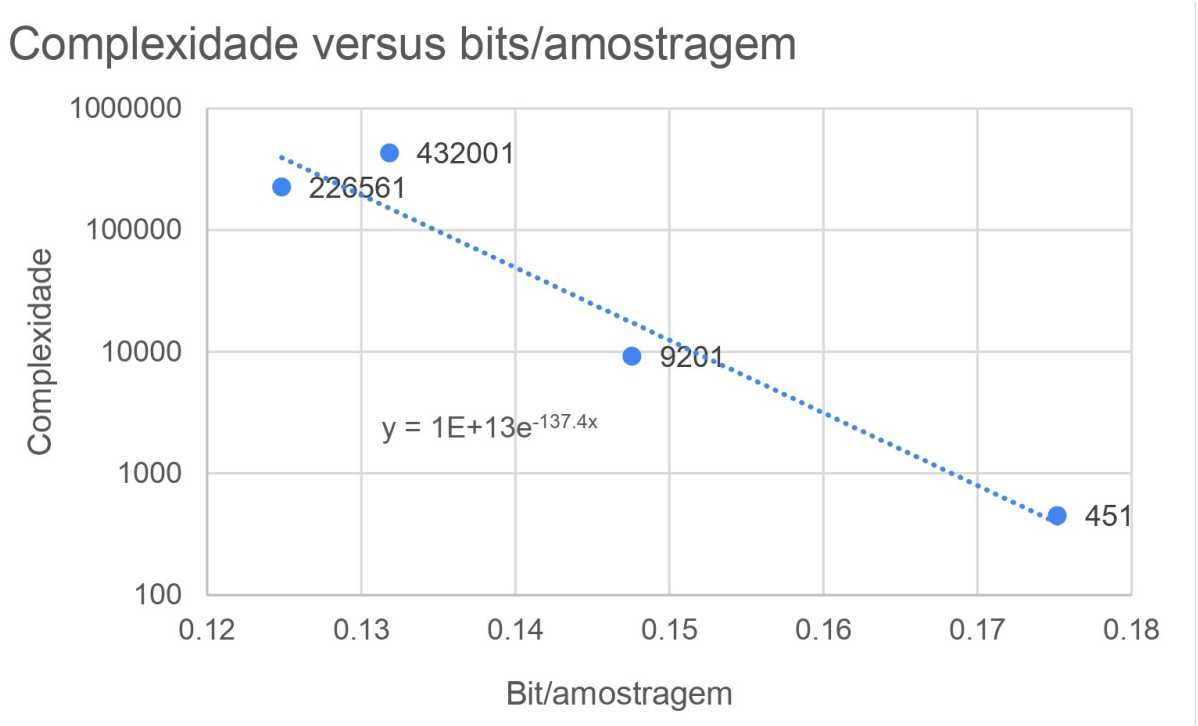

Figura 3.3: Gráfico da curva dos resultados.

Pode-se perceber que os resultados seguem aproximadamente uma curva exponencial, indicando que aumentar a complexidade dos modelos gera melhoras no bits/sample que decrescem muito rápido. Importante frisar que essa função é uma aproximação da realidade, nela uma complexidade de 10^{13} resulta em uma perda de 0. Claro que na realidade uma perda de 0 é impossível, visto que existem contextos iguais que geram saídas diferentes. É importante lembrar que o objetivo é conseguir a probabilidade de o próximo símbolo ser 1 (branco) para cada contexto e não realmente prever qual o próximo símbolo.

3.4 Quanto aos atributos de treinamento

O resultado mais conclusivo é que a taxa de aprendizado que gerou melhores resultados para todas as arquiteturas foi a de 0.01. Esse resultado condiz com a melhor taxa para as redes com contexto adaptável do paper Adaptive Context Modeling for Arithmetic Coding Using Perceptrons [2]. Também há um modelo com taxa de 0.1 e topologia 32; 10; 10; 1 que tem uma resultado muito bom, levando em conta a sua complexidade, apesar de existir um modelo com taxa 0.01 com essa mesma topologia que performou ainda melhor. Importante também ressaltar o melhor ponto do estudo, com arquitetura 32; 640; 320; 1. Esse ponto teve um resultado atipicamente bom. Olhando os modelos intermediários gerados durante o treinamento desse modelo, que precisou de 900 epochs para que sua perda estabilizasse, é possível ver que, mesmo com apenas 300 epochs, ele

já tinha os melhores resultados do estudo todo. Isso indica que os pesos iniciais desse modelo estavam por acaso posicionados em um ponto muito bom da função de perda do modelo, o que permitiu o encontro de um mínimo local menor que a média.

3.5 Modelos treinados com apenas 25 epochs

Posteriormente foram treinados mais 12 modelos, todos com topologias diferentes, para criar um novo gráfico com uma curva obtida a partir de mais pontos. Na Figura 3.4, encontra-se um gráfico com os resultados obtidos por esses mesmos modelos. Todos eles foram treinados com taxa de aprendizado = 0.01, entradas aleatorizadas toda epoch e usando a rotina causal_context_2d [9].

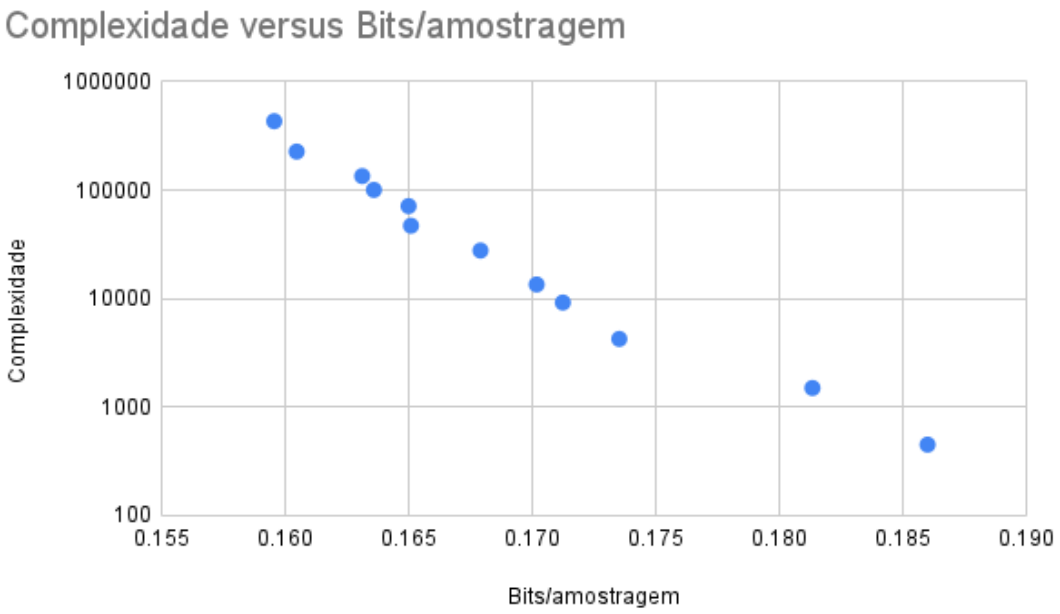

Figura 3.4: Gráfico dos resultados dos modelos treinados com 25 epochs.

É natural que modelos mais complexos precisem de mais treinamento para estabilizar suas perdas, logo faz sentido que nesse novo gráfico a melhora de bits/amostragem do pior modelo para o melhor caia em porcentagem, o que pode ser observado com a queda sendo de 25% nos modelos da Figura 3.2 e só 15% nos modelos da Figura 3.4. Novamente é possível traçar uma curva nesse novo gráfico para entender o comportamento desses modelos. Essa curva está ilustrada na Figura 3.5.

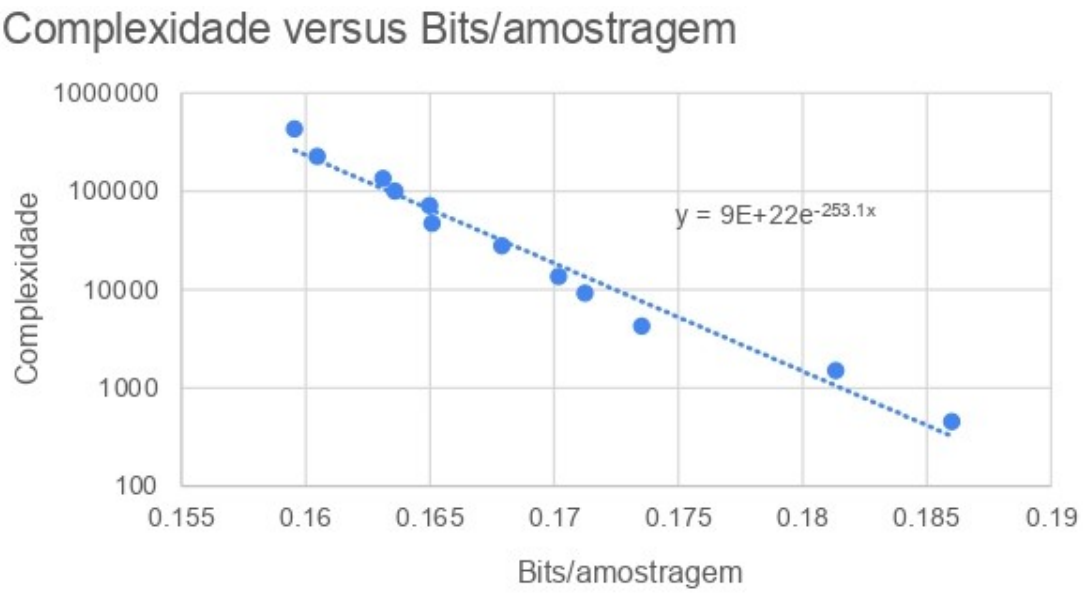

Figura 3.5: Gráfico dos resultados dos modelos treinados com 25 epochs.

Esse resultado parece confirmar a ideia de que a curva de complexidade e bits/amostragem é próxima de uma exponencial, apesar de a forma da exponencial ser diferente para as duas curvas.

Capítulo 4

Conclusão

Este capitulo fala das conclusões tiradas da confecção do projeto e da análise dos resultados e contêm sugestões de formas de expandir e melhorar os resultados aqui encontrados com outros estudos similares.

4.1 Conclusões a partir dos resultados

Através desse estudo, foi possível explorar o treinamento de diversos modelos de rede neuronal para uso em *CA* de imagens binarizadas. O objetivo de analisar vários destes foi atingido e o resultado de que a taxa de aprendizado igual a 0.01 gera melhores resultados (na média) foi bastante consistente nas arquiteturas estudadas, apesar do ruído aleatório natural dos modelos.

Além disso, quando considerado os resultados do paper Adaptive Context Modeling for Arithmetic Coding Using Perceptrons [2], esse resultado parece ser muito consistente nesse tipo de problema, já que no paper todos os modelos testados tinham a mesma arquitetura, mas tinham contextos de tamanhos diferentes. Por outro lado, no estudo atual, foi usado sempre o mesmo tamanho de contexto, mas foram testadas diversas arquiteturas de *MLP*'s com duas camadas escondidas. Nesse sentido, o estudo contribuiu para a conclusão de que essa é uma boa taxa de aprendizado.

Fora esse resultado principal, foi interessante ver que aleatorizar as entradas parece não afetar o desempenho dos modelos, provavelmente pela falta de memória dos mesmos. Mas talvez em redes neuronais recorrentes, cuja arquitetura é caracterizada por neurônios que mandam saídas para neurônios de camadas anteriores, isso não seja verdade.

4.2 Recomendações para possíveis expansões

Para enriquecer a literatura, no que se trata de redes neuronais aplicadas a Compressão Aritmética, seria útil a realização de uma série de outros estudos, tais como: estudos que repetissem o experimento em outras máquinas, que investigassem outros valores para o tamanho do contexto, que tentassem implementar uma taxa de aprendizado variável, que usassem outras imagens para alimentar a rede e que testassem arquiteturas diferentes.

Particularmente, seria interessante descobrir o quão consistente é o resultado da taxa ideal de aprendizado. Uma vez que existam muitos estudos desse tipo, seria possível criar um meta-estudo para tirar conclusões muito confiáveis sobre quais modelos são mais eficientes para esse tipo de problema.

Desta maneira, poderia surgir um guia para treinamento desse tipo de rede, o que economizaria muito tempo e esforço de tentativa e erro nas aplicações reais.

Por fim, também seria importante checar a validade desses resultados para outros problemas de classificação binária que usam redes neuronais, já que diversos problemas distintos podem ser resolvidos com implementações muito semelhantes dessa técnica. Por isso, é provável que configurações que resultam em modelos mais eficientes para determinados problemas, tenham efeitos similares em uma diversidade de outros contextos.

Referências

[1] Omar Adil Mahdi, Mazin Abed Mohammed e Ahmed Jasim Mohamed: *Implementing a novel approach an convert audio compression to text coding via hybrid technique*. IJCSI International Journal of Computer Science Issues, 9(03), 2012. vii, 2

[2] Lucas S. Lopes, Philip A. Chou e Ricardo L. de Queiroz: *Adaptive context modeling for arithmetic coding using perceptrons*. IEEE SIGNAL PROCESSING LETTERS, 29, 2022. vii, 6, 14, 15, 23, 26, 29

[3] McCarthy, John: *What is artificial intelligence?* 2007. vii, 9

[4] Nielsen, Michael: *Neural Networks and Deep Learning*. Determination Press, 2015. vii, 9, 12

[5] Bachman, David: *Advanced Calculus Demystified*. The McGraw-Hill Companies, 2007. vii, 11

[6] Ruder, Sebastian: *An overview of gradient descent optimization algorithms*. 2017. `https://doi.org/10.48550/arXiv.1609.04747`. vii, 12

[7] Li, Ze Nian e Mark S. Drew: *Fundamentals of Multimedia*. Pearson Education, Inc. Pearson Prentice Hall Pearson Education, Inc. Upper Saddle River, NJ 07458, 2004. 3

[8] Contributors, PyTorch: *Documentação do pytorch, parte sobre camadas lineares que especifica o comportamento dos neurônios nesse tipo de camada*. `https://pytorch.org/docs/stable/generated/torch.nn.Linear.html`. 10

[9] Lopes, Lucas S.: *Causal_context_2d, um módulo em python que cria contextos apartir de imagens binarizadas para uso nas redes neuronais*. 16, 23, 25, 27