Pedro Antônio Tibau Velozo

Super Resolução de Imagens e Vídeos com Deep Learning

RELATÓRIO DE PROJETO FINAL

DEPARTAMENTO DE CENTRO TÉCNICO CIENTÍFICO - CTC DEPARTAMENTO DE INFORMÁTICA

Programa de graduação em Engenharia de Computação

Rio de Janeiro
Novembro de 2023

Pontifícia Universidade Católica
do Rio de Janeiro

Pedro Antônio Tibau Velozo

Super Resolução de Imagens e Vídeos com Deep Learning

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador : Professor Alberto Barbosa Raposo
Co-orientador: Francisco Itamarati Secolo Ganacim

Rio de Janeiro
Novembro de 2023

Resumo

Velozo, Pedro; Raposo, Alberto; Ganacim, Francisco. **Super Resolução de Imagens e Vídeos com Deep Learning**. Rio de Janeiro, 2023. 89p. Projeto de Graduação – Departamento de CENTRO TÉCNICO CIENTÍFICO - CTC DEPARTAMENTO DE INFORMÁTICA, Pontifícia Universidade Católica do Rio de Janeiro.

A Super Resolução é um processo de transformação de imagens de vídeos de baixa resolução em alta resolução. Nesse contexto, este trabalho apresenta estudos sobre diferentes modelos baseados em *deep learning* que buscam solucionar o problema descrito. Ao longo do presente trabalho estes modelos selecionados são descritos e intuições por trás de seus funcionamentos são dados. Além disso são mostrados os pontos positivos e negativos dessas diferentes abordagens, apresentando os resultados em forma de gráficos, métricas, imagens e vídeos. Esses resultados são então analisados e comparados. Para garantir a compreensão e explicação com clareza dos pontos levantados, um estudo aprofundado sobre alguns conceitos importantes do âmbito de *deep learning* são apresentados e exemplificados.

Palavras-chave

Super Resolução, Processamento de Imagens e Vídeos, Deep Learning, SRCNN, DBPN, SRGAN, Redes Neurais, Redes Neurais Convolucionais.

Abstract

Velozo, Pedro; Raposo, Alberto (Advisor); Ganacim, Francisco (Co-Advisor). **Super Resolution of Images and Videos with Deep Learning**. Rio de Janeiro, 2023. 89p. Department of SCIENTIFIC TECHNICAL CENTER - CTC IT DEPARTMENT, Pontifical Catholic University of Rio de Janeiro.

Super Resolution is a process of transforming low-resolution video images into high-resolution. In this context, this work presents studies on different models based on deep learning that seek to solve the problem described. Throughout this work these selected models are described and intuitions behind their functioning are given. Furthermore, the positive and negative points of these different approaches are shown, presenting the results in the form of graphs, metrics, images and videos. These results are then analyzed and compared. To ensure clear understanding and explanation of the points raised, an in-depth study of some important concepts within the scope of deep learning are presented and exemplified.

Keywords

Super Resolution, Image and Video Processing, Deep Learning, SRCNN, DBPN, SRGAN, Neural Networks, Convolutional Neural Networks.

Sumário

Lista de Figuras

Lista de Tabelas

1
Introdução

A Super Resolução, também conhecida como SR, é o processo de transformação de imagens e vídeos de baixa resolução em alta resolução, como exemplificado na Figura 1.1.

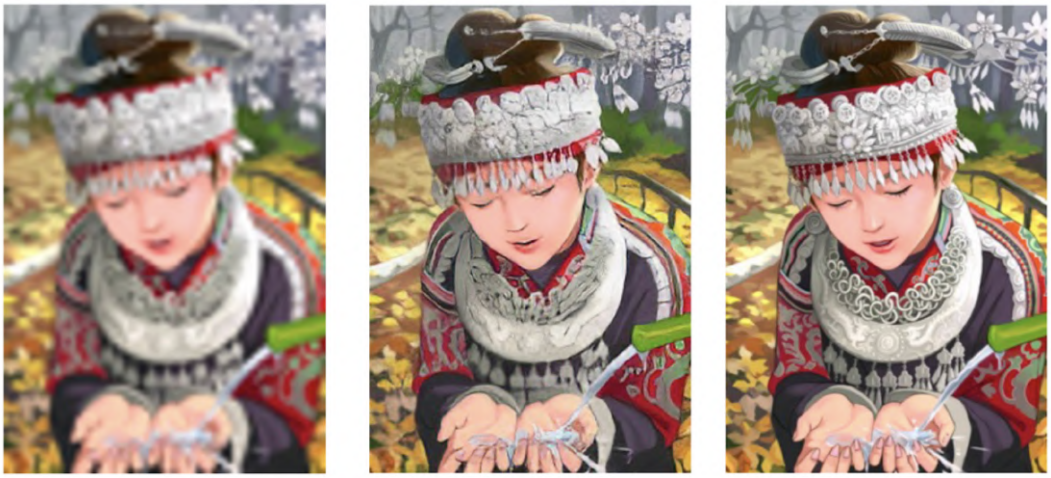

Figura 1.1: Da esquerda para a direita: imagem de baixa resolução, imagem de alta resolução gerada por um método de SR ($\hat{I}_y$), imagem de alta resolução original (I_y). Fonte: Adaptado de Ledig et al. (2017).

Esse é um tema que possui aplicações em diversas áreas de conhecimento. Um exemplo é a área médica, na qual, o uso da técnica se mostra importante para aumentar a qualidade e contraste de imagens em exames, o que pode levar a identificação de doenças em estágios iniciais e de mais fácil tratamento sem a necessidade de alterações de hardware significativas (ROBINSON et al., 2010). Outra área que se beneficia de avanços da SR é a de vigilância. Nessa área, no geral, para se conseguir uma imagem de alta resolução é necessária a instalação de câmeras de alta precisão, o que pode ser algo inviável pelo custo de implementação. Logo, a SR surge como uma alternativa mais acessível e eficiente para melhorar a qualidade das imagens capturadas por câmeras de vigilância de baixa resolução (ZHANG et al., 2010). Com isso, se torna viável uma melhor identificação de pessoas e objetos como placas de carro, auxiliando na prevenção e solução de crimes (VILLENA et al., 2018).

Em resumo, a SR apresenta um grande potencial para melhorar a qualidade das imagens e vídeos em diferentes domínios de aplicação, o que

justifica a importância de aprimorar as técnicas existentes e buscar soluções inovadoras para os desafios enfrentados na área.

O estudo aprofundado de diferentes técnicas de SR, e suas respectivas implementações se adequam ao escopo do Projeto Final por promoverem a prática de diversos conceitos vistos durante todo o curso, como a boa organização e estruturação do código desenvolvido, gerenciamento eficiente de recursos computacionais como os usos da CPU, GPU e memória RAM. Além disso, conceitos de *Machine Learning* como Redes Neurais, Redes Convolucionais, otimização de funções via descida do gradiente, entre outros são explorados extensivamente.

O projeto é desenvolvido em *Python* devido a grande quantidade de ferramentas voltadas para *Data Science* e *Machine Learning* existentes em seu ecossistema. Especificamente, as principais bibliotecas envolvidas no desenvolvimento são:

- ***Pytorch***: biblioteca voltada para a construção de modelos de *Machine Learning*, a partir dela há disponível diversas funcionalidades que facilitam a implementação de todas as etapas do processo de *Machine Learning*, que inclui a preparação dos dados dos *Datasets*, o treinamento e avaliação dos modelos gerados. Com o seu uso é possível criar uma rede neural, por exemplo, em poucas linhas de código;
- **Pandas**: biblioteca com o propósito de auxiliar na análise de dados em geral. No projeto, é usada para gerar gráficos e tabelas que são fundamentais para observar o comportamento dos modelos gerados;
- **Matplotlib**: biblioteca responsável pela criação de visualizações de dados em formatos de gráficos de linhas, barras, dispersão, histogramas e muito mais.

2
Situação Atual

Existem diversas abordagens para solucionar o problema de SR, incluindo técnicas determinísticas como as de *Interpolation-based Upsampling*, que preenchem as lacunas na imagem de baixa resolução por meio da interpolação de pixels vizinhos, gerando assim uma imagem de alta resolução. Dessa forma, nenhum detalhe ou característica é adicionada a imagem e então, embora a imagem resultante tenha dimensões maiores (maior número de pixels), a quantidade total de informação e a resolução real da imagem não são necessariamente aumentadas. Entre essas técnicas destacam-se *Nearest-neighbor Interpolation*, *Bilinear Interpolation* e *Bicubic Interpolation*. Embora esses métodos sejam fáceis de implementar, interpretar, de execução rápida e não exijam treinamento, eles tendem a gerar resultados em blocos de baixa qualidade (WANG; CHEN; HOI, 2020).

O foco deste projeto é estudar abordagens baseadas em *deep learning* para resolver o problema de Super Resolução. Diferentemente das técnicas determinísticas, essas abordagens exigem treinamento, mas têm apresentado resultados mais precisos e de maior qualidade, já que a partir delas é possível acrescentar detalhes que não estão presentes nas imagens originais. Existem diversos métodos que já foram estudados, que variam de complexidade, desde métodos baseados em *Convolutional Neural Networks* (CNN), como o SRCNN (LEDIG et al., 2017), até abordagens que envolvem *Generative Adversarial Nets* (GAN), como o SRGAN (LEDIG et al., 2017) ou ESRGAN (WANG et al., 2018).

Apesar das abordagens acima pertencerem ao ambiente de *deep learning*, suas implementações internas variam drasticamente. Essas diferenças costumam estar presentes em etapas chaves do processo, como em suas arquiteturas, ou seja, como as camadas de suas Redes Neurais ou Convolucionais estão organizadas e se relacionam entre si, ou como os dados de treinamento são tratados, e também que metodologias são usadas para avaliar e treinar os modelos.

3
Proposta e objetivo do trabalho

Este projeto tem como objetivo estudar, compreender e avaliar diversas metodologias baseadas em *deep learning* para solucionar o problema da Super Resolução, destacando as suas vantagens e limitações. Para alcançar esse objetivo, serão analisados diversos artigos publicados sobre o assunto e serão implementadas algumas das soluções propostas. As implementações serão realizadas dentro das limitações de recursos computacionais disponíveis para o projeto, pois é comum que as implementações exijam poder de processamento e tempo de execução significativos.

Serão estudados também assuntos importantes para se trabalhar no campo de *deep learning*. Entre esses está o estudo de conceitos fundamentais relacionados a redes neurais, incluindo os principais tipos de arquiteturas e técnicas de aprendizado utilizadas.

Além disso, estudaremos também como lidar com *datasets*, isto é, como coletar, organizar e tratar os dados que serão usados durante o treinamento e testes dos modelos criados.

Será estudado também como treinar modelos de *deep learning* para Super Resolução, utilizando os *datasets* criados anteriormente, explorando diferentes arquiteturas de redes neurais, técnicas de otimização e ajuste de hiperparâmetros.

Pesquisas e análises de artigos científicos e publicações relacionadas ao tema de Super Resolução e *deep learning* serão realizadas. O objetivo é identificar as tendências e avanços mais recentes na área, bem como as principais lacunas e desafios que ainda precisam ser abordados.

Após essas etapas, o objetivo será a comunicação dos resultados, ou seja, o foco será na aprendizagem de como escrever e apresentar os resultados obtidos no decorrer do projeto. Isso inclui a elaboração de relatórios e apresentações, que sintetizem as descobertas, metodologias e conclusões alcançadas.

4
Estudos teóricos preliminares

A primeira etapa no desenvolvimento do projeto, é a de pesquisa e estudo de diversos conceitos envolvidos no âmbito de *deep learning*, o que é essencial já que previamente a experiência e conhecimentos dos assuntos necessários pelo aluno eram baixos.

Neste capítulo serão abordados conceitos básicos de *deep learning* importantes para a compreensão das seguintes seções deste trabalho. Conceitos como redes neurais, funções de perda, descida do gradiente, entre outros, serão apresentados com base nas explicações dadas por Nielsen (2018).

Cada seção subsequente detalhará essas peças cruciais para a realização bem-sucedida do projeto, oferecendo uma compreensão aprofundada e estruturada de cada elemento fundamental.

4.1 Perceptrons

Um dos assuntos teóricos com a maior relevância para o desenvolvimento do projeto são as redes neurais, dessa forma, o primeiro tópico destacado nesse capítulo é o *perceptron*, também chamado de neurônio artificial, que é o elemento base de uma rede neural. Esse elemento consiste em um modelo matemático que recebe diversos números reais como *inputs* e produz apenas um *output*. Para cada *input*, denotados por $x_1, x_2, \ldots, x_n$, são associados pesos, ou *weights*, representados por $w_1, w_2, \ldots, w_n$ que simbolizam a relevância de específicos *inputs* para os *outputs* e um valor de *bias* indicado pelo símbolo b. A Figura 4.1 ilustra um diagrama de um perceptron.

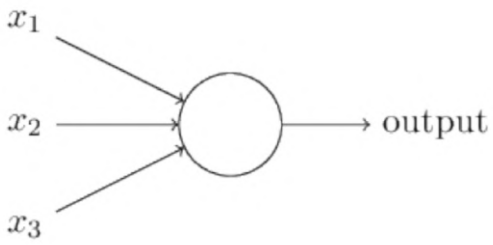

Figura 4.1: Diagrama conceitual de um perceptron.
Fonte: Adaptado de Nielsen (2018).

O output de um perceptron é calculado pela equação (4-1):

$$output = \sigma(wx + b) \tag{4-1}$$

Na qual σ representa a função de ativação, que é uma função na qual é aplicada a combinação linear $wx+b$, possibilitando que a rede neural aprenda a realizar o mapeamento da entrada para as saídas desejadas. Diversas funções podem ser utilizadas para cumprir esse papel, como por exemplo a função sigmoid ou ReLU (*rectified linear unit*), expressas nas equações (4-3) e (4-2) e Figuras 4.3 e 4.2, respectivamente (SHARMA, 2017).

$$\sigma(z) = ReLU = max(0, z) \tag{4-2}$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4-3}$$

A função sigmoid mapeia uma entrada para valores no intervalo entre 0 e 1, dessa forma ela costuma ser utilizada em tarefas de classificação binária, isto é, modelos que determinam a qual classe, de um total de duas, uma entrada

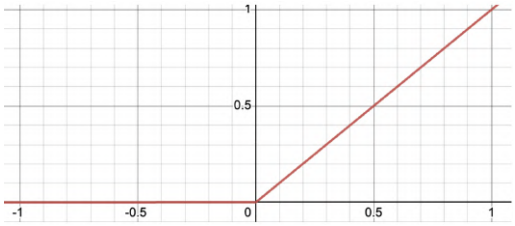

Figura 4.2: Curva ReLU

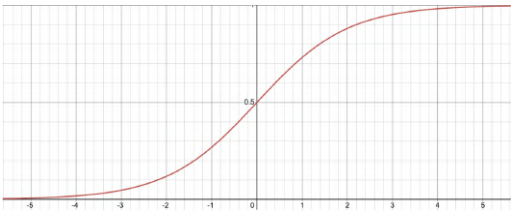

Figura 4.3: Curva sigmoid

pertence. O seu uso em projetos desse tipo é intuitivo, já que é possível mapear as saídas da sigmoid próximas a 0 para uma classe e próximas a 1 para a outra.

Já a função ReLU, é utilizada principalmente em redes neurais convolucionais (CNN), que são abordadas na seção 4.11 em mais detalhe. Essa função mapeia valores abaixo de zero para zero e valores acima de zero para eles mesmos. Uma de suas vantagens em relação a função sigmoid é o seu cálculo ser mais simples e consequentemente mais rápido de ser realizado, já que não é necessário calcular um valor exponencial, como no caso da sigmoid.

Além disso, ReLU possui algumas variações como a *Leaky* ReLU e PReLU. Essas variações surgiram como uma possível forma de melhorar a performance de modelos que utilizam a ReLU como função de ativação. Dessa maneira, suas operações são comparáveis às da ReLU, com algumas diferenças sutis. Ambas as funções seguem o mesmo formato da equação 4-4, porém a *Leaky* ReLU mantém o valor de a constante, enquanto na PReLU, esse valor pode variar durante o treinamento da rede (HE et al., 2015). Para exemplificar, a Figura 4.4 exibe o gráfico das funções para o caso de $a = 0.35$.

$$\sigma(z) = \begin{cases} z, & \text{se } z > 0 \\ az, & \text{se } z \leq 0 \end{cases} \tag{4-4}$$

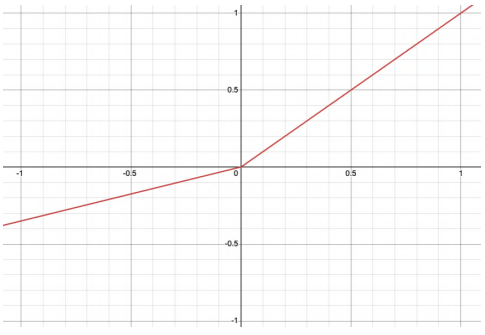

Figura 4.4: Curva *Leaky ReLU* com valor de $a = 0.35$.
Fonte: Autor.

Essas são apenas algumas funções de ativação comumente usadas em redes neurais, porém existem diversas outras, como por exemplo: *Softmax*, que é voltada para classificação entre múltiplas classes na qual é retornado um vetor

de probabilidades onde cada elemento representa a probabilidade de pertencer a uma das classes, *Tanh*, *Binary Step* e *Logistic*. Cada uma dessas possui suas respectivas vantagens e desvantagens e são utilizadas frequentemente em diversos projetos de *deep learning* (SHARMA, 2017).

4.2 Arquitetura de redes neurais

Com o conceito de neurônios artificiais definidos, é possível descrever a arquitetura de uma rede neural, que também pode ser chamada de *multilayer perceptrons* (MLP). As redes são formadas por camadas ou *layers* de neurônios que se conectam entre si, de tal forma que as entradas de uma camada são compostas pelas saídas da camada anterior. De forma geral, o modelo de uma rede neural pode ser descrito conforme a Figura 4.5, na qual os neurônios de entradas são denominados por *input layer*, os neurônios intermediários por *hidden layers* e os neurônios de saída por *output layer*.

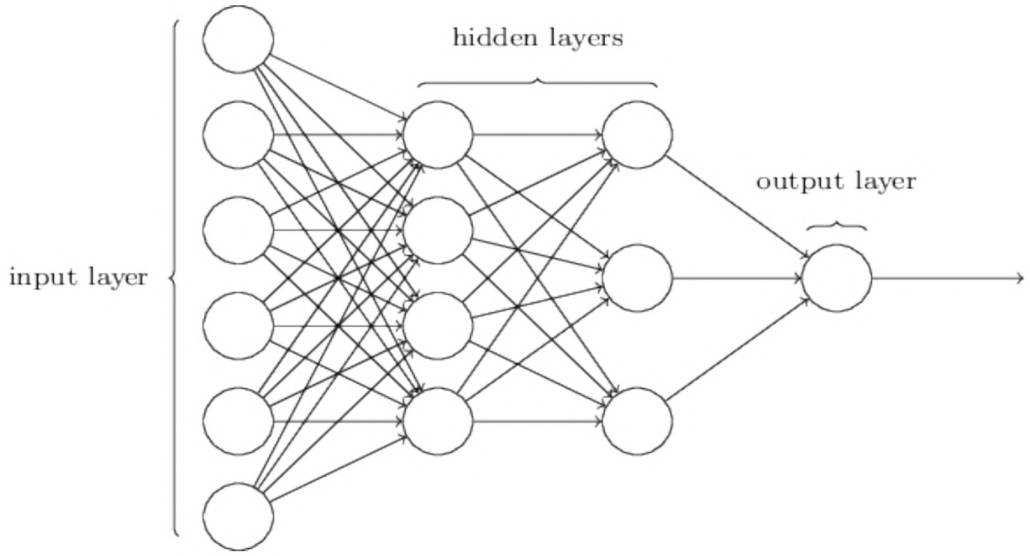

Figura 4.5: Representação das camadas de uma rede neural
Fonte: Adaptado de Nielsen (2018).

Na Figura 4.5 existem seis neurônios na *input layer*, um na *output layer* e há duas *hidden layers*, uma composta por quatro e outra por três perceptrons. Esses valores são apenas um exemplo, já que na realidade, a quantidade dessas camadas, e os seus tamanhos, isto é, quantos neurônios estão presentes em cada uma não é algo constante e pode variar de acordo com as especificações de cada projeto.

A definição dos tamanhos dos *input layers* e *output layers* tendem a ser intuitivos, a camada de entrada terá o número de neurônios iguais ao número de informações que se desejam passar para a rede neural, por exemplo, caso a rede esteja avaliando uma imagem de 64 por 64 pixels, como $64 * 64 = 4096$, a rede terá 4096 *input layers*. Já o tamanho das *output layers* é guiado pelo o que a rede está buscando solucionar, isto é, caso a rede seja voltada para classificações binárias por exemplo, então a saída pode ser apenas um perceptron que tem

sua saída aplicada na função sigmoid, como explicado na seção 4.1. A saída pode ter múltiplos neurônios também, como em uma rede na qual a sua saída é uma representação de uma imagem, dessa forma, a quantidade de neurônios na *output layer* serão iguais ao número de pixels esperado da imagem de saída.

Em relação ao número de *hidden layers*, já foi provado que, na teoria, uma MLP com apenas uma *hidden layer* é capaz de aproximar qualquer função, ou seja, com apenas uma camada teoricamente qualquer entrada pode ser mapeada para qualquer saída desejada (GOODFELLOW et al., 2014, p. 198).

Porém na prática, já foi observado que com apenas uma camada nem sempre os algoritmos de treinamento conseguem alcançar o mapeamento desejado. Assim, redes com múltiplas *hidden layers* costumam ser utilizadas para abordar essa limitação, possibilitando a aprendizagem de representações complexas e hierárquicas dos dados. Cada camada adicional permite a identificação de padrões e características em diferentes níveis de abstração, fornecendo uma maior capacidade de modelagem.

Na prática, não existe uma fórmula exata ou abordagem analítica para determinar o número ideal de camadas e neurônios em um modelo. É fundamental realizar experimentações e avaliar heurísticas para encontrar a configuração mais eficaz que se adeque ao modelo a ser construído. Essa etapa é uma combinação de tentativa e erro, testando diferentes arquiteturas e ajustando os parâmetros com base no desempenho do modelo (BROWNLEE, 2019a).

4.3 Aprendizado com a descida do gradiente

O processo de aprendizagem de uma rede neural consiste em encontrar os valores de *weights* e *biases* que minimizem o erro entre o valor de saída da rede e o valor conhecido esperado. Assim, ao se denotar $\hat{y}$ como a saída do modelo e y como a saída esperada, busca-se encontrar os parâmetros tais que $\hat{y} \approx y$.

Para calcular e quantificar o erro entre esses valores são utilizadas as funções de custo ou perda, ou do inglês *cost functions* ou *loss functions*. Por meio delas, é possível avaliar o quão longe do resultado ideal o modelo se apresenta. Existem diversas funções utilizadas como funções de perda, um exemplo comum é a *Mean Squared Error* (MSE) na qual é calculada a média dos quadrados das diferenças entre os valores preditos e os reais, expressa na equação 4-5.

$$MSE = C(x, w, b) = \frac{1}{n} \sum_{x=1}^{n} |y_x - \hat{y_x}|^2. \tag{4-5}$$

Em que:

- y é a saída esperada do modelo;
- $\hat{y}$ é a saída do modelo;
- n é o número de *inputs* utilizados durante o treinamento;
- w são os *weights* da rede neural;
- b são os *biases* da rede neural;
- x representa o número da entrada atual.

O ato de treinar uma rede neural consiste basicamente em encontrar o mínimo da função custo $C(w, b)$ em relação aos parâmetros w. Para minimizar as funções de custo e obter para elas valores próximos a zero, é utilizada a técnica da descida do gradiente.

O algoritmo da descida do gradiente consiste em se calcular o gradiente da função de custo em relação aos parâmetros da rede $\nabla_{(w,b)}C$ repetidamente, ou seja, o vetor que denota a direção de maior mudança da função, e atualizar os valores dos *weights* e *biases* para que o custo diminua. Dessa forma, após um certo número de iterações é esperado encontrar o mínimo da função. A partir deste ponto, para evitar excessos de notação, refere-se ao gradiente $\nabla_{(w,b)}C$ apenas como ∇C.

Em termos matemáticos, para simplificar a representação será denotado um vetor $v = (w, b)$, sendo w os *weights* e b o *bias*. O método pode ser descrito de acordo com a equação 4-6, na qual C é uma função de m variáveis $v_1, \ldots, v_m$,

e ∇C é o gradiente de C, ou seja, é o vetor composto por todas as derivadas parciais de C em relação a $v_1, \ldots, v_m$. Com ∇C definido, as mudança nas variáveis $v = (v_1, \ldots, v_m)^T$ é descrita na equação 4-7 pelo produto de ∇C com $-\eta$, no qual η é a taxa de aprendizagem, ou *learning rate*, que é um pequeno valor positivo que define o tamanho da mudança de valores entre as iterações do método. Para o funcionamento ideal do algoritmo, a taxa de aprendizado η deve ser pequena o suficiente para convergir para o mínimo de C, e ao mesmo tempo não ser excessivamente pequena para evitar que as mudanças em v sejam minúsculas e que o algoritmo demore mais que o necessário. No entanto, é vital evitar taxas de aprendizado excessivamente altas, pois podem acentuar erros, levando a oscilações indesejáveis e prejudicando a convergência para o mínimo de C. Assim, é essencial encontrar um equilíbrio adequado para a taxa de aprendizado, evitando tanto valores muito pequenos quanto muito grandes.

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T \tag{4-6}$$

$$\Delta v = -\eta \nabla C \tag{4-7}$$

$$v \to v' = v - \eta \nabla C \tag{4-8}$$

A equação (4-7) pode ser reescrita em termos dos *weights* e *biases* como na equação (4-9) e (4-10) (NIELSEN, 2018).

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \tag{4-9}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \tag{4-10}$$

Com essas definições estabelecidas, é crucial destacar que o gradiente ∇C precisa ser calculado de maneira a representar a média de todos os gradientes obtidos para cada dado de entrada durante o treinamento. Isso implica que $-\eta \nabla C$ indica a direção do mínimo do custo para todas as entradas, não apenas para uma em específico. Esse comportamento é formalizado na equação 4-11.

$$\nabla C = \frac{1}{n} \sum_{i=1}^{n} \nabla C(x_i) \tag{4-11}$$

Na qual:

– n é o número de *inputs* utilizados durante o treinamento;

– x representa o número da entrada atual.

O funcionamento do algoritmo pode ser visualizado ao considerar uma função de custo C que recebe apenas duas variáveis, v_1, v_2. Dessa forma é possível representar o método como na Figura 4.6, isto é, como um objeto descendo na direção oposta ao gradiente (representados pela bola e vetor verdes, respectivamente) até o mínimo da função.

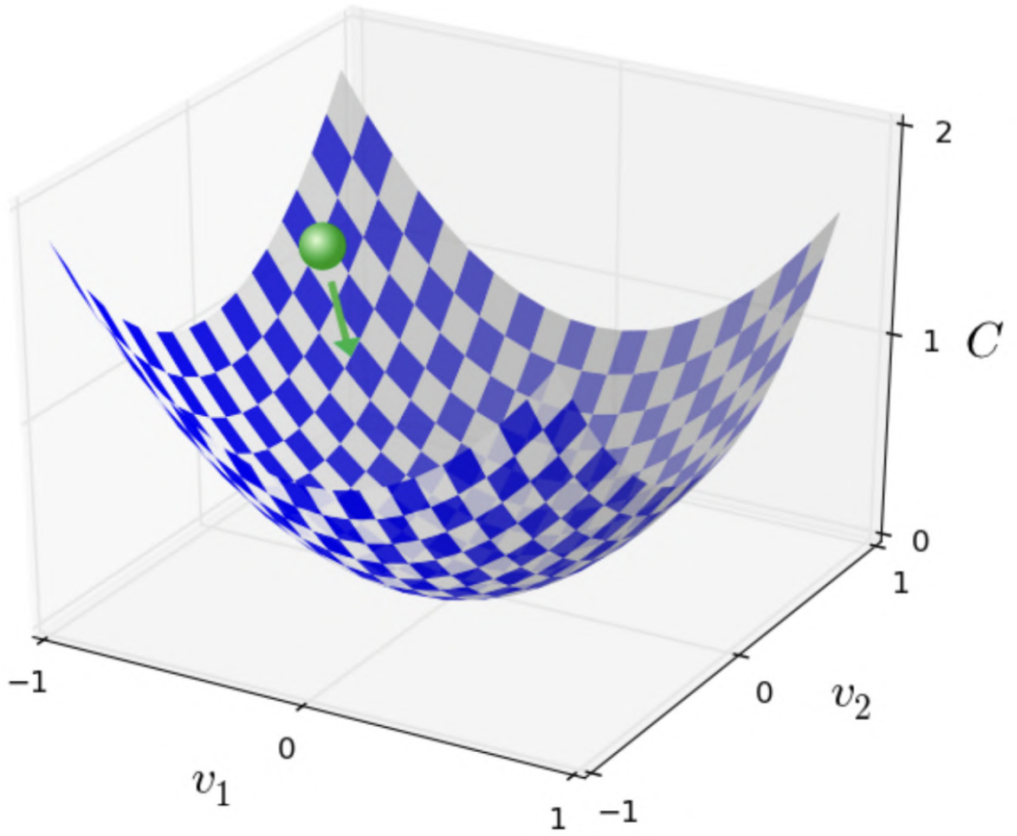

Figura 4.6: Representação visual do funcionamento da descida do gradiente
Fonte: Adaptado de Nielsen (2018).

Uma observação importante é que os mínimos da função de custo encontrados pelo SGD podem, muitas vezes, ser mínimos locais. Para se encontrar os mínimos globais com maior segurança é necessário o uso de técnicas mais complexas, como o uso de "momento", que de forma simples, introduz uma componente de velocidade adicional nos passos do algoritmo. Isso permite ao algoritmo superar mínimos locais devido à aceleração acumulada (AG, 2019). Além disso, também existem técnicas que consistem em se alterar o valor do *learning rate* durante o treinamento. A intuição por trás disso é que, inicialmente, ao usar valores mais altos, a rede pode evitar mínimos locais, e à medida que se aproxima do mínimo global, a taxa de aprendizado é reduzida (AREMU, 2023).

4.4 Descida do gradiente estocástico

Um problema que existe na descida do gradiente é a quantidade de processamento necessário para calcular o gradiente ∇C em cada iteração do algoritmo. Com o propósito de reduzir esse número e, consequentemente acelerar o processo de aprendizagem, foi introduzido o método da descida do gradiente estocástico, ou do inglês *stochastic gradient descent* (SGD). A técnica consiste em calcular uma aproximação de ∇C a partir de um pequeno número m de inputs de treinamento aleatórios, chamados de *mini-batches*, como mostra a equação (4-12) (NIELSEN, 2018).

$$\frac{1}{m}\sum_{j=1}^{m}\nabla C_{X_j} \approx \frac{1}{n}\sum_{x}\nabla C_x = \nabla C \tag{4-12}$$

No qual, o segundo somatório é sobre todos os dados de treinamento.

Para relacionar explicitamente o uso da técnica com o processo de aprendizagem de redes neurais, as equações (4-9) e (4-10) são reescritas como as equações (4-13) e (4-14), nais quais w_k e b_l representam os pesos e biases da rede.

$$w_k \to w'_k = w_k - \eta\frac{1}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial w_k} \tag{4-13}$$

$$b_l \to b'_l = b_l - \eta\frac{1}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial b_l} \tag{4-14}$$

Essas operações são feitas por todos os exemplos de treinamento X_j do *mini-batch* atual. Após isso, outro *mini-batch* é selecionado e o comportamento se repete até que todos os dados de treinamento tenham sido esgotados. Esse processo é chamado de *epoch* e, para concluir o treinamento, diversas epochs podem ser processadas.

O SGD é significativamente mais rápido do que a abordagem original da descida do gradiente. Para exemplificar, em um treinamento com $n = 60.000$ entradas, e tamanho de batch $m = 10$, o aumento de velocidade para estimar o gradiente é de 6.000 vezes. Porém, é importante ressaltar que ao trabalhar com o SGD estão utilizados aproximações do gradiente, de tal forma que quanto menor o tamanho do *mini-batch* escolhido, apesar do ganho de eficiência computacional, maior será a variância observada e pior o resultado. Portanto, é crucial identificar tamanhos ideais para os *mini-batches* que equilibrem eficiência computacional e a variância do resultado.

4.5 O cálculo do gradiente e *backpropagation*

No item 4.3 é descrito como as redes neurais aprendem por meio da descida do gradiente e para isso, é necessário calcular o gradiente da função de custo. Na prática, uma rede neural pode possuir milhões de parâmetros, assim é utilizado o algoritmo de *backpropagation* para fazer o cálculo do gradiente de maneira rápida e eficiente.

A intuição por trás do algoritmo consiste em um conceito chamado de retropropagação, na qual o erro calculado pela função de custo é propagado de volta pelas camadas da rede, ajustando os pesos dos neurônios de acordo com sua contribuição para o erro total. A ideia central é que, durante a retropropagação, o erro é atribuído de forma proporcional à contribuição de cada neurônio para o erro total, ou seja, *weights* e *biases* que possuem uma influência maior sobre a saída da rede sofrem alterações maiores (SANDERSON, 2017).

Para demonstrar a metodologia do *backpropagation* é considerado uma rede neural com L camadas, cada uma com apenas um neurônio, e uma função de custo $C_0 = (\hat{y} - y)^2$, na qual y é a saída calculada pelo modelo e y a saída esperada. A ativação dos neurônios é denotada por $a^{(l)}$, de tal forma que l representa a camada a qual o neurônio pertence. Dessa forma, a camada de saída é chamada de $a^{(L)}$ e a camada anterior de $a^{(L-1)}$. As ativações são compostas pela equação 4-15, sendo σ a função de ativação e w e b sendo os *weights* e *biases* associados ao neurônio escolhido.

$$a^{(l)} = \sigma(w^{(l)}a^{(l-1)} + b^{(l)}) \tag{4-15}$$

Para simplificar a notação das ativações, a combinação linear que é aplicada na função de ativação é denominada por $z^{(l)}$, como demonstra a equação 4-16. Assim, a ativação é denotada como na equação 4-17.

$$z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)} \tag{4-16}$$

$$a^{(l)} = \sigma z^{(l)} \tag{4-17}$$

Para o algoritmo da descida do gradiente é necessário calcular o gradiente ∇C que é composto por todos os *weights* e *biases*, como descrito na seção 4.4 e na equação 4-18.

$$\nabla C = \left(\frac{\partial C}{\partial w^{(1)}}, \frac{\partial C}{\partial b^{(1)}}, \ldots, \frac{\partial C}{\partial w^{(L)}}, \frac{\partial C}{\partial b^{(L)}} \right)^T \tag{4-18}$$

Para calcular os gradientes necessários $\frac{\partial C}{\partial w^{(L)}}$ e $\frac{\partial C}{\partial b^{(L)}}$ referentes a última camada L por exemplo, as equações 4-19, 4-20, são formadas a partir da regra da cadeia e calculam as derivadas parciais para uma entrada k (NIELSEN, 2018).

$$\frac{\partial C_k}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_k}{\partial a^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y) \tag{4-19}$$

$$\frac{\partial C_k}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_k}{\partial a^{(L)}} = \sigma'(z^{(L)}) 2(a^{(L)} - y) \tag{4-20}$$

Para encontrar o valor total correto dos componentes do gradiente é necessário calcular a média de todos as n entradas existentes, como é explicado na seção 4.4 e demonstrado na equações 4-21 e 4-22.

$$\frac{\partial C_k}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}} \tag{4-21}$$

$$\frac{\partial C_k}{\partial b^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial b^{(L)}} \tag{4-22}$$

A ideia por trás da retropropagação origina-se do cálculo da derivada da função de custo em relação à ativação da camada anterior $(l-1)$. Conforme demonstrado na Equação 4-23, esse valor é uma função dos *weights* da camada l, ou seja, $w^{(l)}$. Portanto, para calcular o gradiente da função de custo em relação a $w^{(l)}$ e $b^{(l)}$, a retropropagação é utilizada para determinar como o erro é distribuído de volta à camada anterior, permitindo o ajuste dos pesos e dos vieses de forma a minimizar o erro global do modelo. Esse processo é repetido para cada camada da rede, propagando o erro desde a camada de saída até a camada de entrada.

$$\frac{\partial C_k}{\partial a^{(l-1)}} = \frac{\partial z^{(l)}}{\partial a^{(l-1)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C_k}{\partial a^{(l)}} = w^{(l)} \sigma'(z^{(l)}) 2(a^{(l)} - y) \tag{4-23}$$

O raciocínio explorado neste capítulo é baseado em uma rede MLP simples, com apenas um neurônio por camada, porém a intuição e metodologia são análogos ao caso de redes mais complexas.

Além disso, é importante destacar que para a execução deste trabalho,

não é necessário o desenvolvimento explicito em código do algoritmo descrito, já que a biblioteca *Pytorch* utilizada, mencionada no capítulo 1, já possui uma implementação pronta.

4.6 Exemplo de funcionamento de uma MLP

Para facilitar a compreensão e reforçar os conceitos abordados, é apresentado nesse capítulo um exemplo simples de como uma MLP é utilizada para classificar dígitos escritos a mão. O objetivo da rede é receber como entrada uma imagem de um digito, que é representada como uma matriz de dimensão 28×28 com valores de zero a um, representando a intensidade dos pixels em cada posição da imagem, sendo os zeros pixels brancos e um pixels pretos.

Dessa forma, o exemplo consiste na rede neural ilustrada na Figura 4.7, composta por uma camada de entrada com 728 *perceptrons*, descritos na seção 4.1, sendo que cada um desses recebe um dos valores de pixels da entrada, já que $728 = 28 \times 28$. Apenas uma *hidden layer* é utilizada, e nesse caso contém 15 neurônios escolhidos de forma arbitrária, devido as explicações dadas no capítulo 4.2. E por fim, a rede possui um *output layer* com dez neurônios, cada um representando os números de 0 a 9, de modo que a camada passa por uma função de ativação *softmax*, descrita na seção 4.1, isto é, o neurônio que corresponde ao número escolhido pela rede tenha o valor mais alto em relação a todos os outros neurônios.

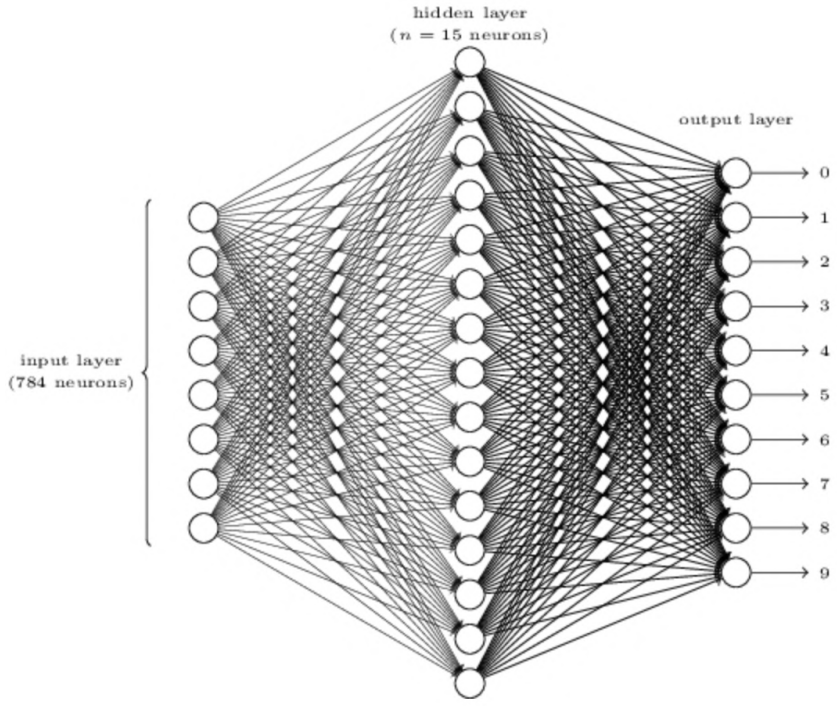

Figura 4.7: Rede MLP utilzada como exemplo
Fonte: Nielsen (2018)

Antes da rede ser treinada, os valores de *weights* e *biases* existentes para cada conexão entre os neurônios tendem a ser aleatórios, esse conceito é

explicado com mais detalhes na seção 4.8, e não conseguem fazer o mapeamento desejado. Dessa forma, é esperado que ao alimentar a rede com uma imagem de qualquer número, a saída será classificada de forma errada. Para exemplificar, a Figura 4.8 representa um digito cinco, pertencente ao *dateset* MNIST, escrito a mão que é a entrada fornecida a rede, e as equações 4-25 e 4-24 mostram a saída correta esperada e um exemplo errado que pode ser gerado pela rede inicialmente respectivamente. A saída errada é calculada utilizando os princípios abordados na seção 4.1, isto é, a saída de cada neurônio é o produto de seus neurônios antecessores com os seus respectivos *weights* e *bieases*, e então esse valor passa por uma função de ativação como a ReLU ou sigmoid.

Figura 4.8: Exemplo de um possível digito cinco que pode ser a entrada da rede MLP.
Fonte: Nielsen (2018)

$$saida\_errada = [0.1, 0.1, 0.1, 0.1, 0.1, 0.2, 0.1, 0.1, 0.1, 0.1] \tag{4-24}$$

$$saida\_correta = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0] \tag{4-25}$$

O processo de aprendizagem da rede consiste basicamente em atualizar os valores de *weights* e *biases*, por meio da técnica da descida do gradiente descrita na seção 4.3, de tal forma que os dígitos de entrada da rede sejam classificados corretamente. Como já mencionado, o que é feito na prática é buscar os valores desses parâmetros que minimizem a função de custo.

Nesse exemplo, é considerado como a função de custo o MSE, descrito também na seção 4.3. Ao aplicar essa função nos vetores *saida_errada* e *saida_correta* se encontra o erro obtido para os valores dos *weights*, *biases* e entrada correntes e esse valor é calculado na equação 4-26.

$$\begin{aligned} MSE = {} & (0.1-0)^2 + (0.1-0)^2 + (0.1-0)^2 + (0.1-0)^2 + (0.2-1)^2 \\ & + (0.1-0)^2 + (0.1-0)^2 + (0.1-0)^2 + (0.1-0)^2 + (0.1-0)^2 \\ & \Rightarrow MSE = 0.73 \end{aligned} \tag{4-26}$$

Para cada exemplo de entrada fornecido à rede, como o dígito cinco ilustrado na Figura 4.8, o MSE é calculado para medir a discrepância entre as saídas erradas e as saídas corretas. No exemplo acima, o MSE é calculado como 0.73, o que indica que as previsões da rede estão consideravelmente distantes das saídas desejadas.

O processo de treinamento da rede neural requer a avaliação do MSE para todos os exemplos de treinamento disponíveis. Uma vez que o MSE tenha sido calculado para todos os exemplos, a técnica de *backpropagation*, descrita na seção 4.5, é aplicada para calcular os gradientes das funções de custo em relação aos *weights* e *biases* da rede.

Com os gradientes em mãos, a descida do gradiente é usada para ajustar os *weights* e *biases* da rede de forma a minimizar o MSE. Isso implica uma atualização iterativa dos parâmetros da rede, de modo que as previsões se tornem cada vez mais próximas das saídas desejadas. Esse processo é repetido para um grande número de épocas até que a rede alcance um desempenho aceitável, classificando corretamente os dígitos manuscritos.

4.7 O problema do desaparecimento do gradiente

Um problema que pode surgir devido a natureza do algoritmo do *back-propagation* é o do *vanishing gradient* (CHI-FENG, 2019), ou desaparecimento do gradiente. Esse problema descreve o comportamento observado nos valores do gradiente que tendem a ser próximos a zero conforme mais camadas são adicionadas no modelo. Dessa forma, cada iteração do algoritmo executa um passo praticamente nulo em direção ao mínimo da função de custo, dificultando o treinamento.

A causa do problema está ligada a certas funções de ativação, como a função sigmoid, descrita na Figura 4.3, que transformam inputs para um intervalo pequeno de valores entre 0 e 1. Para valores de entrada muito distantes de zero a derivada é muito pequena e assim, uma grande mudança na entrada dessa função causa uma pequena variação na sua saída. Esse comportamento é ilustrado na Figura 4.9.

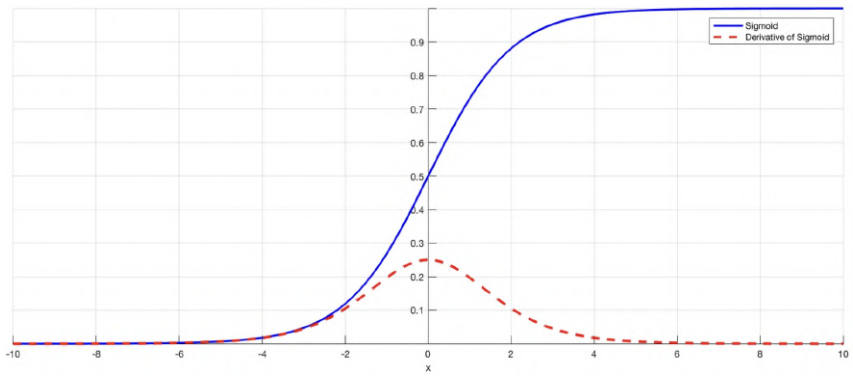

Figura 4.9: Função sigmoid e sua derivada
Fonte: Chi-Feng (2019).

Os gradientes da rede neural encontrados pelo *backpropagation* são calculados pelas multiplicações das ativações de cada camada. Assim, se a rede possuir n camadas que utilizam funções de ativação como a sigmoid, então n pequenas derivadas são multiplicadas, causando em valores que aproximam zero de forma exponencial (CHI-FENG, 2019).

Como solução, é possível utilizar funções de ativação como a $ReLU$, ilustrada na Figura 4.2, que por apenas saturar em uma direção, sofrem menos com o problema (GLOROT; BORDES; BENGIO, 2011).

Além disso, redes residuais se apresentam como uma boa solução e são descritas com mais detalhes na seção 4.9.

4.8 Inicialização dos pesos

Um fator importante que deve se considerar para obter uma convergência rápida no treinamento, é a inicialização dos pesos da rede neural. Uma abordagem comum é inicializar os pesos como valores arbitrários, porém assim, pode haver um retardo ou até mesmo uma paralisia completa do processo de convergência. A desaceleração surge porque as inicializações arbitrárias podem fazer com que as camadas mais profundas da rede recebam entradas com pequenas variâncias, o que, por sua vez, desacelera o *backpropagation*, e retarda o processo de convergência global (KUMAR, 2017).

Um exemplo dos métodos de inicialização mais populares é o *Xavier Initialization* (GLOROT; BENGIO, 2010). Ao seguir essa abordagem, todos os pesos de uma camada são definidos de acordo com a equação 4-27, na qual W representam os pesos calculados e esses estão presentes na distribuição uniforme entre os valores n_j e n_{j+1}, esses sendo os números de neurônios nas camadas j e $j+1$.

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{4-27}$$

4.9 Redes Residuais

Uma arquitetura de rede neural popular, que será explorada pela SRAGN no capítulo 6.3, são as redes residuais. Essas são apresentadas pela pesquisa de Dong et al. (2015) e buscam novas estratégias para serem utilizadas em redes neurais profundas para facilitar o treinamento.

Os problemas que buscam ser tratados são os do desaparecimento do gradiente, descrito na seção 4.7, e o chamado de "degradação da precisão do treinamento", que é um fenômeno observado no qual ao se adicionar mais camadas a um modelo, o erro durante o treinamento pode aumentar.

Essas redes são compostas por blocos denominados blocos residuais e são ilustrados na Figura 4.10. Esses blocos são uma coleção de camadas nas quais as informação são passadas por e ao redor delas por meio das chamadas *skip connections* ou *shortcut connections*.

A intuição por trás dessa abordagem, consiste em a rede aprender os mapeamentos residuais ao invés dos mapeamentos subjacentes. Desse forma, as camadas que poderiam prejudicar a performance são ignoradas.

Também, é esperado que o treinamento seja acelerado já que cada bloco aumenta a quantidade de dados sendo analisados. Isso acontece porque, como as informações do *input* do bloco são passadas adiantes para os blocos subsequentes, a função do bloco deixa de ser captar todas as informações importantes que precisam ser passadas adiante, e sim descobrir quais dados ela pode acrescentar para facilitar os futuros processamentos.

Outra vantagem dessa arquitetura é o fato que, como cada bloco tem um caminho por e através dele, existe um caminho mais curto para os gradientes chegarem nas camadas anteriores e com isso atualizar os seus pesos de forma útil. Esse comportamento é especialmente notável durante o inicio do treinamento quando os pesos são próximos a zero, como mencionado na seção 4.8, já que assim mesmo com o caminho que passa por todas as camadas sendo pouco informativo, ainda é possível fazer atualizações significativas.

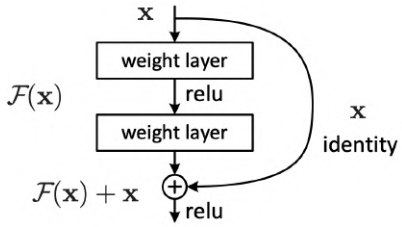

Figura 4.10: Representação de um bloco residual.
Fonte: Dong et al. (2015)

Formalmente, a saída do bloco é definida como $\mathcal{H}(x) = \mathcal{F}(x) + x$, na

qual é chamado de $\mathcal{H}(x)$ o mapeamento das camadas subjacentes, sendo x a entrada do bloco e $\mathcal{F}(x)$ a contribuição desta camada. No caso, a rede aprende $\mathcal{F}(x)$ que é igual a $\mathcal{F}(x) = \mathcal{H}(x) - x$, ou seja, a rede aprende a diferença, ou resíduo, entre a entrada e a saída. O artigo afirma que aprender esse resíduo é mais simples de aprender do que $\mathcal{H}(x)$, principalmente no início do treinamento quando os pesos são valores próximos de zero, e consequentemente as diferenças entre $\mathcal{H}(x)$ e x são pequenas.

O impacto dessa arquitetura é significativo, e desde sua criação, diversas outras arquiteturas e modelos evite estas hipérboles utilizam técnicas e ideias apresentadas nela. Entre esses, se encontra a SRGAN que é abordada no capítulo 6.3, além de outras com contextos de atuação diferentes como a *Wavenet* (OORD et al., 2016), que é um modelo generativo de áudio, ou a *Deeplab* (CHEN et al., 2017), responsável por segmentação semântica de imagens.

4.10 Convoluções

Um tema fundamental para a compreensão dos tópicos abordados nos seguintes capítulos é a operação de convolução. A convolução é uma operação matemática amplamente utilizada em processamento de sinais, visão computacional, aprendizado de máquina e outros campos relacionados. Ela desempenha um papel crucial no processamento de dados, permitindo a aplicação de filtros e a extração de características relevantes dos dados.

Uma convolução modela uma operação que é invariante por translação, ou seja, produz o mesmo efeito, independente da região da imagem em que é aplicada.

No contexto desse trabalho, o estudo das convoluções é direcionado para o seu uso dentro do âmbito do processamento de imagens, ou seja, a operação de convolução é restrita para o seu caso discreto como definido na equação 4-28. Isso significa aplicar a operação de convolução a imagens para criar novas imagens. Quando é afirmado que uma imagem passou por uma convolução, isso significa que ela foi convoluída com uma pequena matriz chamada de *kernel* ou filtro.

Mais especificamente, o comportamento descrito é apresentado com mais detalhes na equação 4-28 (CONTRIBUTORS, 2023a).

$$g(x,y) = w * f(x,y) = \sum_{i=0}^{a}\sum_{j=0}^{b} w(i,j) f(x-i, y-i) \tag{4-28}$$

Em que:

- $g(x,y)$ é a imagem filtrada;
- $f(x,y)$ é a imagem original;
- w é o *kernel*;
- a e b são as dimensões do kernel

De forma simples, a operação de convolução consiste em posicionar o elemento central do *kernel* sobre uma área da entrada, denominada *receptive field* e deslocá-lo pelos pixels da imagem original, multiplicando e somando esses valores gerando assim um novo valor de pixel que representa uma combinação dos pixels avaliados. A Figura 4.11 ilustra a operação de convolução entre uma entrada e um *kernel*. Durante essa operação, o *kernel* é posicionado sobre a entrada, e seus valores são multiplicados pelos valores correspondentes na entrada. Em seguida, esses produtos são somados para gerar um único valor na saída.

Na Figura 4.12, é mostrado um exemplo em que o *kernel* é posicionado no canto superior esquerdo da entrada. A saída parcial correspondente a essa etapa é calculada como 30, pois $10 \times 1 + 25 \times 0 + 20 \times 1 + 32 \times 0 = 30$.

Para concluir o processo e obter a saída final, o *kernel* é deslocado, repetindo as operações mencionadas, até percorrer todos os pixels da entrada. Isso resulta na matriz amarela representada na Figura 4.14.

Figura 4.11: Matriz original em azul e o *kernel* utilizado em verde

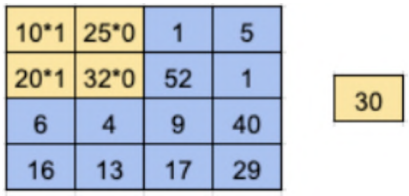

Figura 4.12: Primeira etapa de convolução na esquerda, e na direita em amarelo, a matriz resultante

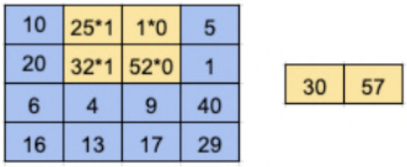

Figura 4.13: Segunda etapa de convolução na esquerda, e na direita em amarelo, a matriz resultante

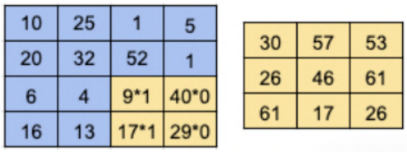

Figura 4.14: Última etapa de convolução na esquerda, e na direita em amarelo, a matriz resultante final

4.10.1 Exemplos de *kernels* conhecidos

O valores que compõem o *kernel* são fundamentais para determinar qual efeito será produzido na imagem original a partir da convolução. Exemplos de *kernels* para manipulações de imagens populares incluem:

– Identitdade: no qual os valores do *kernel* são todos zero com excessão do elemento central que tem valor igual a um, como demonstrado na equação 4-29. Esse *kernel* é não tem efeito nenhum sobre a entrada, ou seja a saída é exatamente igual a entrada.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4-29}$$

– *Gaussian blur*: no qual os valores do *kernel* seguem uma distribuição normal, como exemplificado na equação 4-30 (TIMOTHY, 2021). Esse *kernel* é responsável por causar um efeito de borramento na imagem de entrada.

$$\frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{4-30}$$

– Sobel: no qual o objetivo é a detecção de bordas na imagem. A equações 4-31 e 4-32 mostram exemplos de filtros que destacam as bordas verticais e horizontais, respectivamente (SEAN, 2023).

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{4-31}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{4-32}$$

Para ilustrar o funcionamento desses *kernels* a Figura 4.15 mostra uma imagem original sem alterações, e as Figuras 4.16 , 4.17 e 4.18 exibem o resultado das convoluções com filtros gaussiano, sobel horizontal e vertical respectivamente.

4.10.2 Parâmetros adicionais dos *kernels*

Uma questão importante para se atentar durante o uso de convoluções é que, em geral, as dimensões da imagem gerada tendem a ser menores que a imagem original. Isso se justificava porque, com os exemplos mencionados até aqui, cada pixel da saída, é o resultado da aplicação de uma função em uma janela dos pixels de entrada e, como o *kernel* se move pela imagem em passos fixos, aplicando a operação de convolução a cada posição, é inevitável que a imagem resultante tenha dimensões menores, pois a aplicação do *kernel* não ocorre nas bordas da imagem, reduzindo assim as dimensões da saída em comparação com a entrada.

Para se obter mais controle em relação as dimensões das figuras produzidas, existem os parâmetros *stride* e *padding*.

O *stride* define como o *kernel* se movimenta pela imagem, ou seja, quais os *receptive fields* que serão processados. Por exemplo, ao definir o *stride* como o valor um, se tem na imagem uma grande quantidade de sobreposição entre os *receptive fields*. Por outro lado, ao se atribuir um valor maior ao stride, a quantidade de sobreposições diminui e a saída tem uma dimensão menor (O'SHEA; NASH, 2015). A figura 4.19 ilustra como diferentes valores de *strides* mudam a forma como o *kernel* se desloca pela imagem.

O *padding* é o processo de adicionar uma borda ao redor da imagem de *input* com o propósito de controlar melhor as dimensões das saídas. A Figura 4.20 ilustra o uso do *padding* em uma imagem (O'SHEA; NASH, 2015).

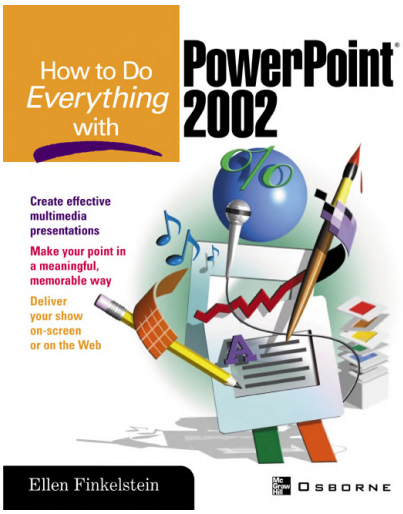

Figura 4.15: Imagem original
Fonte: Autor.

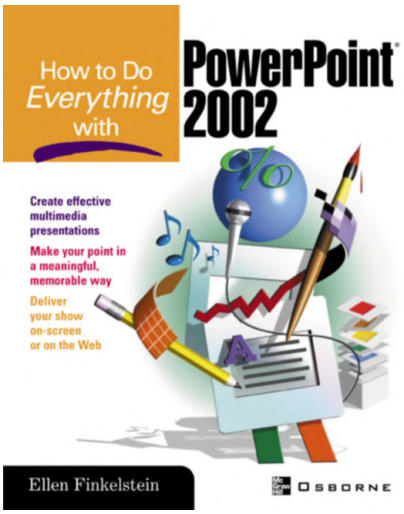

Figura 4.16: Imagem gerada através de uma convolução com *kernel* gaussiano
Fonte: Autor.

Figura 4.17: Imagem gerada através de uma convolução com *sobel* horizontal

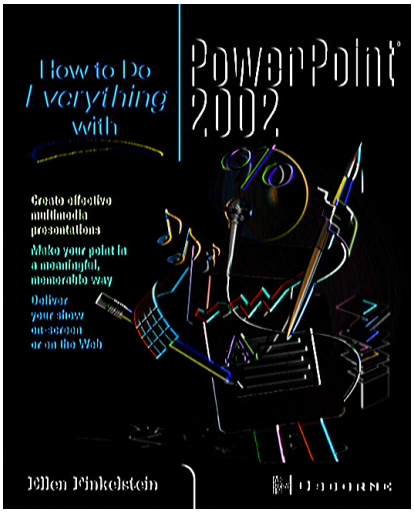

Figura 4.18: Imagem gerada através de uma convolução com *sobel* vertical

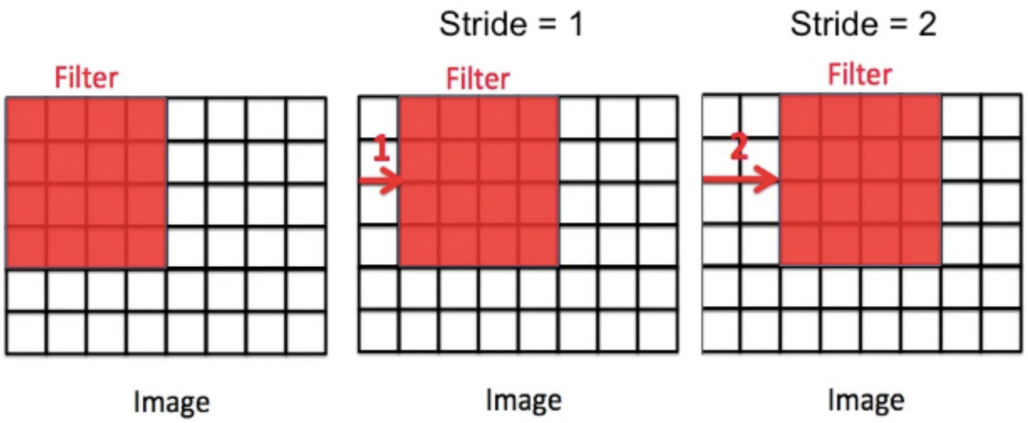

Figura 4.19: Representação visual de como um *filter* é deslocado de forma diferente para valores de *stride* variados.
Fonte: Adaptado de Wu (2017).

Com essas definições feitas, é possível calcular as dimensões de uma imagem gerada por uma convolução a partir da equação 4-33.

$$o = \frac{i + 2p - k}{s} + 1 \tag{4-33}$$

Em que:

- o é a dimensão de saída, após a convolução;
- i é a dimensão de entrada da imagem;
- k é a dimensão do *kernel*;
- p é o tamanho de *padding*;
- s é o tamanho do stride.

A equação é usada para obter o valor de cada dimensão individualmente, por exemplo, ao convolver a imagem de entrada seja 5×7 pixels com um *kernel* de 3×3, considerando um *stride* de tamanho dois e *padding* igual a um, o imagem resultante teria dimensão 3×4, como é demonstrado na equação 4-34 (DUMOULIN; VISIN, 2018, p.15).

$$o_{largura} = \frac{5 + 2 * 1 - 3}{2} + 1 = \frac{4}{2} + 1 = 3 \tag{4-34}$$

$$o_{altura} = \frac{7 + 2 * 1 - 3}{2} + 1 = \frac{6}{2} + 1 = 4 \tag{4-35}$$

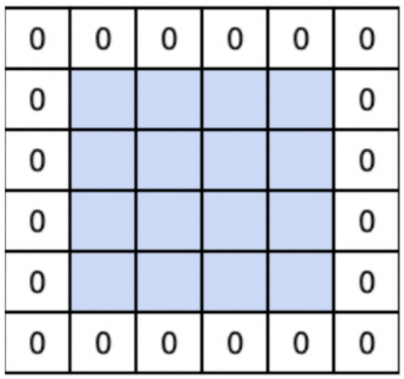

Figura 4.20: Representação visual de como o *padding* é adicionado a uma imagem, representada aqui pela matriz azul.
Fonte: Autor.

4.11 Redes neurais convolucionais

Redes neurais convolucionais, também conhecidas como *convolutional neural networks* ou CNN's, são arquiteturas de redes neurais que possuem melhores capacidades em reconhecimento de padrões de imagem, logo são as redes principais em qualquer tipo de projeto de *deep learning* que envolva processamento de imagens (O'SHEA; NASH, 2015). Assim, é o tipo de arquitetura mais explorada nesse trabalho.

As vantagens obtidas ao utilizar uma CNN no lugar de uma rede neural tradicional são principalmente a redução drástica do número de parâmetros avaliados durante o treinamento, o que facilita o processo como um todo. E também o fato de que a sua performance não depende das características espaciais dos seus dados de treinamento, isto é, as redes convolucionais, por sua natureza, conseguem detectar características independentes de suas posições nas imagens.

As CNN's se diferenciam de uma MLP convencional devido a presença de camadas convolucionais, ou *convolutional layers* que possuem como o objeto que é "aprendido", ou seja, que é modificado durante o treinamento através do *backpropagation*, descrito na seção 4.5, os *kernels* que são abordados na seção 4.10.

De forma similar as redes neurais MLP, os pesos dos *kernels* são os valores atualizados a cada iteração do treinamento. Durante esse processo, a rede aprende a associar os pesos dos *kernels* às características relevantes presentes nas imagens, buscando maximizar a ativação dos filtros quando tais características estão presentes.

Para deixar mais claro o funcionamento e arquitetura de uma CNN a Figura 4.21 ilustra uma implementação simples de uma rede convolucional voltada para classificação de dígitos escritos a mão, composta por cinco camadas. A entrada da rede é uma imagem, que normalmente é representada como matrizes tridimensionais que possuem valores que indicam a intensidade de cada pixel nos canais de cores vermelho, verde e azul (RGB). Já a saída é indicada pela ativação de neurônios específicos na *output layer* que representam qual o dígito que a entrada foi reconhecida como.

De forma geral, o funcionamento da arquitetura exibido na Figura 4.21 consiste nas seguintes etapas chaves: **inicialização**, **camadas convolucionais**, **camada de *pooling*** e, nesse exemplo, **camadas totalmente conectadas**.

A **inicialização** consiste em inicializar a rede com todos os *kernels* e *biases* com valores aletórios, comumente seguindo os princípios abordados no

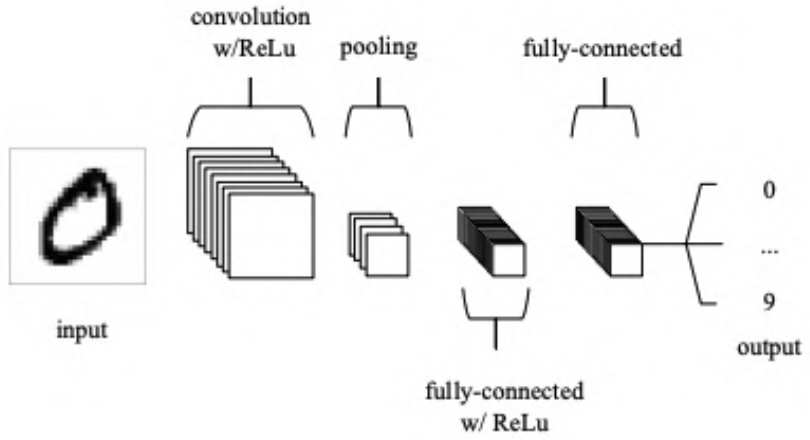

Figura 4.21: Arquitetura de uma CNN simples.
Fonte: O'Shea e Nash (2015).

capítulo 4.8.

Nas **camadas convolucionais**, a imagem é fornecida, e é realizada a convolução entre a imagem e os *kernels*. Assim como descrito na seção 4.10, o *kernel* se move pela entrada calculando o produto escalar para cada um de seus valores. A partir disso, a rede neural aprende quais *kernels* são ativados quando identificam uma característica específica em uma determinada posição da imagem. Essas ativações são conhecidas como *activations*. Dessa forma, a saída de uma camada convolucional é denominada um mapa de ativações, ou *activation maps* ou *feature maps*, em duas dimensões.

Cada *kernel* tem seu correspondente *activation map*, os quais são empilhados ao longo da dimensão de profundidade para formar o volume de saída completo da camada convolucional (O'SHEA; NASH, 2015), como mostra a Figura 4.22. Um exemplo de *activation maps* é mostrado na Figura 4.23, na qual é demonstrado que a partir de uma imagem de *input*, pode ser gerado por exemplo, um *feature map* que detecta bordas verticais e outra que detecta bordas horizontais.

A camada de ***pooling*** é responsável por diminuir a dimensão das saídas das camadas convolucionais, trazendo assim duas consequências benéficas para a performance do modelo. Essas são que a diminuição de dimensionalidade reduz o esforço computacional durante o treinamento, e que essa técnica ajuda a tornar os *feature maps* resultantes mais robustos em relação às mudanças de posição das características na imagem, um conceito conhecido como "invariância local de translação" (GOODFELLOW et al., 2014, p. 342).

A técnica de *pooling*, tem algumas variações, porém para demonstrar seu funcionamento, a operação de *max pooling* é ilustrada na Figura 4.24, na qual áreas de um *activation map*, na esquerda, são selecionadas, e dessas o maior

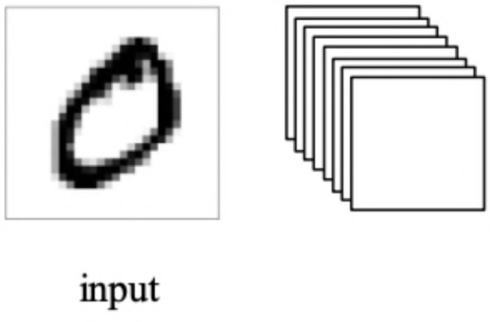

Figura 4.22: Representação visual de como uma imagem de input gera diversos *activation maps*, cada um identificando alguma característica específica da entrada.
Fonte: Adaptado de O'Shea e Nash (2015).

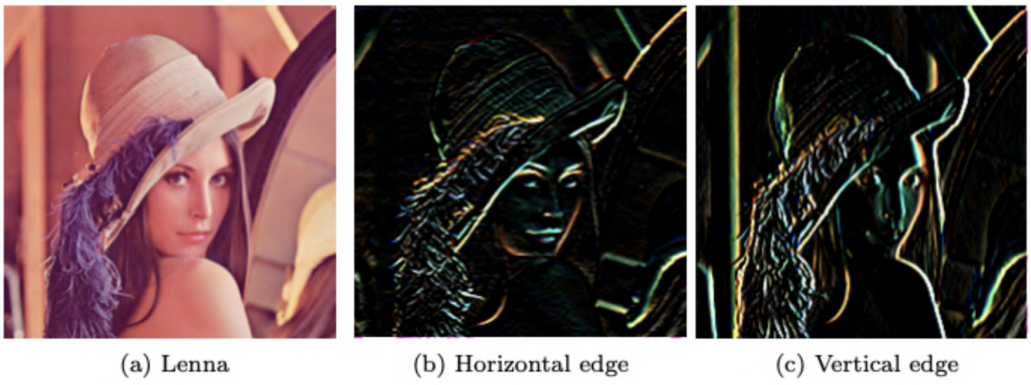

Figura 4.23: Representação da esquerda para a direita da: imagem de entrada, um *activation map* que detecta bordas horizontais, activation map que detecta bordas verticais.
Fonte: Adaptado de Wu (2017).

valor é escolhido como demonstra a matriz na direita.

Nesse caso, as **camadas totalmente conectadas**, são responsáveis por receber a saída das camadas convolucionais e, a partir dela, determinar um valor usado para a classificação, assim como uma MLP de classificação tradicional.

4.11.1 Exemplo de funcionamento de uma CNN

Na última seção 4.10, é descrito as peças fundamentais e o funcionamento geral de uma CNN. Para uma compreensão mais clara, será apresentado um exemplo baseado na arquitetura presente na Figura 4.21 e nas explicações de Starmer (2021).

O exemplo considera uma rede convolucional já treinada e mostra as etapas para se classificar a imagem. No caso, o objetivo é identificar que o digito de entrada é o número zero. A Figura 4.25 mostra como o zero é representado por apenas uma matriz, nos quais os valores 0's representam pixels brancos

Figura 4.24: Exemplo simples da operação de *max pooling*.
Fonte: Autor.

e valores 1's pixels pretos. Além disso, na Figura 4.26, é mostrado o *kernel* usado. Inicialmente, ele possui valores aleatórios, mas, como a rede está pré-treinada, esses valores já estão ajustados corretamente. Além disso, um valor de *bias* igual a dois é considerado.

O exemplo considera que não há *padding* e o *stride* é unitário, dessa forma ao convolver a imagem com o *kernel* o *feature map* presente na Figura 4.27 é criado. Em seguida o *feature map* passa pela função de ativação, nesse caso a ReLU, descrito no capítulo 4.1, gerando o mapeamento presente na Figura 4.28. Em sequência é aplicado o *max pooling* para reduzir a dimensionalidade do *activation map* como demonstra a Figura 4.29. Após essa etapa, o *feature map* é considerado como quatro nós de *input* para uma rede neural MLP tradicional, que tem o seu funcionamento normal, isto é, os valores de entrada são multiplicados com os seus respectivos pesos pré-treinados, então esse valor passa por uma função de ativação e na saída o nó que representa o símbolo zero está ativado, indicando que o resultado encontrado pelo modelo está correto. Esse último comportamento é exibido na Figura 4.30.

Para o desenvolvimento desse trabalho, serão apresentados outras arquiteturas de redes neurais convolucionais mais complexas e com características diferentes. Porém, o funcionamento básico de redes dessa família será similiar ao exemplo apresentado.

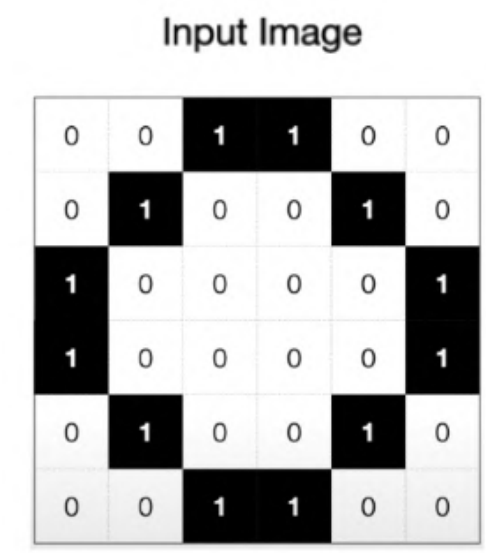

Figura 4.25: Representação do número zero.

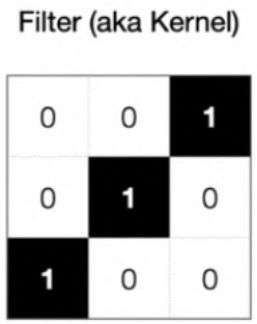

Figura 4.26: *Kernel* pré-treinado.

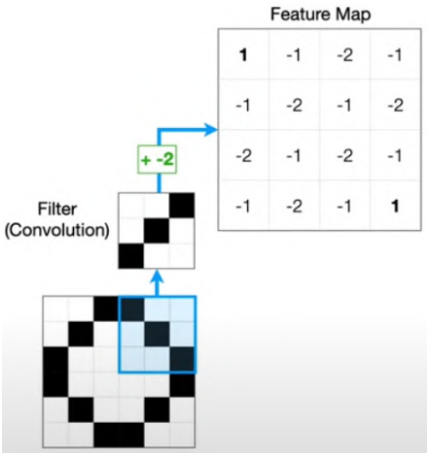

Figura 4.27: *Feature map* calculado após a convolução com o *kernel*.

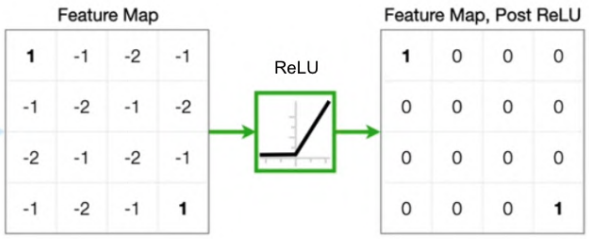

Figura 4.28: *Feature map* após a aplicação da função de ativação ReLU.

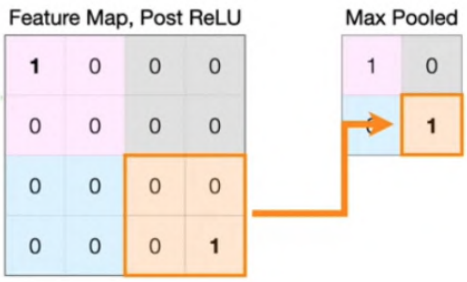

Figura 4.29: *Feature map* após a aplicação do *max pooling*.

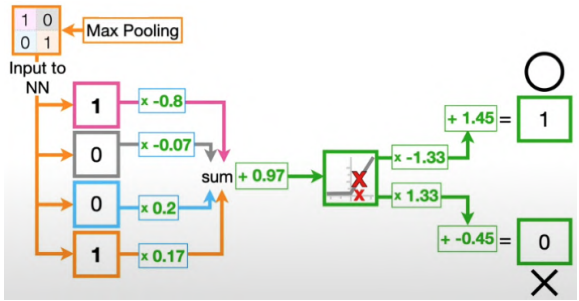

Figura 4.30: MLP que recebe como entradas o *feature map* após o *pooling*.

Figura 4.31: Exemplo demonstrando as etapas de uma rede neural convolucional de classificação simples. Fonte: Adaptado de Starmer J. 2021

5
Método

O processo de desenvolvimento do projeto consiste em selecionar modelos de *deep learning* que buscam resolver o problema da super resolução, realizar os estudos necessários, e buscar implementar uma versão desses modelos. E com isso, conseguir avaliar objetivamente cada modelo. Nesta seção, analisam-se os temas frequentemente adotados no treinamento dos diversos modelos avaliados. No capítulo seguinte (Capítulo 6), discutem-se os detalhes desses modelos.

5.1
Datasets

Para treinar os modelos, é essencial adquirir dados que serão utilizados como entradas para as redes neurais, nos quais todos os processamentos discutidos na seção 4 são executados. Esse conjunto de dados é denominado o *dataset* do modelo.

No caso deste trabalho, os *datasets* são constituídos por uma coleção de imagens. Para modelos de super resolução, existem diversos *datasets* populares que variam significativamente em qualidade e quantidade total de imagens no conjunto (WANG; CHEN; HOI, 2020). Exemplos comuns utilizados neste projeto, são os *datasets* "Set14" (ZEYDE; ELAD; PROTTER, 2012) e "Flickr2K" (LIM et al., 2017), que possuem 14 e 2650 imagens respectivamente.

5.1.1
Dataset Augmentation

Quanto maior for o conjunto de dados utilizado para o treinamento, melhores tendem a ser os resultados dos modelos de *deep learning*. Para atingir esse objetivo, utiliza-se uma técnica chamada *dataset augmentation*, que consiste em aplicar transformações nos conjuntos de dados existentes, criando novas amostras. As transformações podem incluir rotações, espelhamentos, redimensionamentos, cortes e outros ajustes nas imagens. Dessa forma, é possível aumentar o número de inputs para os modelos de forma relevante, sem ser necessário a busca por novas imagens.

Um exemplo de manipulação feita neste projeto é o de extração de *patches* de imagens. Esse processo consiste em retirar pequenos trechos da imagem, denominados *patches* e utilizar esses como os dados de treinamento, de tal forma que a partir de uma imagem, é possível se gerar dezenas de novas amostras para serem consumidas pelos modelos (DONG et al., 2015). A Figura 5.1 demonstra exemplos de *patches* que podem ser obtidos de uma imagem.

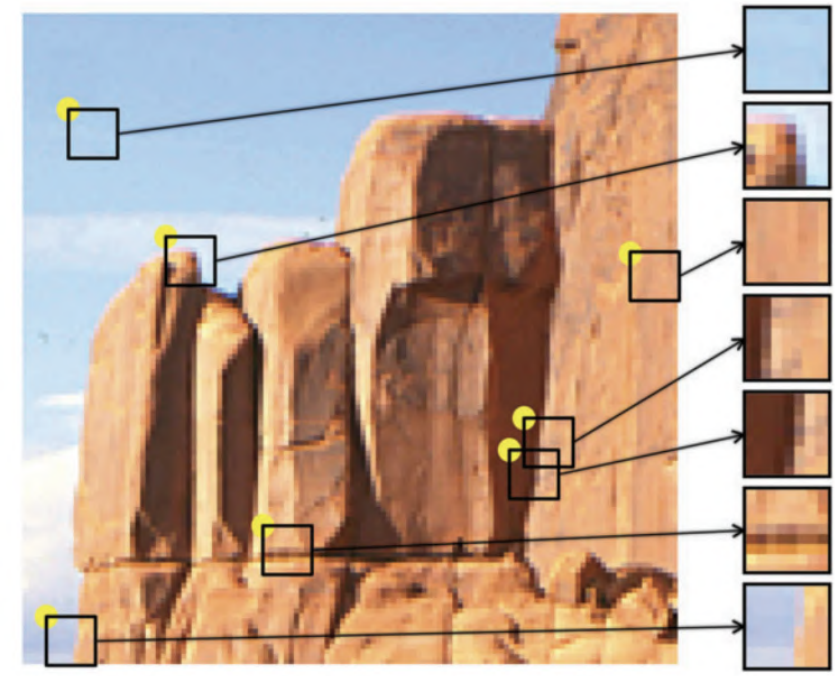

Figura 5.1: Exemplo de *patches* que podem ser extraídos de uma imagem.
Fonte: Stevens et al. (2014)

Neste trabalho, a técnica de extração de *patches* é utilizada durante o treinamento das redes neurais não só por aumentar a quantidade de dados disponíveis, mas também por diminuir a quantidade de processamento total necessário e assim reduzir o tempo total de treino. Isso é uma consequência dos *patches* possuírem uma dimensão menor do que as imagens das quais foram extraídas.

Além dessa técnica, existem também estudos que indicam que, ao se adicionar algum tipo de ruído no dados de treinamento, é promovido que a rede seja treinada a ter resultados mais abrangentes, ou seja, a rede aprende a lidar com variações e imperfeições nos dados de entrada. A introdução controlada de ruído durante o treinamento pode aumentar a robustez do modelo, tornando-o mais capaz de lidar com dados do mundo real, que frequentemente contêm ruídos e imperfeições (BROWNLEE, 2019b).

Um exemplo comum de tipo de ruído inserido é o *Gaussian Noise*, que consiste em somar nos valores de pixel das imagens um conjunto de números

com média zero e desvio padrão de 0.1. Um exemplo da aplicação desse ruído em uma imagem é exibido na Figura 5.2.

Figura 5.2: Exemplo de uma imagem na qual é aplicada ruídos gaussianos
Fonte: Autor

Esse tratamento de se adicionar ruído nas imagens também é utilizado em alguns momentos nesse trabalho para buscar melhores resultados durante o treinamento.

5.1.2 Análise das frequências de *patches*

Uma técnica também explorada neste trabalho é a análise das frequências dos *patches*, abordados na seção 5.1.1. O uso dessa estratégia se mostra importante porque busca garantir que os dados de *input* para os treinamentos contem informação suficiente para que seja possível aprender características das imagens a partir deles.

Para alcançar esse resultado, no momento em que os *patches* são extraídos das imagens originais, é calculada a transformada inversa de Fourier (IDFT), e a partir dela, o espaço de frequência é dividido em dois e é tomada a metade com maiores valores. Com essa metade das maiores frequências presentes no *patch*, é calculada a média dos valores absolutos das transformadas de Fourier nestas frequências. Utilizando essa abordagem, foi empiricamente determinado um valor de limiar que resulta em uma proporção satisfatória de *patches* com características relevantes.

Essa análise é feita porque imagens com mais características presentes, como bordas ou texturas, tendem a possuir frequências significativamente maiores que imagens sem essas qualidades (GOMES; VELHO, 2002). Imagens com frequências baixas, em geral, oferecem uma possibilidade de aprendizado menor para as redes neurais. Assim, se busca evitar analisar esse tipo de imagem, reduzindo o número total de dados de entrada das redes e, consequentemente, diminuindo o tempo de processamento total, sem impactar substancialmente o resultado final do modelo.

Essa análise se mostra especialmente útil em modelos que possuem como dados de treinamento *patches* pequenos, isto é, que possuem largura e altura pequenos, ao redor de 30 pixels cada. Já em *patches* maiores o impacto tende a ser significativamente menor, isso porque quanto maior o tamanho dos *patches*, mais informação está contida dentro dele então casos com nenhuma informação útil são consideravelmente mais raros.

Para exemplificar os resultados dessa análise, a Figura 5.3 mostra na esquerda um exemplo de *patch* que é descartado, e na direita um *patch* aceito.

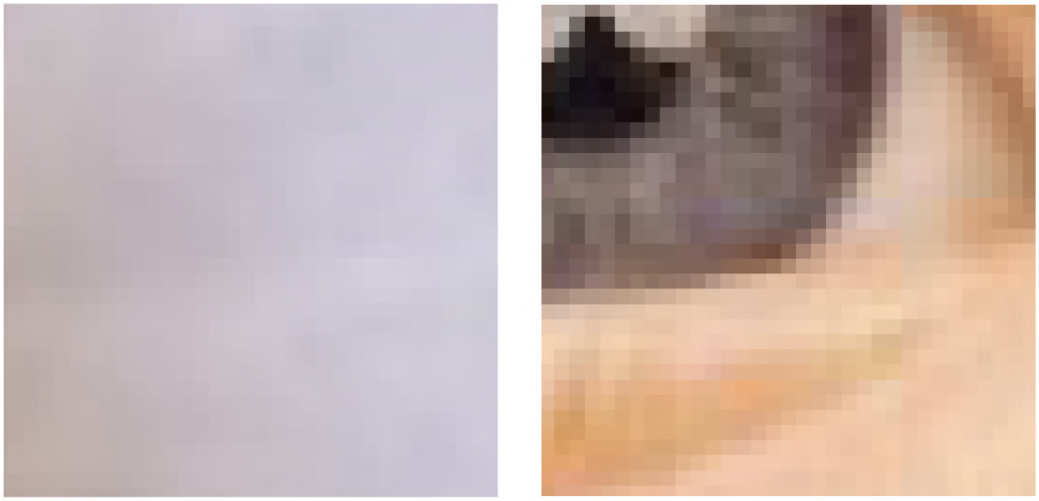

Figura 5.3: Na esquerda um *patch* descartado e na direita um *patch* aceito.
Fonte: Autor.

5.1.3 Divisão do *dataset*

Um problema que pode surgir ao treinar um modelo de *deep learning* é que ele pode apenas apresentar resultados bons para entradas que já foram processadas durante o treinamento. Quando existe esse comportamento, se diz que o modelo está sofrendo com o problema de *overfitting*.

Para impedir que isso aconteça, os *datasets* costumam ser divididos entre três conjuntos: treino, validação e teste, sendo usualmente compostos por 80%, 10% e 10% dos dados contidos no dataset.

O treinamento é apenas executado nos conjuntos de treino e validação, de tal forma que os *weights* e *biases* apenas são atualizados a partir dos resultados no conjunto de treino, e no de validação as perdas são calculadas apenas para examinar como o modelos se comporta em dados que nunca foram avaliados antes. Dessa forma, se ao final do treinamento o custo estiver mínimo em ambos os conjuntos, é garantido que o problema de *overfitting* não ocorreu.

O conjunto de testes é composto por dados que não são usados em nenhum momento do treinamento. Assim as avaliações do modelo com esses dados, demonstram como realmente o modelo se comporta sem nenhuma possível influência dos dados de treinamento (BAHETI, 2021).

5.2 Métricas de avaliação de modelos

Uma vez com os modelos treinados, é necessário utilizar métricas para avaliar a sua performance. Uma das métricas utilizadas é a própria *cost function* aplicada durante o treinamento, que quanto mais perto de zero, mais preciso é o *output*. Porém, dependendo da função de custo utilizada, a interpretação dos resultados pode ser difícil e confusa (SANDS, 2021). Dessa forma, outras métricas são utilizadas para julgar os modelos. Existem diversas métricas utilizadas em projetos de *deep learning*, porém dentro do contexto de Super Resolução as duas principais métricas são voltadas para a medição da semelhança entre imagens e são descritas nas subseções a seguir.

5.2.1 PSNR

Uma das métricas importantes que são utilizadas neste trabalho é a *Peak signal-to-noise ratio* (PSNR) que é um valor expresso em decibéis (dB), descrito na equação (5-1). Com ela, é quantificado a quantidade de ruído ou distorção introduzida durante o processo de compressão ou reconstrução. Quanto maior o valor do PSNR, melhor a qualidade do sinal reconstruído (WANG; CHEN; HOI, 2020).

$$\text{PSNR} = 10 \log_{10} \frac{L^2}{\frac{1}{N} \sum_{i=1}^{N} (I(i) - \hat{I}(i))^2} \tag{5-1}$$

Na qual:

L é o valor máximo que um pixel pode ter, geralmente igual a 255;

N é o número de pixels da imagem;

I é a imagem verdadeira;

$\hat{I}$ é a imagem gerada pelo modelo.

5.2.2 SSIM

A outra principal métrica utilizada no trabalho e no contexto de Super Resolução é o *structural similarity index measure* (SSIM), que é um modelo

baseado na percepção humana que avalia a degradação de uma imagem como uma mudança percebida nas informações estruturais, levando em consideração aspectos como a diferença de luminância e contraste entre as imagens. Dessa forma, quanto maior o valor de *SSIM* entre duas imagens, melhor é a qualidade da imagem super resolvida, indicando que a imagem resultante se assemelha mais à imagem de alta resolução original, de acordo com a percepção humana (NILSSON; AKENINE-MöLLER, 2020).

A definição matemática do SSIM é descrita na equação 5-2 (CONTRIBUTORS, 2023b).

$$SSIM(x, y) = l(x, y)^{\alpha} c(x, y)^{\beta} s(x, y)^{\gamma} \tag{5-2}$$

Sendo que $l(x, y)$ avalia a semelhança na luminosidade, $c(x, y)$ compara o contraste e $s(x, y)$ mede a semelhança na estrutura entre as imagens de entrada x e y. As definições de $l(x, y)$, $c(x, y)$ e $s(x, y)$ se encontram nas equações 5-3, 5-4 e 5-5 respectivamente. Além disso, α, β e γ são parâmetros que comumente são iguais a um.

$$l(x, y) = \frac{2\mu_x \mu_y}{\mu_x^2 + \mu_y^2 + c_1} \tag{5-3}$$

$$c(x, y) = \frac{2\sigma_x \sigma_y}{\sigma_x^2 + \sigma_y^2 + c_2} \tag{5-4}$$

$$s(x, y) = \frac{\sigma_{xy} + c_3}{\sigma_x \sigma_y + c_3} \tag{5-5}$$

De tal forma que:

- μ_x: A média das amostras de pixels de x;
- μ_y: A média das amostras de pixels de y;
- σ_x^2: A variância de x;
- σ_y^2: A variância de y;
- σ_{xy}: A covariância entre x e y;
- $c_1 = (k_1 L)^2$: Variável para estabilizar a divisão com denominador fraco;
- $c_2 = (k_2 L)^2$: Variável adicional para estabilização;
- L: A faixa dinâmica dos valores dos pixels (tipicamente $2^{\#bits\ per\ pixel} - 1$);
- $k_1 = 0.01$ e $k_2 = 0.03$ por padrão.

5.2.3 Exemplos de cálculo das métricas utilizadas

Com o propósito de ilustrar valores reais calculados para as métricas PSNR e SSIM, descritas nas seções 5.2.1 e 5.2.2 respectivamente. A Figura 5.4 demonstra os resultados calculados. Na coluna da esquerda são mostradas as imagens originais, e na coluna da direita, se encontram versões das imagens originais após serem aplicados um ruído gaussiano, descrito na seção 5.1.1, com média igual a zero e desvio padrão igual a 0.1.

Os valores das métricas foram calculados entre as imagens da coluna da esquerda com as da direita. Os resultados estão presentes na Tabela 5.1

Tabela 5.1: Valores de PSNR e SSIM calculados entre as imagens presentes nas Figuras 5.4

Imagem	**PSNR**	**SSIM**
Mulher / Mulher com ruído	24.8163	0.3516
Pássaro / Pássaro com ruído	24.5622	0.3142
Aplicação em 2 imagens iguais	∞	1.0

Figura 5.4: Exemplo dos valores de PSNR e SSIM obtidos ao comparar as imagens da coluna esquerda com as da coluna direita.
Fonte: Autor

6
Modelos

Os modelos estudados, implementados e avaliados se encontram nos capítulos a seguir.

6.1
SRCNN

6.1.1
Descrição do modelo

O primeiro modelo abordado por este trabalho foi o *Super-Resolution Convolutional Neural Network* (SRCNN) (DONG et al., 2015). Esse modelo foi o primeiro a ser analisado por ter sido uma das pesquisas pioneiras para o âmbito de super resolução por meio de técnicas de *deep learning*, e por possuir um design relativamente simples.

O modelo pertence à classe conhecida como *pre-upsampling* SR, na qual a imagem de entrada tem sua resolução aumentada por técnicas de interpolação antes de ser alimentada para a rede neural, como ilustrado na Figura 6.1. A ideia é que a rede seja capaz de adicionar detalhes a uma imagem ampliada por métodos tradicionais, melhorando a qualidade final.

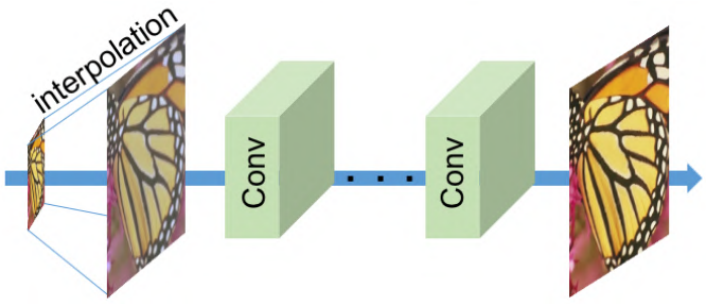

Figura 6.1: Diagrama do método *pre-upsampling*.
Fonte: Haris, Shakhnarovich e Ukita (2018)

O modelo consiste em uma rede convolucional com apenas uma camada de entrada, uma oculta e uma de saída. Essas camadas estão representadas na Figura 6.2 e são denominadas por *Patch extraction and representation*, *Non-linear mapping* e *Reconstruction*, respectivamente.

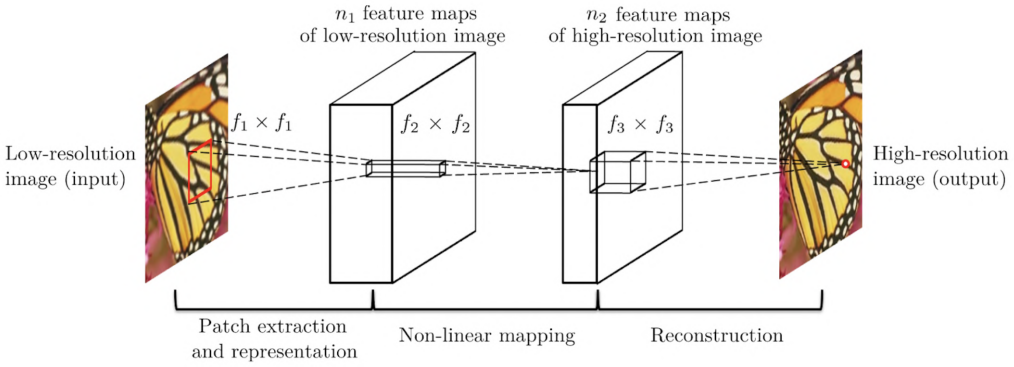

Figura 6.2: Arquitetura do modelo SRCNN.
Fonte: Dong et al. (2015)

A camada de *Patch extraction and representation* é composta por uma camada convolucional convencional, como descrita na seção 4.11, com n_1 filtros de tamanhos $c \times f_1 \times f_1$, no qual c é o número de canais da imagem de entrada, que para imagens coloridas tem seu valor como três, e f_1 é a dimensão do *kernel*. A saída da camada é composta por n_1 *activation maps*, e nelas são aplicadas a função de ativação ReLU, mencionados nas seções 4.11 e 4.1 respectivamente.

A etapa de *Non-linear mapping* tem um formato similar a camada anterior, apenas mudando os tamanhos dos filtros para $n_1 \times f_2 \times f_2$. Sua saída é o mapeamento de cada vetor de entrada de dimensão n_1 para um outro vetor de dimensão n_2. Cada elemento do vetor de dimensão n_2 é conceitualmente uma representação de um patch de alta resolução que é usado para a reconstrução.

A partir de experimentos explorados no artigo original, os valores das variáveis mencionados anteriormente, para uma boa relação entre performance e velocidade são $c = 3$, $n_1 = 64$, $n_2 = 32$, $f_1 = 9$, $f_2 = 5$ ou $f_2 = 1$, $f_3 = 5$, e todas as camadas convolucionais não possuem *padding*. Essas configurações e outras abordagens são exploradas na seção 6.1.2 seguinte.

6.1.2 Avaliação do modelo

O modelo é treinado a partir de uma combinação dos *datasets* "Flickr2k" já mencionando anteriormente e conjunto "DIV2K" (AGUSTSSON; TIMOFTE, 2017) que possui 1000 imagens com resolução de 2K. A partir dessa coleção é aplicada a metodologia de extração de *patches*, descrita na seção 5.1.1 para expandir o número de dados e reduzir dimensão total e consequentemente o custo computacional durante o processamento. Os *patches* extraídos possuem uma dimensão de 33×33 pixels. No total, é utilizado ao redor de 98.000 patches como conjunto de treinamento, e 21.000 como conjunto de validação.

Essas imagens extraídas são consideradas os dados de alta resolução para

o modelo, e as entradas em baixa resolução são uma versão dessas figuras que sofrem um processo de redução de resolução ou, em inglês *downsampling*, por meio de interpolação bicúbica para uma forma de 8×8 pixels seguido por um aumento de volta para a forma original de 33×33 pixels. Assim, esses dados representam y e $\hat{y}$ após o treinamento a partir das entradas, como descritos na seção 4.3, respectivamente. A Figura 6.3 exibe exemplos de *patches* considerados como y e $\hat{y}$.

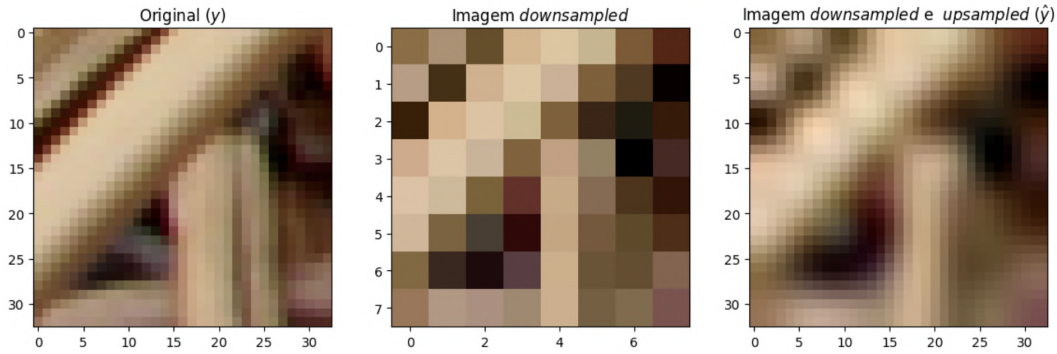

Figura 6.3: Da esquerda para a direita: *patch* original considerado como entrada de alta resolução (y); *patch* após a operação de *downsampling*; *patch* após as operações de *dowsampling* seguida de *upsampling*, considerada a entrada em baixa resolução da rede $\hat{y}$. Fonte: Autor.

Durante o treinamento algumas abordagens diferentes são testadas, essas sendo os casos nos quais $f_2 = 1$ e $f_2 = 5$. Para cada um desses também é feito um experimento com *patches* que passaram ou não pela análise de frequência descrita na seção 5.1.2, aqui denominamos esses *patches* como filtrados e não filtrados, respectivamente, para analisar a mudança de desempenho no modelo. De forma sucinta, as situações testadas são:

- $f_2 = 1$, com *patches* filtrados;
- $f_2 = 1$, com *patches* não filtrados;
- $f_2 = 5$, com *patches* filtrados;
- $f_2 = 5$, com *patches* não filtrados;

As comparações entre tempos total de treinamento é ilustrado na Figura 6.4, além disso, as curvas de perdas em cima dos conjuntos de treino e validação pelas *epochs* estão na Figura 6.5. Ao observar as figuras, nota-se que todos os modelos convergem para um valor próximo em poucas *epochs*. Além disso, a Figura 6.4 mostra que os modelos com o *kernel* $f_2 = 5$ possuem um tempo maior por *epoch*, o que é justificado já que quanto maior o tamanho do filtro, mais operações são realizadas o que leva a um tempo maior de processamento

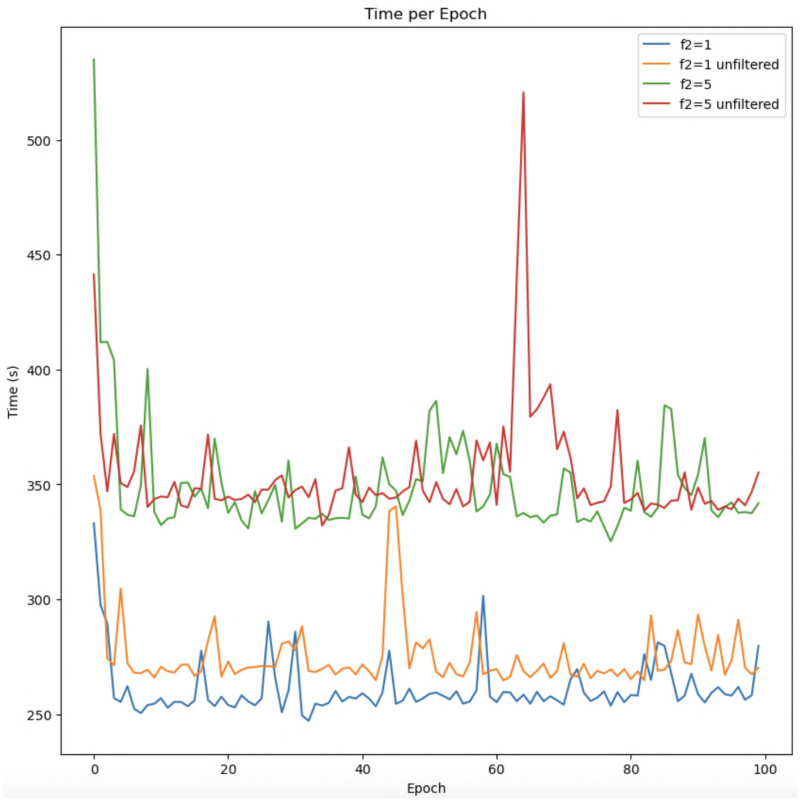

Figura 6.4: Comparação entre os tempos totais em segundos para o treinamento de cada especificação do modelo
Fonte: Autor.

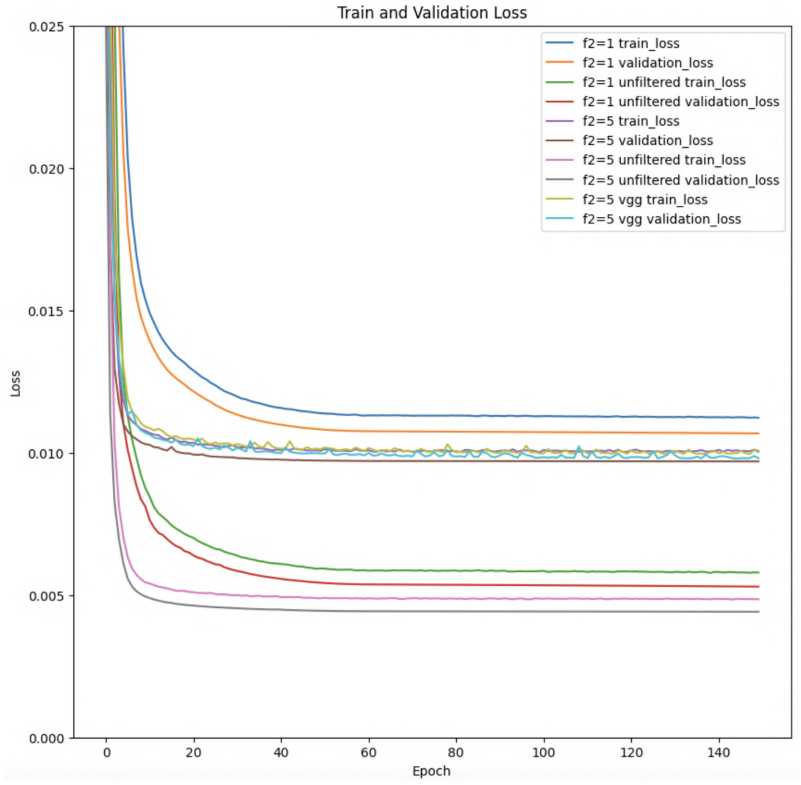

Figura 6.5: Comparação entre as *losses* apresentadas para cada variação do modelo
Fonte: Autor.

6.1.3 Métricas

Para avaliar o resultado da SRCNN, foi calculado o valor PSNR e SSIM, descritos na seção 5.2, médio dos modelos treinados em cima das imagens do *dataset* "Set14". Esse conjunto possui 14 imagens e é considerado como um *benchmark*, ou seja, é popularmente usado para avaliar a performance de modelos de super resolução (WANG; CHEN; HOI, 2020).

A Tabela 6.1 exibe os resultados obtidos. Todos as especficações diferentes superam a interpolação bicúbica, isto é, possuem um valor de PSNR maior. A variação do modelo com a melhor desempenho é a que possui o *kernel* $f_2 = 5$ treinada em cima de dados filtrados.

Tabela 6.1: Valores de PSNR e SSIM calculados para diferentes variações do modelo SRCNN a partir do *dataset* "Set14", com a linha em cinza destacando o melhor resultado obtido

Variações SRCNN	**PSNR**	**SSIM**
f2=1	22.824215	0.634771
f2=1 não filtrado	22.736856	0.625142
f2=5	23.128874	0.645081
f2=5 não filtrado	23.005547	0.639881
Interpolação bicúbica	22.933111	0.659410

6.1.4 Imagem de exemplo

Para exemplificar os resultados obtido, a Figura 6.6 mostra as imagens de alta resolução original, a de baixa resolução, a gerada por meio de interpolação bicúbica e a alcançada por meio do modelo.

É possível observar que as principais diferenças entre as imagens geradas por interpolação bicúbica e pelo modelo são nas suas bordas. O modelo retira a aparência "pixelada" e dá mais destaques as margens.

Figura 6.6: Na primeira linha, da esquerda para a direita: imagem de alta resolução original, imagem de baixa resolução. Na segunda linha, da esquerda para a direita: imagem obtida por interpolação bicúbica, imagem gerada pelo modelo SRCNN.
Fonte: Autor.

6.2 DBPN

6.2.1 Descrição do modelo

O modelo estudado a seguir é o *Deep Back-Projection Networks For Super-Resolution* (DBPN) (HARIS; SHAKHNAROVICH; UKITA, 2018) que foi escolhido por ser um abordagem mais recente e complexa que a SRCNN, analisada na seção 6.1.

O modelo se encontra na classe conhecida como *iterative up and down-sampling* SR. Essa classe é proposta por esse modelo e desde então, já foi empregada em diversas outras pesquisas, como a *Super-resolution feedback*

network (SRFBN) (LI et al., 2019) e *Recurrent Back-Projection Network* (RBPN) (HARIS; SHAKHNAROVICH; UKITA, 2019) por exemplo.

Essa metodologia é ilustrada na Figura 6.7 e consiste em efetuar o *upsampling* e *downsampling* das imagens durante o fluxo dos dados. Ou seja, aumentar e diminuir as dimensões dos *feature maps* gerados pelas camadas convolucionais de forma alternada e ao fim, gerar a imagem super resolvida.

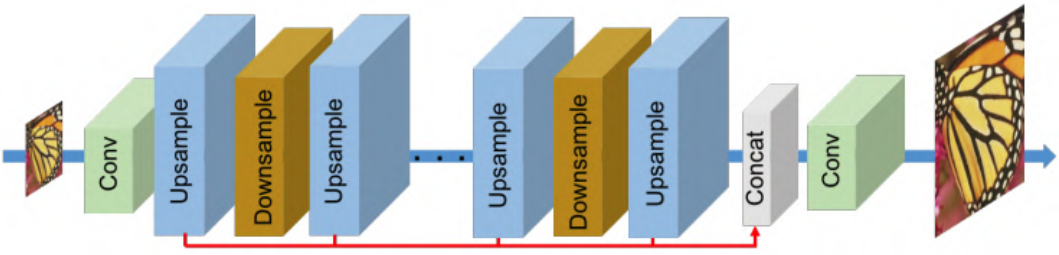

Figura 6.7: Diagrama do funcionamento da metodologia *iterative up and downsampling*. Fonte: Adaptado de Haris, Shakhnarovich e Ukita (2018).

A ideia por trás dessa abordagem é que em arquiteturas mais convencionais, como a SRCNN descrita no capítulo 6.1, e em diversas outras, costuma-se aplicar uma série de camadas convolucionais para extrair o máximo de características possíveis da imagem em baixa resolução e ao fim retornar esses *feature maps* gerados para as dimensões desejadas, com a intenção de inserir características não presentes na imagem de entrada. É proposto pelo artigo que essas metodologias não são capazes de fazer um mapeamento completo das imagens em baixa resolução para alta resolução, devido as poucas características disponíveis na imagem em baixa resolução. Já nas metodologias baseadas em *iterative up and downsampling*, esse esquema possibilita que as redes preservem os componentes de alta resolução ao aprenderem vários operadores de *upsample* e *downsample*, gerando características mais profundas para construir inúmeras informações de baixa resolução e alta resolução. Além disso, a arquitetura baseada no uso desses operadores possui uma natureza autocorretiva, que também busca gerar resultados melhores.

As estruturas fundamentais da DBPN, que efetivamente executam as operações *upsample* e *downsample* são os blocos de projeção, chamados de *Up-Projection Units* e *Down-Projection Units* respectivamente, que tem seus funcionamentos descritos a seguir.

6.2.2 *Up-projection unit* e *Down-projection unit*

O funcionamento do *up-projection unit* é ilustrado pela Figura 6.8 e consiste no sequência de passos a seguir:

1. Ampliação de escala: $H_t^0 = (L^{t-1} * p_t) \uparrow_s$

2. Redução de escala: $L_0^t = (H_0^t * g_t) \downarrow_s$
3. Residual: $e_t^l = L_0^t - L^{t-1}$
4. Ampliação de resíduo: $H_1^t = (e_t^l * q_t) \uparrow_s$
5. *Feature map* de saída: $H_t = H_0^t + H_1^t$

Sendo $*$ a operação de convolução abordada no capítulo 4.11, $\uparrow_s$ e $\downarrow_s$ os operadores de *upsampling* e *downsampling* com o fator de escala s, e por fim, g_t são as camadas de convolução que diminuem a escala da entrada, e p_t e q_t são as camadas de deconvolução, que efetuam a operação de convolução transposta, que é similar as convoluções porém ela aumenta a dimensionalidade de sua entrada, na etapa t.

O seu funcionamento consiste em receber como *input* os *feature maps* de baixa resolução L^{t-1}, previamente computados e os mapear para um novo *feature map* intermediário H_0^t (item 1), então a operação de redução de escala é feita em H_0^t para gerar L_0^t (item 2). Em sequência, é calculada a diferença, ou resíduo e_t^l, entre os *feature maps* encontrado inicialmente L^{t-1} e o reconstruído L_0^t (item 3). A partir disso é gerado um novo mapeamento intermediário H_1^t (item 4), e por fim, a saída do bloco H_t é calculada como a soma dos *feature maps* intermediários encontrados (item 5).

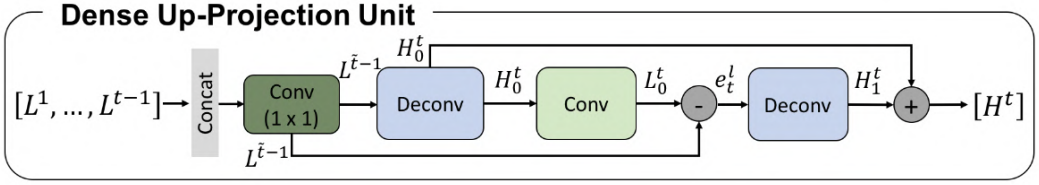

Figura 6.8: Esquema de um *up-projection Unit*.
Fonte: Adaptado de Haris, Shakhnarovich e Ukita (2018).

O funcionamento do *down-projection unit* é similar ao *up-projection unit* e tem o seu esquema ilustrado na Figura 6.9 e tem os procedimentos descritos a seguir:

1. Redução de escala: $L_t^0 = (H_t * g_t') \downarrow_s$
2. Ampliação de escala: $H_t^0 = (L_t^0 * p_t') \uparrow_s$
3. Residual: $e_t^l = H_t^0 - H_t$
4. Ampliação de resíduo: $L_t^1 = (e_t^l * g_t') \downarrow_s$
5. *Feature map* de saída: $L_t = L_t^0 + L_t^1$

A descrição do que acontece nessa rotina é análoga a sequência de passos presentes nos *up-projection units*, porém as operações de *upsampling* e *downsampling* são invertidas para se obter ao fim, um *feature map* que representa o espaço em baixa resolução.

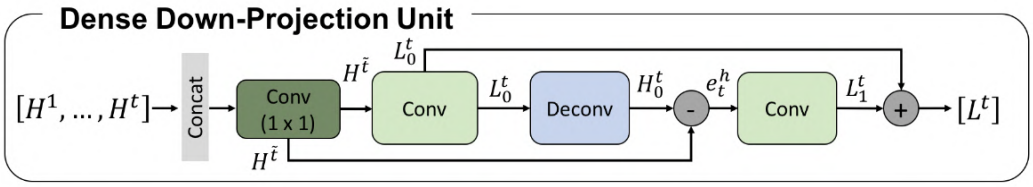

Figura 6.9: Esquema de um *down-projection Unit*.
Fonte: Adaptado de Haris, Shakhnarovich e Ukita (2018).

A intuição por trás dessas operações é que ao se calcular os *feature maps* de saída como a soma dos *feature maps* gerados e os seus resíduos em relação a entrada, acontece um processo de autocorreção das características extraídas, isto é, ao acrescentar o *activation map* residual na saída, espera-se que possíveis ajustes sejam feitos nos valores encontrados.

6.2.3 Arquitetura da rede

Com os blocos de projeção definidos, é possível compreender a arquitetura completa da DBPN demonstrada na Figura 6.10. O modelo recebe como entrada uma imagem em baixa resolução e então passa por três etapas, dais quais a última retorna a imagem de alta resolução desejada. Essas são denominadas ***initial feature extraction***, ***back-projection stages*** e ***reconstrucion***.

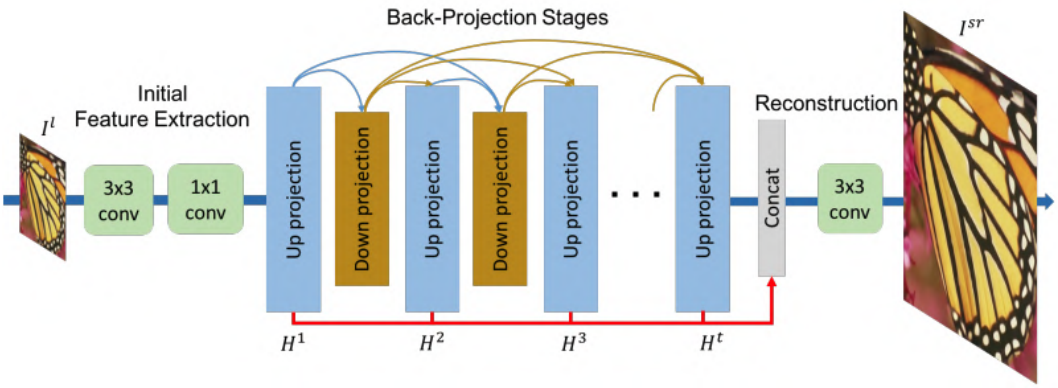

Figura 6.10: Arquitetura do modelo DBPN
Fonte: Adaptado de Haris, Shakhnarovich e Ukita (2018).

No momento inicial de ***initial feature extraction***, a imagem de entrada em baixa resolução passa por duas camadas convolucionais com o

intuito de gerar os primeiros *feature maps* que serão utilizados pelo processo seguinte.

Em seguida, começa a fase de ***back-projection stages***, que utilizam os *feature maps* criados anteriormente. Essa é composta por uma sequência intercalada de T *up-projection units* e $T - 1$ *down-projection units*, nas quais as saídas de cada bloco são concatenadas entre si e então são utilizados como as entradas dos blocos subsequentes. Dessa forma, o comportamento de autocorreção, descrito na seção 6.2.2 é reforçado.

Como valores para T, o artigo sugere três configurações para a rede. Esses sendo $T = 2$, T = 4 e T = 6, de tal forma que quanto maior o valor de T, maior a capacidade de extração de *features* das entradas, porém também é maior o nível de esforço computacional necessário para treinar o modelo. Dessa forma, é preciso fazer uma escolha considerando essa relação, e para este trabalho, o valor de $T = 6$ é considerado, já que ele promete resultados superiores em relação as outras variações.

Ao fim do processo o conjunto concatenado de *feature maps* em alta resolução passam por uma última camada convolucional que gera a imagem super resolvida desejada. A utilização da concatenação de todos os *feature maps*, criados nas diversos blocos de projeção, para gerar a imagem final, proporciona a rede uma forte habilidade de reconstrução de características ausentes na imagem em baixa resolução, já que todas as *features* calculadas em diversos níveis de abstração são considerados.

6.2.4 Avaliação do modelo

O modelo é avaliado com o mesmo *dataset* de origem utilizado para o treinamento da SRCNN, descrito na seção 6.1.2, isto é uma combinação das coleções "Flickr2k" e "DIV2K", porém nesse caso, os *patches* extraídos possuem dimensões de 128×128, e nesse caso, não é necessário fazer a filtragem de frequência, abordada no capítulo 5.1.2, já que com imagens desse tamanho, não se apresentou figuras que não possuem características que possam ser extraídas pelo modelo durante o treinamento. No total, é utilizado ao redor de 88.000 *patches* como conjunto de treinamento, e 13.000 como conjunto de validação.

Os *patches* extraídos são considerados os dados em alta resolução, e neles são aplicados uma interpolação bicúbica para reduzir sua escala em quatro vezes, ou seja suas dimensões ficam iguais a 32×32, e essa versão reduzida são as entradas de baixa resolução para o modelo.

Inicialmente o treinamento é feito por um total de 100 *epochs*, com uma

duração de aproximadamente oito minutos por epoch, utilizando como *learning rate*, abordado na seção 4.3, um valor variável no formato de cosseno, isto é, o valor inicialmente é igual a 10^{-3} e diminui até a 10^{-6} seguindo o formato de uma curva cosseno. As curvas de perdas assim como a curva do *learning rate* utilizado se encontram na Figura 6.11 e, nela observa-se que rapidamente, isto é, em poucas *epcohs*, o erro diminui rapidamente e se estabiliza em valor fixo, aparentando ter atingido um *plateau*.

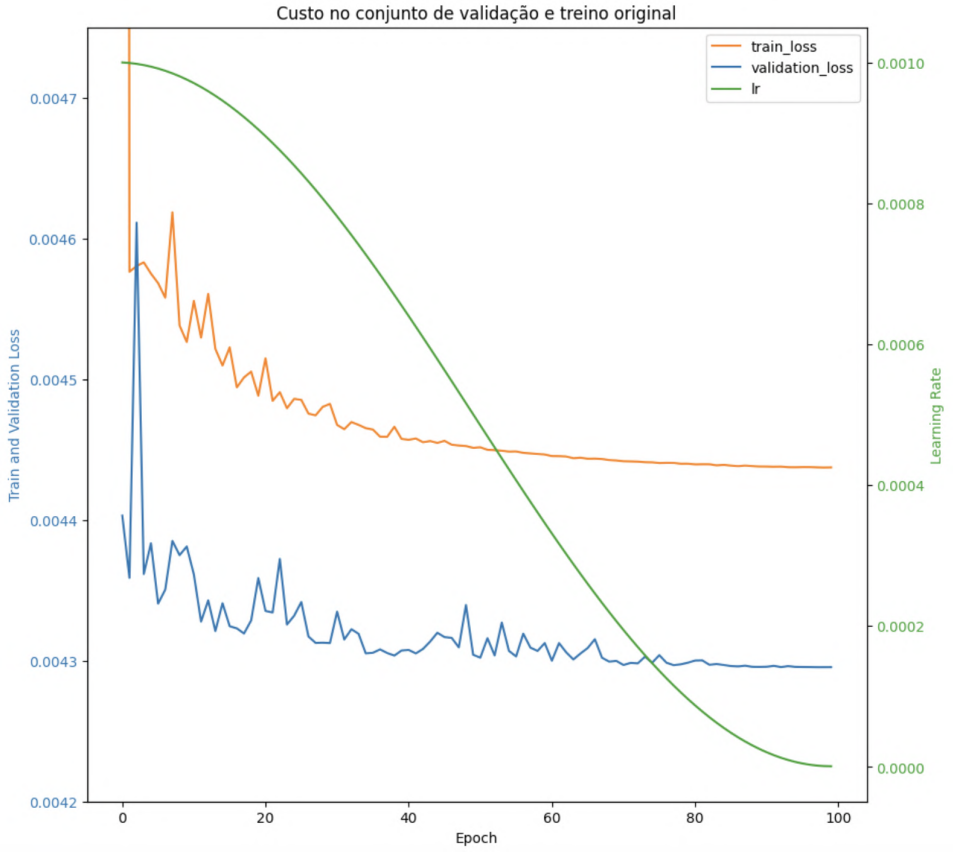

Figura 6.11: Curvas de custo do conjunto de treino em azul, de validação em laranja e a curva da variação do *learing rate* em verde.
Fonte: Autor.

Porém, os resultados encontrados (que são abordados com mais detalhes na seção seguinte 6.2.5), apesar de bem sucedidos no mérito de aumentar a resolução da imagem de entrada, não foram tão bons quanto esperados ou em relação aos resultados obtidos no artigo original. Assim, outras variações do modelo foram testadas para buscar obter resultados superiores.

As variações testadas variam em comportamento do *learning rate*, número de *epochs* computadas, mudanças de *dataset* utilizado e são apresentados a seguir.

A primeira mudança feita foi treinar novamente o modelo pelo dobro de *epochs*, ou seja, 200 *epochs* e também reiniciar o comportamento do *learning rate*, isto é, voltar o seu valor para o número original e o diminuir novamente

para observar se o comportamento das curvas de custo iriam se modificar. Porém foi observado que as métricas calculadas, presentes na seção 6.2.5, e o gráfico de curvas de custo, presente na Figura 6.12 foram praticamente idênticas a primeira iteração do modelo com a metade do treinamento total.

Após isso, foi feito o treinamento novamente substituindo o *learning rate* variável, por um valor constante de 10^{-3}. Nesse caso a performance foi ligeiramente inferior a instancia do modelo treinada com o *learning rate* variável. A Figura 6.13 mostra como o comportamento dos custos se mantém no mesmo padrão já encontrado.

Em sequência foi adicionado o *Gaussian noise*, descrito na seção 5.1.1, nos *patches* extraídos para o treinamento. E nesse caso, uma piora não insignificante foi registrada nas métricas. O comportamento das curvas de custo, presente na Figura 6.14 segue novamente o mesmo padrão, porém com um *plateau* presente como um valor mais alto.

Por fim, foi testado repetir o treinamento com um *learning rate* com um formato triangular, isto é, que oscila entre valores de 10^{-3} e 10^{-6}, com a esperança que ao se aumentar e diminuir os valores repetidamente, se durante o treinamento um mínimo local fosse encontrado, seria possível sair dele. Esse experimento resultou em um modelo que divergiu completamente, de tal forma que a sua função de custo atinge valores na magnitude de 10^{21}, como demonstrado na Figura 6.15 e assim, tanto as métricas calculadas, como as imagens geradas tiveram resultados péssimos como consequência. Esse caso, apesar de ser negativo, tem a sua importância, já que exemplifica como a escolha de um *learning rate* adequado é fundamental para que a rede seja treinada corretamente, assim como é mencionado na seção 4.3.

6.2.5 Métricas

Para avaliar os resultados da DBPN, foi utilizado a mesma metodologia que na seção 6.1.3, isto é, são calculados os valores de PSNR e SSIM a partir da aplicação do modelo em imagens do dataset "Set14". Esses resultados são exibidos na Tabela 6.2, e nela pode se observar como as variações de treinamento impactam as métricas calculadas.

Com exceção do caso em que o treinamento foi realizado com o *learning rate* variando de forma triangular, todos os métodos de treinamento superaram a interpolação bicúbica. O treinamento original, que consistiu em 100 *epochs* com o *learning rate* variando como uma função cosseno, conforme descrito na seção 6.2.4, obteve o melhor desempenho, sendo destacado em cinza. É importante destacar também que o treinamento com o *learning rate* triangular,

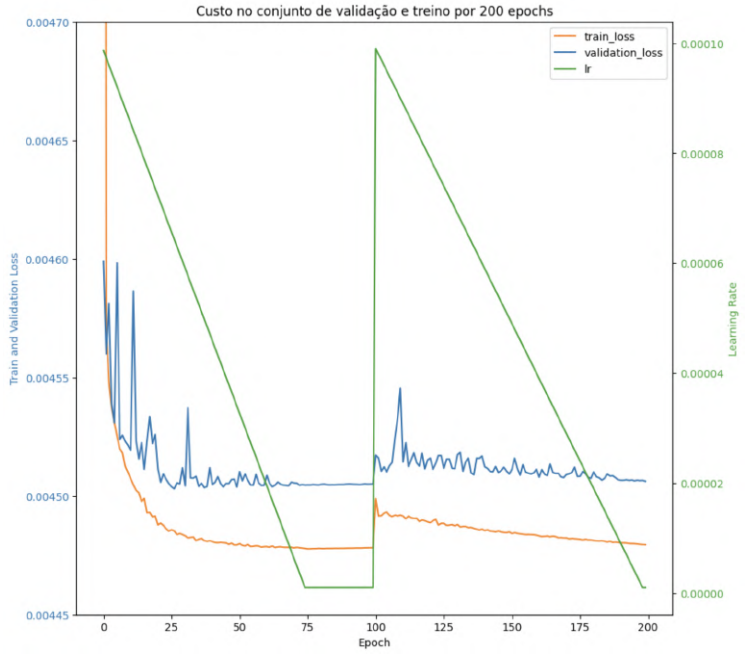

Figura 6.12: Curvas de custo avaliada por 200 *epochs* do conjunto de treino em azul, de validação em laranja e a curva da variação do *learing rate* em verde.
Fonte: Autor.

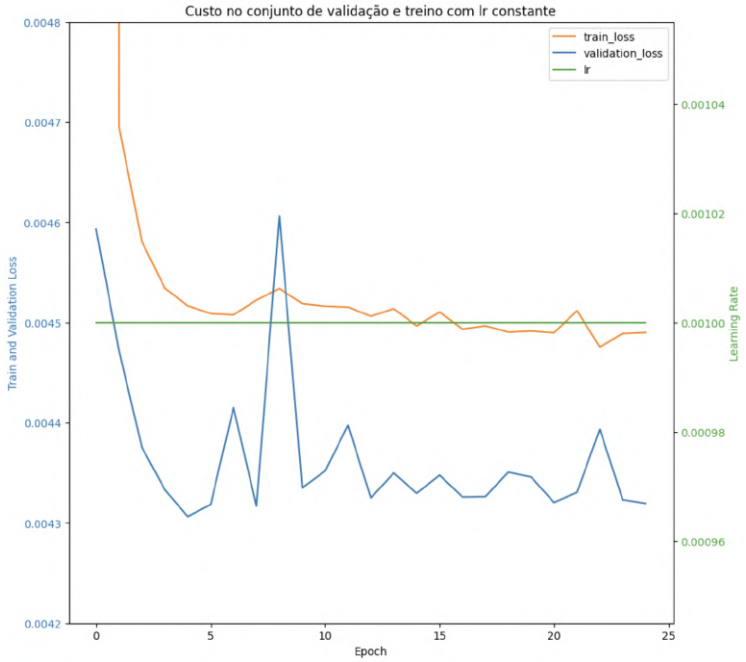

Figura 6.13: Curvas de custo avaliada, com um valor de *learning rate* constante, *epochs* do conjunto de treino em azul, de validação em laranja e a curva da variação do *learing rate* em verde.
Fonte: Autor.

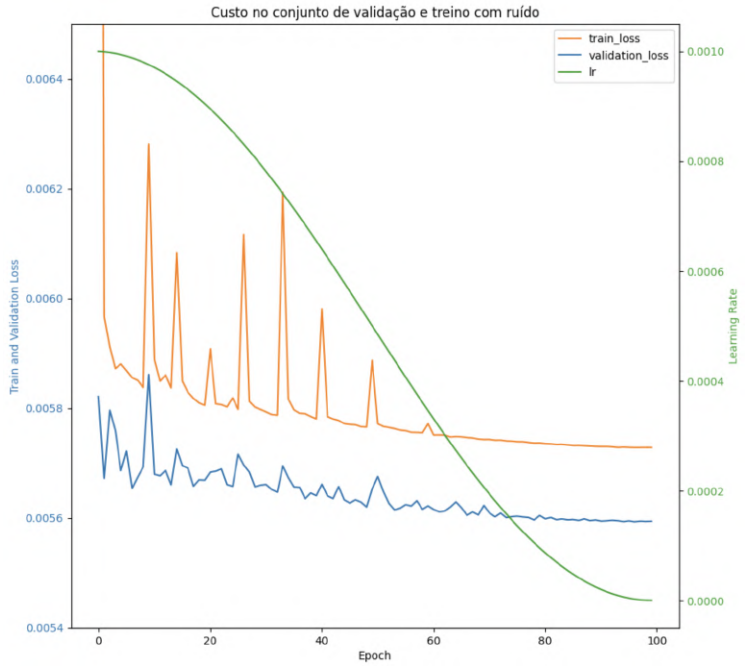

Figura 6.14: Curvas de custo avaliada, com o *Gaussian noise* aplicado nos dados de treinamento, do conjunto de treino em azul, de validação em laranja e a curva da variação do *learing rate* em verde.
Fonte: Autor.

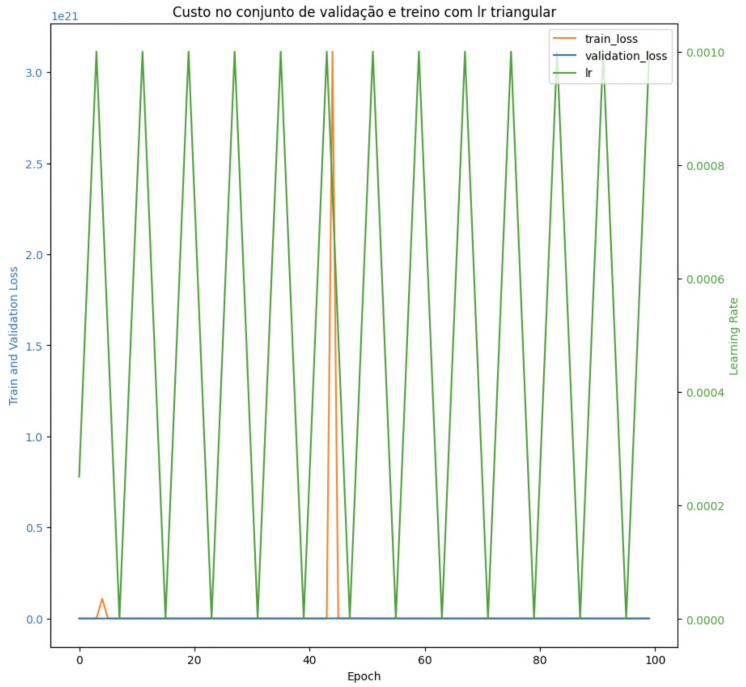

Figura 6.15: Curvas de custo avaliada, com o *learning rate* variando de forma triangular, conjunto de treino em azul, de validação em laranja .
Fonte: Autor.

colorido de rosa, obteve resultados significativamente inferiores.

Tabela 6.2: Valores de PSNR e SSIM calculados para diferentes variações do modelo SRCNN a partir do *dataset* "Set14", com a linha em cinza destacando o melhor resultado obtido

Variações DBPN	**PSNR**	**SSIM**
Original	23.522642	0.6629516
Learning rate constante	23.486498	0.6609573
200 *epcohs*	23.520082	0.6628312
Gaussian noise nos dados	22.989210	0.6036008
Learning rate triangular	-96.925797	3.627203e-12
Interpolação bicúbica	22.933111	0.659410

Os resultados de forma geral foram positivos porém, como já comentado no capítulo 6.2.4, não foram tão bons quanto o esperado. Mesmo com os treinamentos com parâmetros diferentes, não foi possível superar o resultado encontrado originalmente. A explicação de os resultados não serem tão bons quanto o obtido no artigo original muito provavelmente é devido a quantidade de dados usados para o treinamento, já que nesse trabalho o tamanho do *dataset* total foi ao redor de 100.000 imagens, e para o treinamento do modelo oficial foi utilizado principalmente o conjunto *ImageNet* (DENG et al., 2009), que contém mais de 1.250.000 imagens.

Além disso, uma possível explicação para a performance dos modelos DBPN, e SRCNN apresentado na seção 6.1, é fato dos dois possuírem arquiteturas relativamente superficiais, isto é, que possuem um número de camadas convolucionais pequenos, catorze e duas respectivamente. É possível que caso mais camadas desse tipo fossem adicionadas, mais características conseguiriam ser capturadas das imagens, gerando resultados finais superiores.

6.2.6 Imagem de exemplo

Para exemplificar os resultados obtidos, a Figura 6.16 mostra as imagens de alta resolução original, a de baixa resolução, a gerada por meio de interpolação bicúbica e a alcançada por meio da instância do modelo com o melhor resultado apontado pelas métricas na seção 6.2.5.

Pode-se notar que as discrepâncias mais significativas entre as imagens geradas por meio da interpolação bicúbica e as produzidas pelo modelo estão concentradas nas áreas das bordas. O modelo suaviza a sensação de "pixelização" e realça os detalhes nas margens, assim como observado no modelo SRCNN, exemplificado no capítulo 6.1.4.

Além disso, para demonstrar como apenas a variação do *learning rate* pode impactar significativamente a performance do modelo, a Figura 6.17 exibe como a mesma imagem da borboleta, presente na Figura 6.16, ao ser aplicada na instância do modelo treinada com o *learning rate* seguindo o comportamento presente na Figura 6.15, é incompreensível.

Figura 6.16: Na primeira linha, da esquerda para a direita: imagem de alta resolução original, imagem de baixa resolução; Na segunda linha, da esquerda para a direita: imagem obtida por interpolação bicúbica, imagem gerada pelo modelo DBPN.
Fonte: Autor.

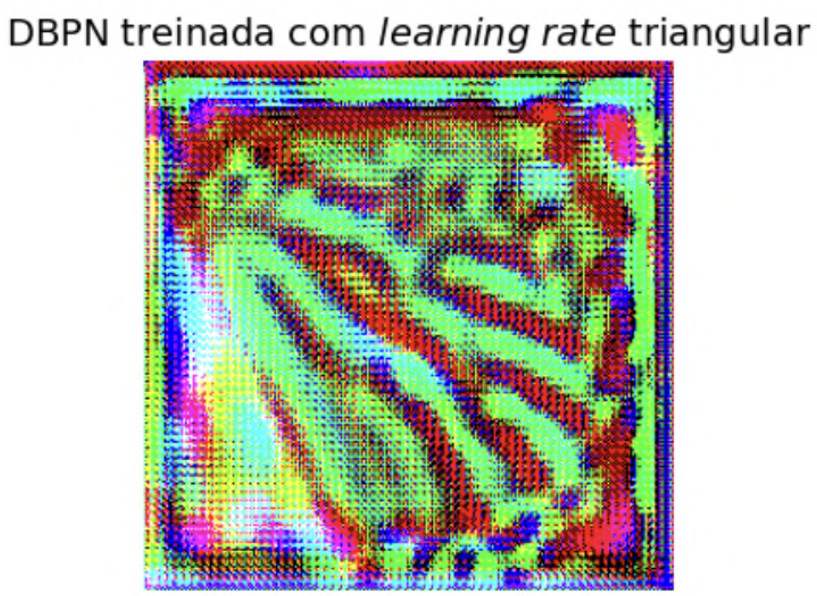

Figura 6.17: Demonstração de como escolher *learning rate* não apropriado para o treinamento pode gerar resultados ruins.
Fonte: Autor.

6.3 SRGAN

6.3.1 Descrição do modelo

O próximo modelo estudado é o *super-resolution generative adversarial network* (SRGAN) (LEDIG et al., 2017) que foi selecionado por ser uma abordagem consideravelmente diferente das redes estudadas nas seções anteriores e por possuir resultados interessantes.

O grande diferencial da arquitetura da SRGAN para outros modelos mais convencionais é que o seu treinamento consiste em avaliar duas redes neurais, a rede Discriminadora e a Geradora. Além disso, possui um foco em proporcionar alternativas para as abordagens que buscam apenas melhorar a performance do modelo a partir de uma função objetiva, como o uso de MSE como função de custo durante o treinamento. Assim, o modelo propõe uma *loss function* denominada *perceptual loss* explicada em mais detalhes na seção 6.3.3 que tem uma prioridade maior em ensinar a rede a reproduzir detalhes de altas frequências nas imagens geradas e não necessariamente em possuir bons resultados nas métricas de avaliações comuns, como a PSNR descrita na seção 5.2.1.

A rede geradora é a uma rede neural que, a partir da imagem de entrada, gera uma nova imagem, com características adicionadas. Essa é a rede que

efetua o processo de Super Resolução.

Conceitualmente, a rede discriminadora é uma rede neural de classificação binária, que recebe como entrada uma imagem e tem como a sua saída um valor indicando se a entrada é uma imagem verdadeira ou se foi fabricada pela rede geradora.

O processo de treinamento como um todo consiste em, simultaneamente, treinar a rede discriminadora para saber afirmar se as suas imagens de entrada são verdadeiras ou falsas e, a partir disso, treinar a rede geradora para criar imagens que buscam enganar a rede discriminadora. Dessa forma, a função de perda da rede geradora possuí um valor que representa se a imagem é percebida como verdadeira ou não. Assim, é esperado que ao fim do treinamento, as imagens geradas pela rede geradora serão tão próximas do valor real que a rede discriminadora não saberá as distinguir de imagens reais.

A intuição dessa abordagem que a torna diferente dos outros modelos estudados nesse trabalho é que, como é utilizada uma função de perda que não é baseada exclusivamente nos valores entre as diferenças de pixel entre as imagens reais e geradas pelo modelo, as figuras resultantes possuem uma aparência consideravelmente mais nítida. Isso porque, funções de perda como a MSE, detalhada na seção 5.2, ensinam as redes que, a partir das diversas soluções plausíveis, o valor que mais aproxima o erro a zero é uma média das possibilidades, promovendo assim, imagens que tendem a possuir aparências mais borradas. Já a GAN é incentivada pela sua função de custo própria, que é detalhada na seção 6.3.3 a produzir figuras que possuam características mais destacadas para se assimilar com as figuras reais. A Figura 6.18 busca ilustrar o comportamento descrito, nela pode-se observar como a solução baseada em GAN (em amarelo) gera um resultado significativamente mais nítido que as soluções baseads em MSE (em azul).

Consequentemente, devido à natureza da função de perda usada pela SRGAN, que não se limita a buscar apenas diferenças nos valores de pixel para minimizar suas perdas, as principais métricas analisadas na seção 5.2, que são usadas para avaliar o desempenho dos modelos, frequentemente não produzem resultados favoráveis. Em alguns casos, essas métricas podem até mostrar resultados piores em comparação com métodos determinísticos, como a interpolação. Isso levanta dúvidas sobre a eficácia das métricas usadas nesse contexto. Mesmo quando as imagens parecem ser visualmente semelhantes às originais aos olhos humanos, muitas vezes o valor do PSNR delas é inferior ao de imagens que tiveram sua resolução aumentada, por exemplo, usando interpolação bicúbica.

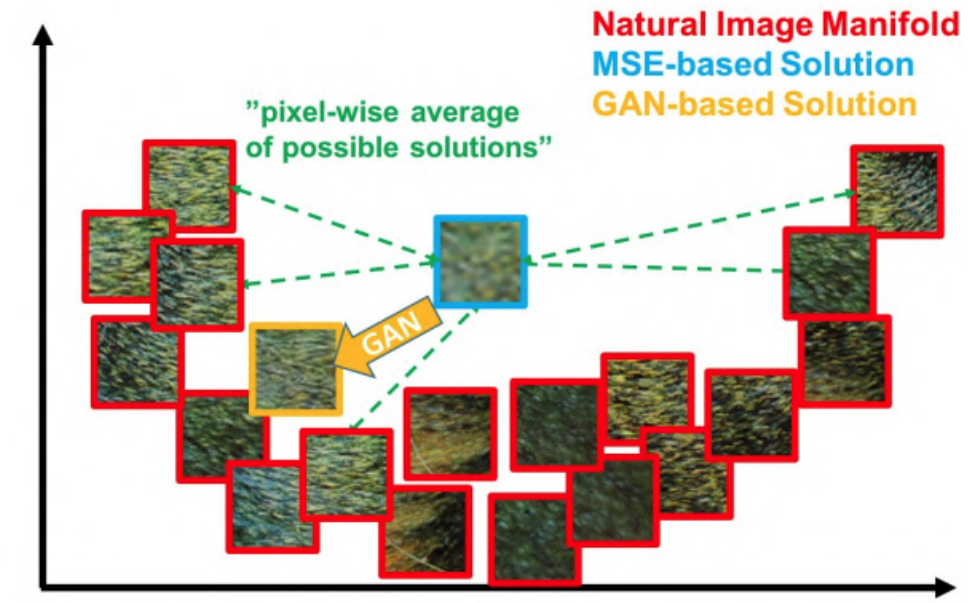

Figura 6.18: Diferença conceitual entre as funções de custo de redes baseadas em MSE e em GAN
Fonte: Adaptado de Ledig et al. (2017)

6.3.2 Arquitetura do modelo

Como já mencionado a SRGAN é composta por duas redes neurais, a rede geradora e discriminadora que têm suas arquiteturas descritas a seguir.

6.3.2.1 Rede geradora

A rede geradora tem sua arquitetura ilustrada na Figura 6.19. A rede é composta por diversas camadas convolucionais que seguem o formato $kAnBsC$ no qual A, B e C representam o tamanho do *kernel*, o número de filtros e tamanho do *stride*.

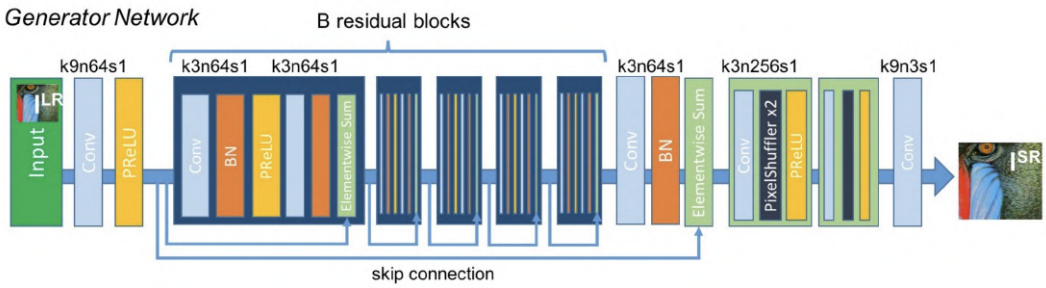

Figura 6.19: Arquitetura da rede geradora.
Fonte: Ledig et al. (2017)

A rede possui como entrada a imagem em baixa resolução I^{LR} na qual é aplicada uma camada convolucional com $A = 9$, $B = 64$ e $C = 1$ seguida de uma função de ativação PReLU, descrita na seção 4.1.

Em sequência os dados são aplicados em uma rede residual, como descrita no capítulo 4.9, formada por $B = 16$ blocos residuais.

Em seguida, uma camada convolucional semelhante à primeira camada da rede é aplicada, seguida por uma camada de *batch normalization*. Esta última é um método que normaliza as ativações dos neurônios da rede, evitando grandes discrepâncias nos valores obtidos e acelerando o treinamento da rede (IOFFE; SZEGEDY, 2015).

Por fim, são aplicados blocos compostos por operações de *pixel shuffle*, que é uma abordagem para aumentar a resolução de um *feature map*, isto é fazer o seu *upsample*, de forma que seja evitado distorções como borramento (SHI et al., 2016). Seguido por uma última camada convolucional que resulta na imagem de saída em resolução ampliada I^{SR}.

6.3.2.2 Rede discriminadora

A rede discriminadora tem sua arquitetura ilustrada na Figura 6.19 e possui a mesma convenção de escrita para as camadas convolucionais, isto é $kAnBsC$.

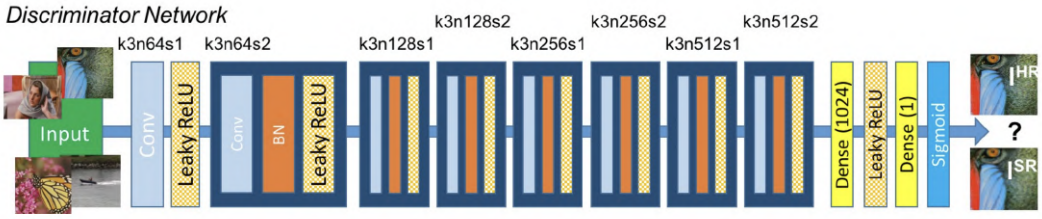

Figura 6.20: Arquitetura da rede discriminadora.
Fonte: Ledig et al. (2017)

A rede possui como entrada imagens que são inicialmente aplicadas em uma camada convolucional com $A = 3$, $B = 64$ e $C = 1$ seguida de uma função de ativação *Leaky* ReLU, descrita na seção 4.1.

Após isso é aplicado uma sequência de blocos com uma uma camada convolucional, seguido de *batch normalization* e *Leaky* ReLU com *strides* alternados entre os valores de 1 e 2.

Ao fim da cadeia de blocos anteriores, os dados seguem um fluxo similar ao descrito no exemplo da seção 4.11.1, isto é, passam por uma camada densa, ou seja totalmente conectada, de 1024 neurônios, em seguida por uma ativação por meio da *Leaky* ReLU, posteriormente por mais uma camada densa com

apenas 1 neurônio. Nesse último neurônio é aplicado a função Sigmoid para mapear o valor final para um número próximo de 0 ou 1, representando uma imagem gerada pela rede geradora ou uma imagem natural verdadeira.

6.3.3 Função de custo: *perceptual loss*

As função de custo utilizada durante o treinamento da SRGAN é nomeada como *perceptual loss*, ou "perda perceptiva", e pode ser descrita pela equação 6-1. Na qual o primeiro termo (l^{SR}_{VGG}) é denominado *content loss*, e o segundo ($10^{-3}l^{SR}_{Gen}$) *adversarial loss*, ou "perda de conteúdo" e "perda adversária" respectivamente.

$$l^{SR} = l^{SR}_{VGG} + 10^{-3}l^{SR}_{Gen} \tag{6-1}$$

6.3.4 Função de custo: *content loss*

A *content loss* l^{SR}_{VGG} é calculada utilizando como base uma rede neural conhecida como VGG (*Visual Geometry Group*). A VGG é uma rede neural profunda que foi desenvolvida originalmente para tarefas de classificação de imagens. Sua arquitetura é ilustrada na Figura 6.21 e é composta por 16 camadas convolucionais seguidas por camadas totalmente conectadas que tem como saída um vetor com o tamanho igual ao número de classes que estão sendo classificadas, no caso igual a 1000 (SIMONYAN; ZISSERMAN, 2015).

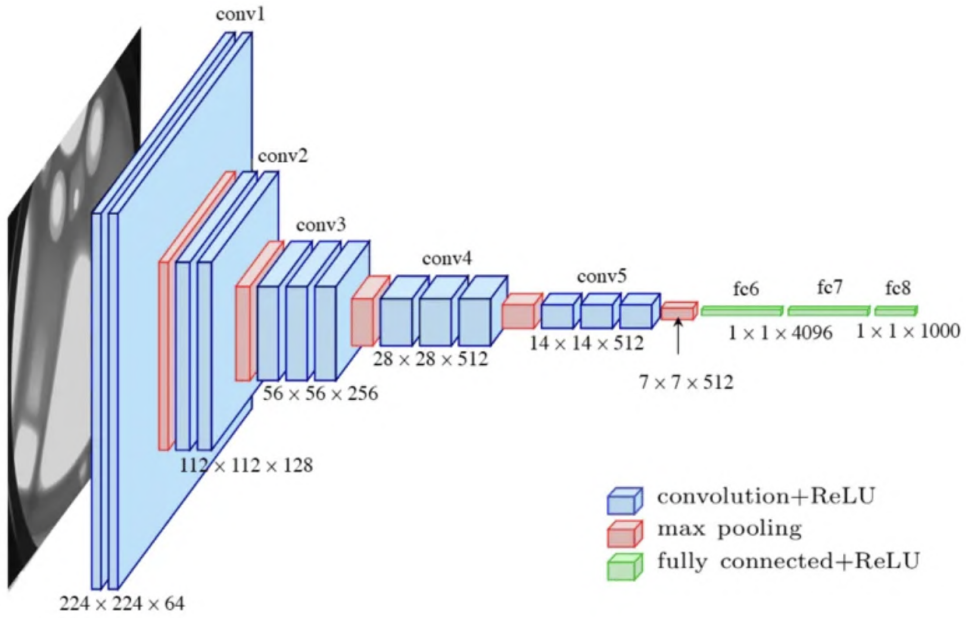

Figura 6.21: Arquitetura da rede VGG16
Fonte: Ferguson et al. (2017)

Para o treinamento da SRGAN é utilizada uma instância pre-treinada da VGG que foi treinada a partir do dataset "Imagenet" (DENG et al., 2009) que possuí 14 milhões de imagens que representam mil classes de objetos diferentes.

Nela são aplicadas as imagens de alta resolução original e a imagem obtida pela rede geradora e são extraídas os *feature maps* obtidos após um certo número de convoluções. Com isso, a *content loss* é descrita na equação 6-2, na qual é calculado a diferença dos quadrados das representações de características extraídas pela VGG entre a imagem gerada pela rede generativa, denotada como $G_{\theta_G}(I^{LR})$, e a imagem de referência em alta resolução, denotada como I^{HR}, sendo $\phi_{i,j}$ o mapa de características obtido pela j-ésima convolução (após a ativação) antes da i-ésima camada de *maxpooling*, descrita na seção 4.11, da rede VGG.

$$l^{SR}_{VGG/i.j} = \frac{1}{W_{i,j}H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2 \tag{6-2}$$

Ao calcular a diferença dos quadrados das representações de características entre a imagem gerada e a imagem de referência, é feito essencialmente uma comparação entre o conteúdo visual das duas imagens em múltiplas escalas e níveis de abstração. Se as representações de características forem semelhantes, isso indica que a imagem gerada preserva o conteúdo visual importante da

imagem de referência.

Esta abordagem é crucial para o treinamento da SRGAN, uma vez que seu objetivo é gerar imagens que não apenas tenham alta resolução, mas também mantenham a estrutura e os detalhes visuais da imagem original. A VGG atua como um "avaliador" que quantifica o quão bem a imagem gerada se assemelha à imagem de referência em termos de conteúdo visual, incentivando assim a geração de imagens de alta qualidade.

6.3.5 Função de custo: *adversarial loss*

A *adversarial loss* l^{SR}_{Gen} é calculada a partir dos resultados obtidos pela rede discriminadora ao avaliar as imagens criadas pela geradora. Mais especificamente, essa função é descrita na Equação 6-3. Na qual, N é o número total de imagens avaliadas e $D_{\theta_D}(G_{\theta_G}(I^{LR}))$ é a probabilidade da imagem reconstruída $G_{\theta_G}(I^{LR})$ ser a imagem em alta resolução natural.

$$l^{SR}_{Gen} = \sum_{n=1}^{N} - \log D_{\theta_D}(G_{\theta_G}(I^{LR})) \tag{6-3}$$

Essa abordagem encoraja a rede a favorecer soluções que estão no âmbito de imagens naturais, tentando enganar a rede discriminadora.

6.3.6 Avaliação do modelo

O modelo é treinado a partir de uma combinação dos *datasets* "Flickr2k" já mencionando anteriormente e conjunto "DIV2K" que possui 1000 imagens com resolução de 2K. A partir dessa coleção é aplicada a metodologia de extração de *patches*, descrita na seção 5.1.1 para expandir o número de dados e reduzir dimensão total e consequentemente o custo computacional durante o processamento. Os patches extraídos possuem uma dimensão de 128×128 pixels, e nesse caso, não é necessário fazer a filtragem de frequência, abordada no capítulo 5.1.2, já que com imagens desse tamanho, não se apresentou figuras que não possuem características que possam ser extraidas pelo modelo durante o treinamento. No total, é utilizado ao redor de 88.000 imagens como conjunto de treinamento, e 13.000 como conjunto de validação.

O treinamento é feito por 150 epochs, com a duração de aproximadamente cinco minutos por *epoch*, totalizando assim doze horas e trinta minutos para a execução. Durante o processo são registrados os valores das funções de custo tanto da rede geradora quanto a discriminadora.

A Figura 6.22 mostra o comportamento da função de custo apresentado pela rede geradora. Nela pode se perceber como o custo, apesar de possuir algumas oscilações, vai diminuindo de forma clara, sem indícios de já ter alcançado um platô no qual não há mais mudanças significativas entre valores obtidos. Dessa forma, é possível afirmar que provavelmente o resultado poderia ser melhorado caso treinado por mais *epochs*.

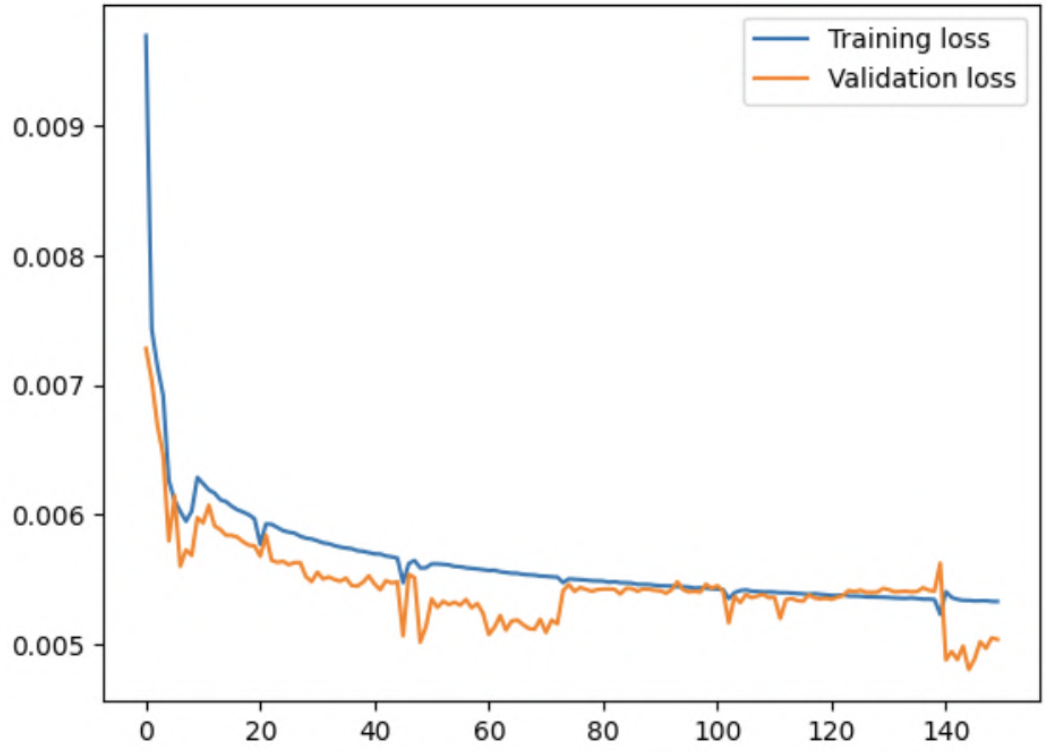

Figura 6.22: Curvas das funções de perda da rede geradora no conjunto de treinamento e validação, em auzl e laranja, respectivamente.
Fonte: Autor

As comparações entre as perdas da rede discriminadora e geradora são destacadas na Figura 6.23. O gráfico revela que a rede geradora, representada em laranja, diminui ao longo do treinamento e, em momentos específicos, como em torno das *epochs* 10, 20 e 100, a rede discriminadora, em azul, falha em distinguir entre imagens geradas e imagens reais. Isso resulta em ajustes na rede discriminadora, que, nas iterações subsequentes, recupera sua capacidade de distinguir corretamente a origem das imagens. Devido aos ajustes na rede discriminadora, a rede geradora, cuja função de custo inclui um componente relacionado à rede discriminadora, também é ajustada. Após um certo número de épocas, a rede geradora volta a enganar a rede discriminadora. Assim, otimizar o custo de uma rede contribui para o ajuste da outra.

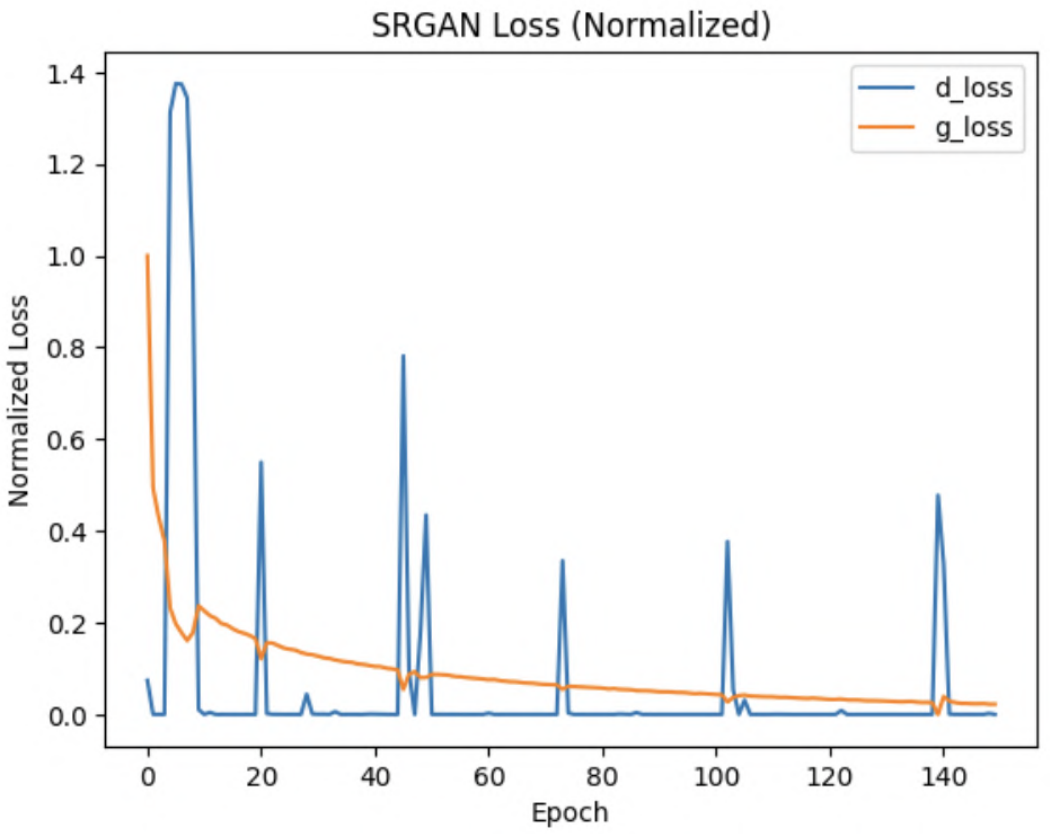

Figura 6.23: Curvas das funções de perda da rede geradora e discriminadora, em auzl e laranja, respectivamente.
Fonte: Autor

6.3.7 Métricas

Para avaliar os resultados da SRGAN, foi utilizado a mesma metodologia que na seção 6.1.3, isto é, são calculados os valores de PSNR e SSIM a partir da aplicação do modelo em imagens do dataset "Set14". Esses resultados são exibidos na Tabela 6.3, e nela é perceptível que os valores obtidos pela SRGAN são melhores que os da interpolação bicúbica. Porém essa diferença é pequena em valores absolutos, mesmo com o resultado visual, ou seja, para a percepção humana apresentado pelo modelo sendo significativamente superior.

Tabela 6.3: Valores de PSNR e SSIM calculados no modelo SRGAN a partir do *dataset* "Set14"

	PSNR	**SSIM**
SRGAN	23.160303	0.672183
Interpolação bicúbica	22.933111	0.659410

A Figura 6.24 mostra a comparação entre a imagem original em alta e baixa resolução, e também aumentadas por meio de interpolação bicúbica e por

atuação do modelo SRGAN. A comparação deixa claro como a performance da rede é substancialmente melhor do que a abordagem por interpolação. A rede consegue aumentar a resolução da imagem sem apresentar aparência pixelada ou borrada. Se mostrando assim como uma excelente abordagem para resolver o problema de super resolução.

Figura 6.24: Na primeira linha, da esquerda para a direita: imagem de alta resolução original, imagem de baixa resolução; Na segunda linha, da esquerda para a direita: imagem obtida por interpolação bicúbica, imagem gerada pelo modelo SRGAN.
Fonte: Autor.

6.4 Experimentos

Devido aos bons resultados obtidos pela modelo SRGAN utilizando a rede VGG como parte da sua função de custo, abordado na seção 6.3, é proposto por este trabalho que essa abordagem pode ser utilizada em outros modelos convolucionais para melhorar seus resultados.

Para testar essa hipótese, a função de custo da rede SRCNN, explicada no capítulo 6.1, é modificada para não ser apenas a MSE entre os valores do pixels, e sim uma parcela de MSE, e outra a diferença entre os feature maps da imagem original e imagem gerada pelo modelo ao serem aplicados na rede VGG. De forma mais clara, a função de custo é descrita na equação 6-4.

$$loss = \beta l_{VGG} + (1 - \beta) l_{MSE} \tag{6-4}$$

Na qual, β é um valor entre 0 e 1, que retrata a porcentagem da função de custo que é atribuído a rede VGG. Além disso, l_{VGG} representa o mesmo que $l^{SR}_{VGG/i.j}$, descrito na equação 6-2 e no capítulo 6.3.4, e l_{MSE} é o MSE entre pixels convencional, como apresentado na seção 5.2.

O modelo é treinado por 150 epochs com essa nova função de perda e apresenta as métricas contidas na Tabela 6.4

Tabela 6.4: Valores de PSNR e SSIM calculados o modelo SRCNN a partir do *dataset* "Set14", com a modificação da função de custo

	PSNR	**SSIM**
f2=5 (VGG)	23.177210	0.647865

Na Tabela 6.5 é exibido a comparação entre os valores apresentados previamente na seção 6.1.3 e os novos valores encontrados a partir do experimento. Dessa forma, nota-se que o resultado comprova a hipótese proposta, já que as métricas calculadas apresentam valores ligeiramente maiores. Logo, essa ideia pode ser explorada com outros valores de β para buscar resultados melhores ainda.

Tabela 6.5: Valores de PSNR e SSIM calculados para diferentes variações do modelo SRCNN a partir do *dataset* "Set14", com a linha em cinza destacando o resultado obtido com a modificação da função de custo

	PSNR	**SSIM**
f2=5 (VGG)	23.177210	0.647865
f2=1	22.824215	0.634771
f2=1 não filtrado	22.736856	0.625142
f2=5	23.128874	0.645081
f2=5 não filtrado	23.005547	0.639881
Interpolação bicúbica	22.933111	0.659410

Apesar dos resultados das métricas serem melhores ao se usar a *loss function* proposta, ao visualizar imagens super resolvidas pelo modelo, a melhora na performance não é óbvia, como é exemplificado na Figura 6.25.

Figura 6.25: Comparação entre imagens geradas pelo modelo utilizando MSE como a função de custo e também com VGG
Fonte: Autor

6.5 Comparação entre modelos

Para fazer uma comparação objetiva entre os modelos estudados e suas variações, os valores das métricas já exibidos pelo projeto são compilados na Tabela 6.6.

Além disso, as Figuras 6.26 e 6.27 a seguir mostram comparações entre imagens geradas pelos modelos. Com base nas imagens presentes nestas figuras e Tabela 6.6 pode-se tirar conclusões interessantes. Primeiramente, o modelo com a melhor performance com base na métrica PSNR, foi o DBPN, descrito na seção 6.2, e com base na SSIM foi o modelo SRGAN. Ao analisar as imagens, é fácil de notar que as imagens mais nítidas e com melhores definições de características como bordas e texturas são as geradas pelo modelo SRGAN. Isso demonstra um conceito importante discutido no capítulo 6.3, isto é, que as métricas comumente usadas dentro do contexto de super resolução, não indicam realmente qual é o modelo que gera os melhores resultados em termos de percepção humana. Na prática, essas métricas favorecem abordagens focadas em valores de pixel e não em qualidade das características presentes nas imagens geradas.

Tabela 6.6: Valores de PSNR e SSIM calculados para os modelos SRGAN, DBPN e as diferentes variações do modelo SRCNN a partir do *dataset* "Set14"

	PSNR	**SSIM**
SRCNN	23.128874	0.645081
SRCNN (VGG)	23.177210	0.647865
DBPN	23.522642	0.662952
SRGAN	23.160303	0.672183
Interpolação bicúbica	22.933111	0.659410

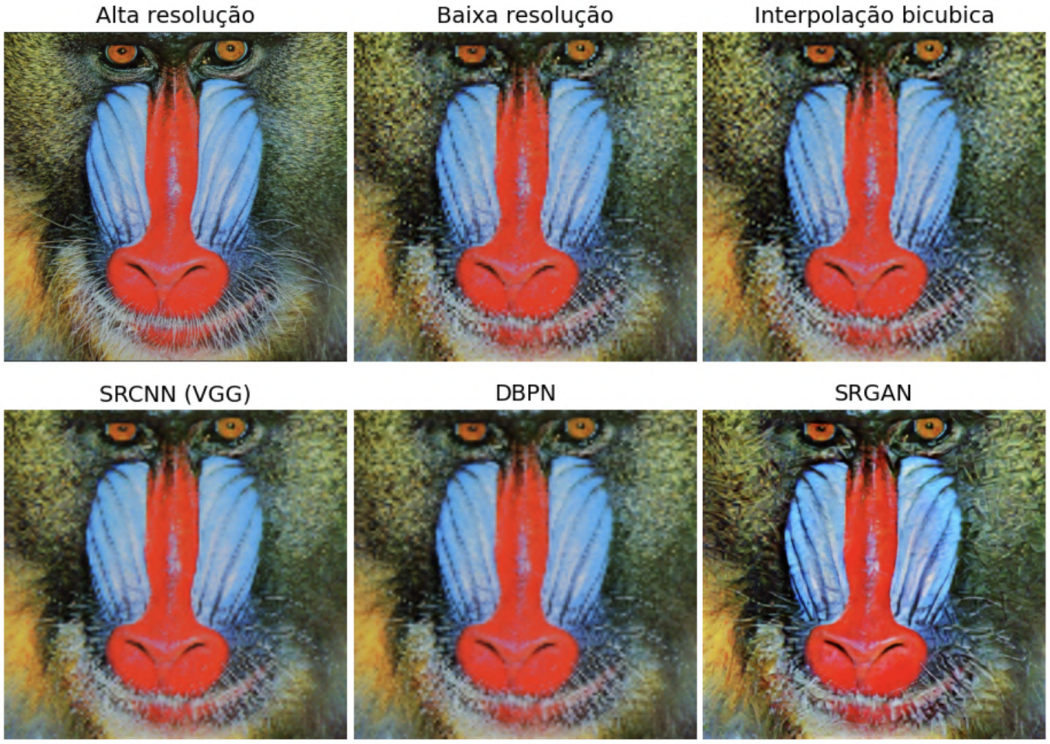

Figura 6.26: Comparação entre imagens geradas pelos três modelos SRCNN, DBPN e SRGAN
Fonte: Autor

Outro ponto importante que também é abordado na seção 6.3 e é bem ilustrado nas imagens presentes na Figuras 6.26 e 6.27 é como a abordagem baseado em GAN produz imagens com possíveis características bem definidas, mesmo que, possivelmente essas características não sejam realistas em relação ao conteúdo original. Isso contrasta com abordagens que buscam minimizar apenas o MSE, como no caso da SRCNN e DBPN, que de certa forma encontra uma solução que é análoga a média das possíveis soluções. Um exemplo desse comportamento é a imagem do babuíno presente na Figura 6.26, que na versão gerada pela SRGAN, tem os seus pelos numa configuração completamente diferente da imagem original.

Adicionalmente, é importante destacar que os modelos DBPN e SRCNN apesar de possuírem valores de métricas consideravelmente diferentes, ao se

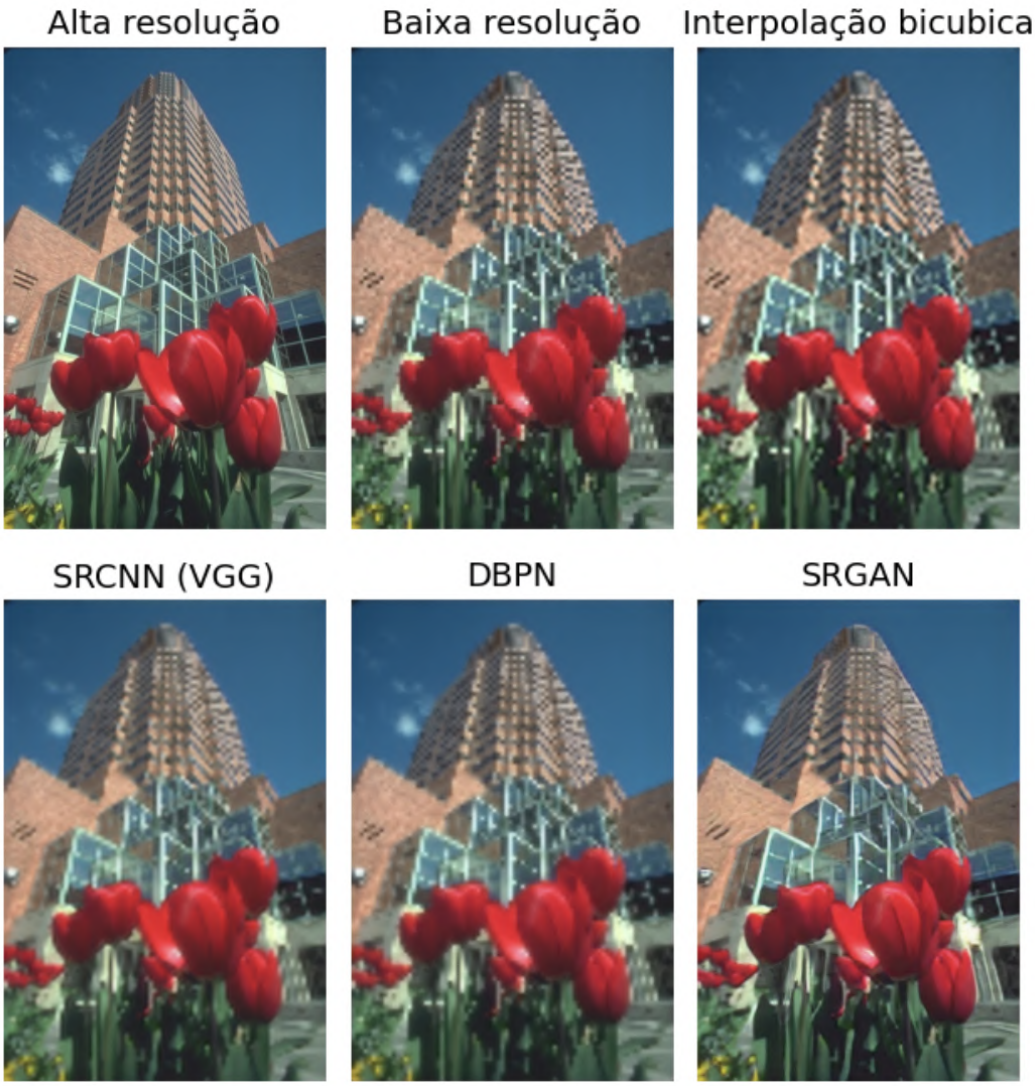

Figura 6.27: Comparação entre imagens geradas pelos três modelos SRCNN, DBPN e SRGAN
Fonte: Autor

analisarem as imagens geradas por esses modelos, a diferença de qualidade das imagens para a percepção humana é muito pequena. Esse comportamento reforça ainda mais o ponto de que modelos baseados em MSE tendem a preservar uma qualidade de borramento das imagens.

7
Aplicação em vídeos

Uma área de aplicação importante para as soluções de Super Resolução é a ampliação da resolução de vídeos. Os métodos aplicados para esse fim são diretamente relacionados com as abordagens para imagens singulares, isso porque um vídeo são apenas uma sequência de quadros, ou do inglês *frames* que nada mais são do que imagens. Portanto, para melhorar a resolução de um vídeo como um todo, basta percorrer cada *frame*, aplicar um método de Super Resolução, seja ele baseado em *deep learning* ou não, para aprimorar a qualidade da imagem e, em seguida, organizar todas essas novas imagens super resolvidas em uma nova sequência.

Nesse trabalho, essa metodologia é aplicada em alguns vídeos para demonstrar os resultados notáveis que podem ser alcançados. Através da aplicação de técnicas de Super Resolução, observa-se uma significativa melhoria na qualidade das imagens e na clareza dos vídeos. Não é necessária fazer uma análise detalhada das melhorias de cada *frame* dos vídeos, já que esse tipo de detalhamento é explorado na seção 6.5.

Dessa forma, para se observar os vídeos gerados é disponibilizado um *link* para uma pasta do *Google Drive* com os vídeos produzidos. O link se encontra a seguir: https://drive.google.com/drive/folders/1oYzF52Zo87IDsACEXThtX-ffdvyt3BdY?usp=sharing.

8
Conclusão

Ao fim deste trabalho foi realizado o estudo, implementação e avaliação de modelos de *deep learning* que buscam resolver o problema da Super Resolução, cada um possuindo arquiteturas, características e nuances diferentes. Dessa forma, foi possível destacar suas limitações, pontos negativos e positivos, além de propor abordagens que podem melhorar os resultados originais.

Para o desenvolvimento do projeto foram estudados diversos conceitos relacionados com o contexto de inteligência artificial e *deep learning*. Temas como os componentes e o funcionamento de redes neurais (MLP) e redes neurais convolucionais (CNN), algoritmos como descida do gradiente e *backpropagation*, métricas de avaliação, foram fundamentais para a implementação, treinamento e análise dos modelos.

Durante a evolução do projeto, alguns obstáculos e dificuldades foram encontrados. Inicialmente, os esforços foram dedicados principalmente para a leitura dos artigos que descrevem os modelos selecionados e, a partir destes, suas implementações e treinamentos. Dessa forma, os estudos das partes teóricas fundamentais não foram dadas suas devidas importâncias. Isso gerou grandes dificuldades na compreensão e execução dos modelos, o que provocou no surgimento de *bugs*, erros durante os treinamentos, resultados ruins e análises equivocadas. As consequências dessa escolha de prioridade entre estudar e implementar, foram atrasos e contratempos em relação aos resultados. Dessa forma, é de se esperar que caso os estudos tivessem sido priorizados desde o início, o desenvolvimento do trabalho teria sido mais fluido.

Além disso, a definição inicial do escopo, isto é, os modelos selecionados, os cronogramas criados foram feitos considerando que não haveria imprevistos ou dificuldades significativas durante a execução do projeto. No entanto, a realidade revelou-se mais complexa, e a ausência de uma margem para acomodar desafios inesperados contribuiu para a ampliação dos contratempos enfrentados. Esta experiência ressalta a necessidade de uma abordagem mais cautelosa na formulação de cronogramas, levando em consideração potenciais contratempos e permitindo ajustes conforme necessário ao longo do desenvolvimento do projeto.

Em relação a oportunidades possíveis para a continuação deste trabalho,

existem diversos caminhos. Este projeto introduz uma análise para modelos de Super Resolução baseados em *deep learning*, porém se limitou a três modelos considerados como marcos da área. Dessa forma, novos modelos podem ser explorados e as tendencias e experimentos aqui implementados têm a capacidade de ser expandidos para buscar resultados melhores.

Referências

AG, A. **Optimization in Deep Learning — medium.com**. 2019. <https://medium.com/ai%C2%B3-theory-practice-business/optimization-in-deep-learning-5a5d263172e>. [Acessado em 5 de Novembro de 2023].

AGUSTSSON, E.; TIMOFTE, R. Ntire 2017 challenge on single image super-resolution: Dataset and study. In: **The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops**. [S.l.: s.n.], 2017.

AREMU, T. **Learning Rate Schedulers — pub.towardsai.net**. 2023. <https://pub.towardsai.net/learning-rate-schedulers-6bd7ae60ed47>. [Accessado em 11 de Novembro de 2023].

BAHETI, P. **Train Test Validation Split: How To & Best Practices [2023] — v7labs.com**. 2021. <https://www.v7labs.com/blog/train-validation-test-set#:~:text=In%20general%2C%20putting%2080%25%20of,dimension%20of%20the%20data%2C%20etc>. [Acessado em 20 de Junho de 2023].

BROWNLEE, J. **How to Configure the Number of Layers and Nodes in a Neural Network - MachineLearningMastery.com — machinelearningmastery.com**. 2019. <https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/>. [Acessado em 12 de Outubro de 2023].

BROWNLEE, J. **Train Neural Networks With Noise to Reduce Overfitting - MachineLearningMastery.com — machinelearningmastery.com**. 2019. <https://machinelearningmastery.com/train-neural-networks-with-noise-to-reduce-overfitting/>. [Acessado em 22 de Outubro de 2023].

CHEN, L.-C. et al. **DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs**. arXiv, 2017. ArXiv:1606.00915 [cs]. Disponível em: <http://arxiv.org/abs/1606.00915>.

CHI-FENG, W. **The Vanishing Gradient Problem — towards-datascience.com**. 2019. <https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484>. [Acessado em 1 de Novembro de 2023].

CONTRIBUTORS, W. **Kernel (image processing) - Wikipedia — en.wikipedia.org**. 2023. <https://en.wikipedia.org/wiki/Kernel_(image_processing)>. [Acessado em 14 de Outubro de 2023].

CONTRIBUTORS, W. **Structural similarity - Wikipedia — en.wikipedia.org**. 2023. <https://en.wikipedia.org/wiki/Structural_similarity>. [Acessado em 14 de Outubro de 2023].

DENG, J. et al. Imagenet: A large-scale hierarchical image database. In: IEEE. **2009 IEEE conference on computer vision and pattern recognition**. [S.l.], 2009. p. 248–255.

DONG, C. et al. **Image Super-Resolution Using Deep Convolutional Networks**. arXiv, 2015. ArXiv:1501.00092 [cs]. Disponível em: <http://arxiv.org/abs/1501.00092>.

DUMOULIN, V.; VISIN, F. **A guide to convolution arithmetic for deep learning**. arXiv, 2018. ArXiv:1603.07285 [cs, stat]. Disponível em: <http://arxiv.org/abs/1603.07285>.

FERGUSON, M. et al. Automatic localization of casting defects with convolutional neural networks. In: **2017 IEEE International Conference on Big Data (Big Data)**. Boston, MA: IEEE, 2017. p. 1726–1735. ISBN 978-1-5386-2715-0. Disponível em: <http://ieeexplore.ieee.org/document/8258115/>.

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In: JMLR WORKSHOP AND CONFERENCE PROCEEDINGS. **Proceedings of the thirteenth international conference on artificial intelligence and statistics**. [S.l.], 2010. p. 249–256.

GLOROT, X.; BORDES, A.; BENGIO, Y. Deep sparse rectifier neural networks. In: GORDON, G.; DUNSON, D.; DUDíK, M. (Ed.). **Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics**. Fort Lauderdale, FL, USA: PMLR, 2011. (Proceedings of Machine Learning Research, v. 15), p. 315–323. Disponível em: <https://proceedings.mlr.press/v15/glorot11a.html>.

GOMES, J.; VELHO, L. **Computação gráfica: imagem**. [S.l.]: IMPA, 2002.

GOODFELLOW, I. J. et al. **Generative Adversarial Networks**. arXiv, 2014. ArXiv:1406.2661 [cs, stat]. Disponível em: <http://arxiv.org/abs/1406.2661>.

HARIS, M.; SHAKHNAROVICH, G.; UKITA, N. **Deep Back-Projection Networks For Super-Resolution**. arXiv, 2018. ArXiv:1803.02735 [cs]. Disponível em: <http://arxiv.org/abs/1803.02735>.

HARIS, M.; SHAKHNAROVICH, G.; UKITA, N. **Recurrent Back-Projection Network for Video Super-Resolution**. arXiv, 2019. ArXiv:1903.10128 [cs]. Disponível em: <http://arxiv.org/abs/1903.10128>.

HE, K. et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: **Proceedings of the IEEE international conference on computer vision**. [S.l.: s.n.], 2015. p. 1026–1034.

IOFFE, S.; SZEGEDY, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv, mar. 2015. ArXiv:1502.03167 [cs]. Disponível em: <http://arxiv.org/abs/1502.03167>.

KUMAR, S. K. **On weight initialization in deep neural networks**. arXiv, 2017. ArXiv:1704.08863 [cs]. Disponível em: <http://arxiv.org/abs/1704.08863>.

LEDIG, C. et al. **Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network**. arXiv, 2017. ArXiv:1609.04802 [cs, stat]. Disponível em: <http://arxiv.org/abs/1609.04802>.

LI, Z. et al. **Feedback Network for Image Super-Resolution**. arXiv, 2019. ArXiv:1903.09814 [cs]. Disponível em: <http://arxiv.org/abs/1903.09814>.

LIM, B. et al. Enhanced deep residual networks for single image super-resolution. In: **The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops**. [S.l.: s.n.], 2017.

NIELSEN, M. A. misc, **Neural Networks and Deep Learning**. Determination Press, 2018. Disponível em: <http://neuralnetworksanddeeplearning.com/>.

NILSSON, J.; AKENINE-MöLLER, T. **Understanding SSIM**. arXiv, 2020. ArXiv:2006.13846 [cs, eess]. Disponível em: <http://arxiv.org/abs/2006.13846>.

OORD, A. v. d. et al. **WaveNet: A Generative Model for Raw Audio**. arXiv, 2016. ArXiv:1609.03499 [cs]. Disponível em: <http://arxiv.org/abs/1609.03499>.

O'SHEA, K.; NASH, R. **An Introduction to Convolutional Neural Networks**. arXiv, 2015. ArXiv:1511.08458 [cs]. Disponível em: <http://arxiv.org/abs/1511.08458>.

ROBINSON, M. D. et al. New applications of super-resolution in medical imaging. **Super-resolution imaging**, CRC Press Boca Raton, Florida, v. 2010, p. 384–412, 2010.

SANDERSON, G. **Backpropagation calculus | Chapter 4, Deep learning — youtube.com**. 2017. <https://www.youtube.com/watch?v=tIeHLnjs5U8>. [Acessado em 20 de Junho de 2023].

SANDS, J. **Can I use a metric as a loss function? — jonathan-sands.com**. 2021. <https://jonathan-sands.com/metric/loss/2021/05/13/Metric-vs-Loss.html>. [Acessado em 5 de Novembro de 2023].

SEAN, S. **An Implementation of Sobel Edge Detection - Rhea — projectrhea.org**. 2023. <https://www.projectrhea.org/rhea/index.php/An_Implementation_of_Sobel_Edge_Detection>. [Accessed 13-11-2023].

SHARMA, S. **Activation Functions in Neural Networks — towardsdatascience.com**. 2017. <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>. [Acessado em 17 de Outubro de 2023].

SHI, W. et al. **Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network**. arXiv, 2016. ArXiv:1609.05158 [cs, stat]. Disponível em: <http://arxiv.org/abs/1609.05158>.

SIMONYAN, K.; ZISSERMAN, A. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. arXiv, 2015. ArXiv:1409.1556 [cs]. Disponível em: <http://arxiv.org/abs/1409.1556>.

STARMER, J. **Neural Networks Part 8: Image Classification with Convolutional Neural Networks (CNNs) — youtube.com**. 2021. <https://www.youtube.com/watch?v=HGwBXDKFk9I>. [Acessado em 15 de Outubro de 2023].

STEVENS, A. et al. The potential for Bayesian compressive sensing to significantly reduce electron dose in high-resolution STEM images. **Microscopy**, v. 63, n. 1, p. 41–51, fev. 2014. ISSN 2050-5698, 2050-5701. Disponível em: <https://academic.oup.com/jmicro/article-lookup/doi/10.1093/jmicro/dft042>.

TIMOTHY. **Image Convolution Filtering — timothy_terati**. 2021. <https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b>. [Acessado em 14 de Outubro de 2023].

VILLENA, S. et al. Image super-resolution for outdoor digital forensics. usability and legal aspects. **Computers in industry**, Elsevier, v. 98, p. 34–47, 2018.

WANG, X. et al. **ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks**. arXiv, 2018. ArXiv:1809.00219 [cs]. Disponível em: <http://arxiv.org/abs/1809.00219>.

WANG, Z.; CHEN, J.; HOI, S. C. H. **Deep Learning for Image Super-resolution: A Survey**. arXiv, 2020. ArXiv:1902.06068 [cs]. Disponível em: <http://arxiv.org/abs/1902.06068>.

WU, J. Introduction to Convolutional Neural Networks. 2017.

ZEYDE, R.; ELAD, M.; PROTTER, M. On single image scale-up using sparse-representations. In: SPRINGER. **Curves and Surfaces: 7th International Conference, Avignon, France, June 24-30, 2010, Revised Selected Papers 7**. [S.l.], 2012. p. 711–730.

ZHANG, L. et al. A super-resolution reconstruction algorithm for surveillance images. **Signal Processing**, Elsevier, v. 90, n. 3, p. 848–859, 2010.