



UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

Transfer Learning para classificação e detecção de imagens com ImageNet

Rodrigo Malta Esteves

Orientador: Prof. Carlos Tadeu Pagani Zanini

Rio de Janeiro - RJ, Brasil

Junho de 2022

Transfer Learning para classificação e detecção de imagens com ImageNet

RODRIGO MALTA ESTEVES

Trabalho Final de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a conclusão do curso de pós-graduação.

Orientador: Carlos Tadeu Pagani Zanini

Rio de Janeiro, Setembro de 2022

Transfer Learning para classificação e detecção de imagens com ImageNet

Rodrigo Malta Esteves

Orientador: Carlos Tadeu Pagani Zanini

Trabalho Final de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a conclusão do curso de pós-graduação.

Prof.Dr. Carlos Tadeu Pagani Zanini
IM-UFRJ

Prof.Dr.
IM-UFRJ

Prof.Dr.
IM-UFRJ

Rio de Janeiro, Setembro de 2022

Resumo

Transfer Learning para classificação e detecção de imagens com ImageNet

Rodrigo Malta Esteves

Orientador: Carlos Tadeu Pagani Zanini

Neste trabalho são introduzidos os conceitos necessários para entender o funcionamento de redes neurais MLP (camadas de perceptrons, funções de ativação, método de treinamento e otimizadores) e redes neurais convolucionais (camadas convolucionais, filtros, strides, padding e camadas de pooling). São introduzidas diferentes arquiteturas popularmente utilizadas em tarefas de classificação que envolvem transfer learning e, em seguida, são comparados modelos para classificação de imagens. Finalmente, são introduzidos alguns dos fundamentos da detecção de objetos e é realizada uma aplicação em imagens e vídeos.

Para o desenvolvimento do trabalho serão utilizados como referência os modelos de Aprendizado Profundo pré-treinados no ImageNet e MSCOCO com Keras 2.3.0 e TensorFlow 2.0. Também foi utilizado o serviço do Google Colaboratory para suprir a falta de capacidade computacional.

Palavras chave: Classificação de imagens, Detecção de objetos, Redes Neurais Convolucionais, Transfer Learning, Xception, Yolo.

Sumário

1	Introdução e objetivos	p. 4
2	Redes Neurais	p. 6
2.1	Perceptron	p. 6
2.2	Single Layer Perceptrons	p. 7
2.3	Multilayer Perceptron	p. 8
2.4	Funções de ativação	p. 9
2.4.1	Step	p. 10
2.4.2	Sigmoid	p. 11
2.4.3	Tangente Hiperbólica	p. 11
2.4.4	ReLU	p. 11
2.5	Treinamento de redes neurais via método dos gradientes	p. 12
2.5.1	Backpropagation	p. 12
2.5.2	Método de gradiente com momentos	p. 14
2.5.3	Adagrad	p. 15
2.5.4	RMSProp	p. 16
2.5.5	Adam	p. 17
2.6	MLP para regressão e classificação	p. 17
2.7	Redes neurais convolucionais	p. 18
2.7.1	Camadas convolucionais	p. 19
2.7.2	Filtros	p. 20
2.7.3	Stride e padding	p. 22

2.7.4	Camadas de Pooling	p. 24
3	Classificação em imagens	p. 26
3.1	Arquiteturas	p. 26
3.2	Transfer Learning	p. 32
3.3	Aplicação	p. 34
4	Deteção em imagens	p. 40
4.1	Estrutura	p. 41
4.2	Redes totalmente convolucionais	p. 44
4.3	Caixas de ancoragem	p. 44
4.4	You Only Look Once (YOLO)	p. 46
4.5	Aplicação	p. 49
5	Conclusão e trabalhos futuros	p. 57
	Lista de Figuras	p. 58
	Lista de Tabelas	p. 61
	Referências	p. 62

1 Introdução e objetivos

O custo computacional elevado e o tempo necessário para desenvolver um modelo de redes neurais são problemas comuns, principalmente quando realizamos tarefas de visão computacional, como classificação de imagens e detecção de objetos, devido ao grande número de camadas necessário para tornar a rede neural eficiente.

Uma abordagem possível para resolver esse problema é reutilizar, no novo modelo que está sendo desenvolvido, os pesos das camadas intermediárias pré-treinadas de um modelo que performe uma tarefa similar ao do novo modelo. Esse método transferirá a parte mais geral do conhecimento do modelo já treinado, e servirá como ponto de partida para o treinamento do novo modelo. Essa abordagem é conhecida como *transfer learning*, ou transferência de aprendizado, e é amplamente utilizada no treinamento de redes neurais. No contexto de visão computacional, algumas bases de dados populares para treinamento são o *ImageNet*[1], o *MSCOCO*[2], *MNIST*[3] e o *CIFAR-10*[4].

Neste trabalho serão propostos modelos para classificação de imagens de animais, utilizando pesos pré-treinados no *ImageNet*, e detecção de animais em imagens e vídeos, utilizando pesos pré-treinados no *MSCOCO*. Os modelos serão avaliados e comparados utilizando diferentes critérios. Algumas imagens de classes que compõe o conjunto de treinamento foram obtidas pelo Google Imagens e classificadas manualmente, e outras foram obtidas através do *Open Images Dataset*.

O objetivo desse trabalho é utilizar algoritmos de Transferência de Aprendizado para classificar múltiplos conjuntos de dados e comparar os resultados com a forma de aprendizado mais tradicional. Em uma etapa final, um modelo é treinado para detecção de objetos em imagens e vídeos. Em um primeiro momento serão introduzidos os conceitos gerais de redes neurais densas, redes neurais convolucionais e transferência de aprendizado. Em seguida, aplicamos essas definições em um contexto de classificação de imagens. Após essa aplicação são introduzidos resumidamente conceitos que compõe a metodologia de detecção de imagens. Finalmente, aplicamos os conceitos apresentados em um caso

real.

Esta introdução compõe o **Capítulo 1**. O **Capítulo 2** trata, ainda que de forma resumida, da composição e metodologia por trás das redes neurais, começando com uma explicação sobre os diferentes tipos de perceptrons, seguindo por uma apresentação das funções de ativação mais comuns, método dos gradientes, regressão e classificação. Por fim, são apresentados os conceitos que compõe as redes neurais convolucionais. O **Capítulo 3** introduz algumas das arquiteturas de redes convolucionais importantes e o conceito transferência de aprendizado. Em seguida, são realizadas aplicações envolvendo classificação de imagens e alguns modelo são comparados. No **Capítulo 4** algumas ideias relacionadas ao funcionamento dos algoritmos de detecção de objetos são apresentadas, e uma aplicação é feita para imagens e vídeos de animais. Finalmente, no **Capítulo 5**, uma conclusão e algumas sugestões para trabalhos futuros são apresentadas.

2 Redes Neurais

2.1 Perceptron

A unidade mais fundamental de uma rede neural profunda é chamada de neurônio artificial ou *perceptron*, que recebe um input, processa esse input, faz com que ele passe por uma função de ativação, e retorna um output. Os *Perceptrons* se baseiam, em sua forma mais geral, em um neurônio artificial chamado *threshold logic unit* (TLU). O TLU (Figura 1) calcula a soma ponderadas dos inputs e, então, aplica-se uma *função degrau* à soma, sendo $step : \mathbb{R} \rightarrow \{0, 1\}$, $step(x) = \begin{cases} 0, & \text{se } x < 0 \\ 1, & \text{se } x \geq 1 \end{cases}$, retornando um output binário como resultado.

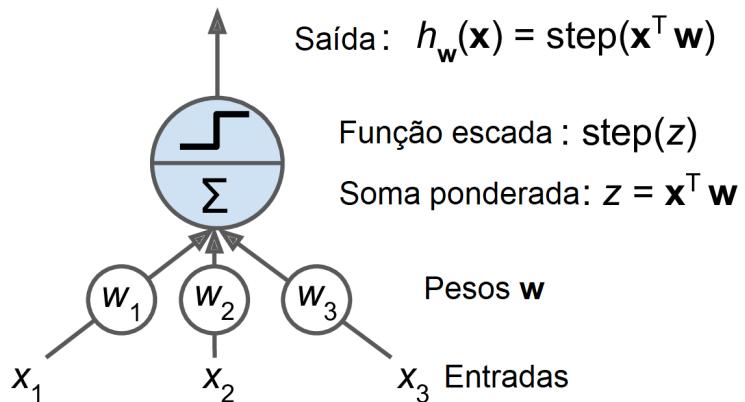


Figura 1: Threshold Logic Unit - Fonte: [5].

A soma ponderada dos inputs é dada por $z = w_1x_1 + \dots + w_nx_n = \mathbf{x}^T\mathbf{w}$, onde x_i são entradas do vetor de inputs e w_i são os pesos e a *step function* é dada por $h_W(\mathbf{x}) = \text{step}(z)$. Os elementos do vetor $\mathbf{x} = (x_1, \dots, x_n)$ também são chamados de covariáveis ou features. O exemplo mais comum de *função degrau* é apresentado pela Equação 3.1. Outra função comum utilizada é a *função sinal*.

$$z_i = \begin{cases} 1, & \text{se } w_i x_i + b \geq 0 \\ 0, & \text{se } w_i x_i + b < 0 \end{cases} \quad (2.1)$$

Um TLU pode ser usado para classificação binária simples. É computada uma combinação linear dos inputs e, caso o resultado exceda um limiar, retorna como output a classe positiva. Caso contrário, retorna a classe negativa.

2.2 Single Layer Perceptrons

O modelo Perceptron de uma camada é composto de uma única camada de perceptrons, onde cada perceptron está conectado à todos os inputs. Quando todos os neurônios de uma camada estão conectados à todos os neurônios na camada anterior, a camada é chamada de *camada densa* (ou *dense layer*). Todos os neurônios que recebem as entradas da rede formam a *camada de entrada* (ou *input layer*). Além disso, um neurônio de viés que retorna sempre 1 é adicionado. Um exemplo desse modelo Perceptron é apresentado na Figura 2.

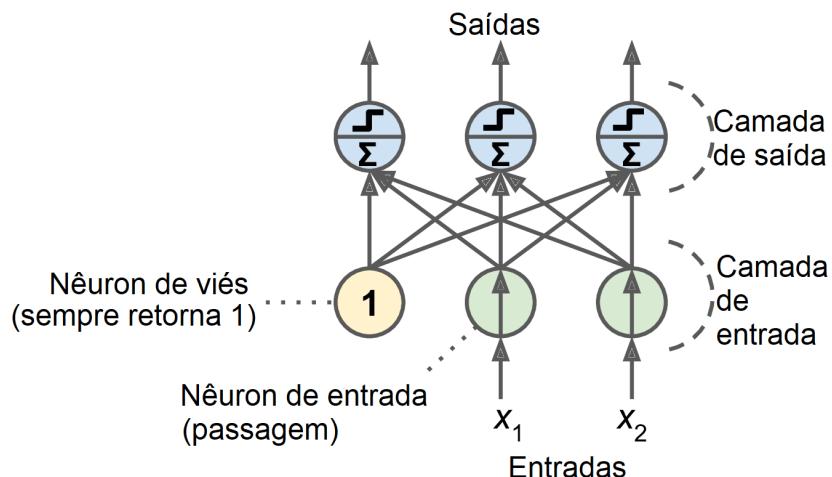


Figura 2: Perceptron com 2 neurônios de input, 1 neurônio de viés, e 3 neurônios de output - Fonte: [5]

As saídas de uma *camada densa* podem ser obtidas através da Equação 3.2. Como anteriormente, nessa equação o X representa a matriz das entradas, o W é a matriz dos pesos, o b é o vetor de viés, e a função $\phi()$ é chamada de *função de ativação* (funções de ativação serão vistas em um capítulo futuro). Neste caso, a função de ativação é uma função de degrau.

$$h_{W,b}(X) = \phi(XW + b) \quad (2.2)$$

O treinamento dos Perceptrons é realizado usando uma variante da Regra de Hebb (Hebb, D.O. (1949)) que leva em consideração o erro feito pela rede quando realiza uma predição. A regra de aprendizado do Perceptron reforça conexões que ajudem a reduzir o erro de previsão. Uma instância de treino é introduzida por vez e, para cada instância, é realizada uma predição. Para cada neurônio que retorna uma previsão errada, há um reforço nos pesos dos inputs que teriam ajudado a realizar a previsão correta. A regra de atualização de pesos do Perceptron é apresentada pela Equação 3.3.

$$w_{i,j}(\text{próximo passo}) = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \quad (2.3)$$

Nesta equação $w_{i,j}$ é o peso entre o i -ésimo neurônio de entrada e o j -ésimo neurônio de saída, o x_i é o i -ésimo valor de entrada da instância de treinamento atual, o \hat{y}_j é a saída do j -ésimo neurônio de saída da instância de treinamento atual, o y_j é a saída alvo do j -ésimo neurônio de saída da instância de treinamento atual, e o η é a taxa de aprendizado.

2.3 Multilayer Perceptron

Diferente do perceptron onde existe apenas uma camada de entrada e uma de saída, o Multilayer Perceptron (MLP) possui uma camada de entrada, uma ou mais camadas de perceptrons, chamadas de *camadas ocultas* (ou *hidden layers*), e uma camada final de saída (Figura 3).

A camada de entrada recebe as features que serão processadas. A primeira *camada oculta* recebe as features da camada de entrada. As outras camadas ocultas recebem como entrada a saída de cada perceptron da camada anterior. A tarefa de previsão e classificação é realizada pela camada de output, que recebe os outputs da última camada oculta. As MLPs são capazes de aproximar funções contínuas, e não apenas lineares como o perceptron simples.

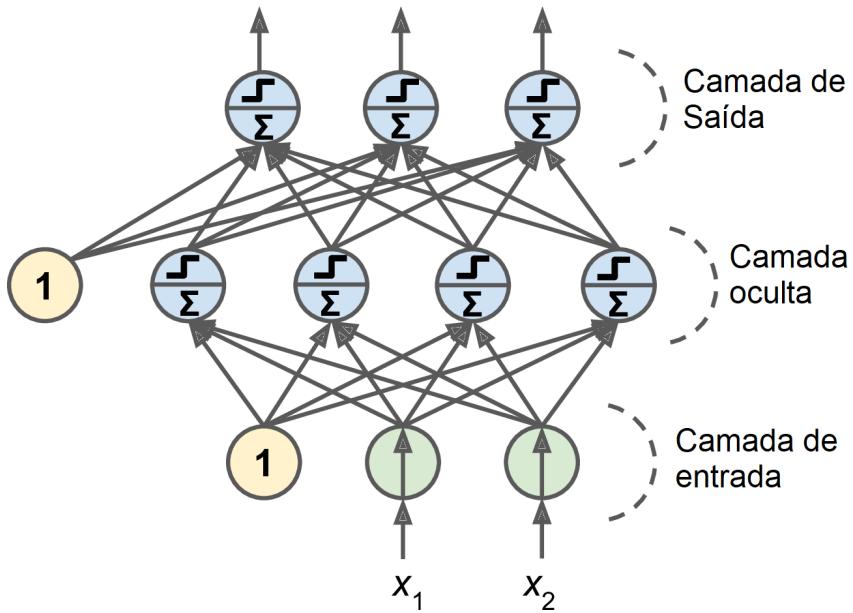


Figura 3: Multilayer Perceptron - Fonte: [5]

O MLP é um exemplo de algoritmo de *feedforward*, onde o sinal flui em apenas uma direção (da entrada até a saída) e as entradas são combinadas com os pesos iniciais em um soma ponderada e sujeitos à função de ativação, assim como o perceptron. O que torna esse caso diferente é que em cada combinação linear é propagada para a próxima camada. Isso ocorre em todas as camadas ocultas até a camada de saída.

Para que realmente haja aprendizado, não basta que o algoritmo propague o resultado até a última camada e pare nela. Caso isso acontecesse, não seria possível aprender os pesos que minimizam a função custo. Uma maneira de ajustar esses pesos de forma iterativa é utilizar o algoritmo de *backpropagation* via *método dos gradientes*.

2.4 Funções de ativação

Caso a rede possua apenas transformações lineares, a saída da rede será uma transformação linear dos valores de entrada. Se não tivermos alguma não-linearidade entre as camadas, então mesmo uma camada profunda será equivalente a uma camada única e não seremos capazes de resolver problemas complexos. Já uma rede neural profunda com ativações não-lineares é capaz de aproximar qualquer função contínua arbitrariamente bem para arquiteturas (número de camadas e neurônios por camadas) suficientemente complexas (Hornik, et al. 1989, Mhaskar, et al. 2017). Na Figura 4 é possível ver os diferentes comportamentos das funções que serão apresentadas, assim como as suas derivadas.

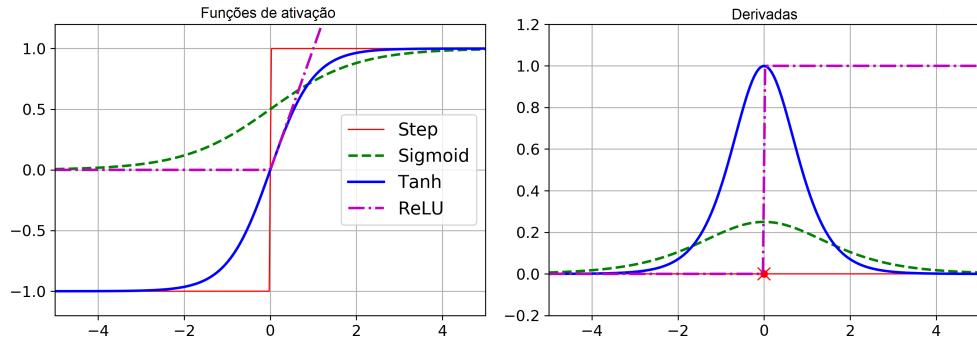


Figura 4: Funções de ativação - Fonte: [5]

2.4.1 Step

Já apresentada anteriormente, é a função de ativação utilizada na forma mais simples de neurônio. É linear e utilizada para classificação binária simples. Recebe como entrada qualquer valor real e retorna os valores 0 ou 1, caso o valor exceda um certo limiar. Um outro nome para essa função é *função de heaviside*.

$$step : \mathbb{R} \rightarrow \{0, 1\}$$

$$step(z) = \begin{cases} 0, & \text{se } z < 0 \\ 1, & \text{se } z \geq 1. \end{cases}$$

Um outro tipo de função degrau é a *função sinal*. Assim como a versão anterior, recebe qualquer valor real. Entretanto, neste caso, a função pode retornar os valores -1, 0 ou 1.

$$sgn : \mathbb{R} \rightarrow \{-1, 0, 1\}$$

$$sgn(z) = \begin{cases} -1, & \text{se } z < 0 \\ 0, & \text{se } z = 0 \\ 1, & \text{se } z > 1. \end{cases}$$

Como essas funções têm derivada nula para todos os pontos diferentes de 0, o treinamento da rede pelo método dos gradientes não é viável. Por esse motivo, uma mudança fundamental na arquitetura do MLP é a substituição das funções step e sign por funções que possuam derivadas bem definidas em todos os pontos, permitindo que o gradiente descendente progrida a cada passo.

2.4.2 Sigmoid

A função de ativação sigmoid também é chamada de função logística. É a mesma função utilizada nos modelos de classificação com regressão logística. Ela recebe qualquer valor real como entrada e retorna valores entre 0 e 1. Valores de entrada elevados retornarão valores mais próximos de 1, e valores de entrada negativos de alta magnitude retornarão valores mais próximos de 0. Essa função é contínua, diferenciável, e apresenta a seguinte forma:

$$\begin{aligned}\sigma : \mathbb{R} &\rightarrow \{0, 1\} \\ \sigma(z) &= \frac{1}{1+e^{-z}}.\end{aligned}$$

2.4.3 Tangente Hiperbólica

Assim como a função logística, essa função de ativação é contínua e diferenciável, mas retorna valores entre -1 e 1 (ao invés de 0 e 1). Isso faz com que o output de cada camada seja aproximadamente centrado em torno do 0 no começo da etapa de treinamento, o que em geral ajuda a acelerar a convergência. A tangente hiperbólica tem a forma:

$$\begin{aligned}\tanh : \mathbb{R} &\rightarrow \{-1, 1\} \\ \tanh(z) &= 2\sigma(2z) - 1.\end{aligned}$$

2.4.4 ReLU

A função *rectified linear unit*, ou *ReLU* (Nair and Hinton 2010), é contínua, porém não diferenciável em $z = 0$ (a curva muda abruptamente, fazendo um "v"), e sua derivada vale 0 para $z < 0$ e 1 para $z > 0$. Na prática, a função ReLU tem a vantagem de ser computacionalmente eficiente, e por isso se tornou bastante popular. Destaca-se ainda o fato de que essa função não possui máximo, o que ajuda a reduzir problemas durante o gradiente descendente, segundo Nair e Hinton.

$$\begin{aligned}\text{ReLU} : \mathbb{R} &\rightarrow \{0, \infty\} \\ \text{ReLU}(z) &= \max(0, z)\end{aligned}$$

2.5 Treinamento de redes neurais via método dos gradientes

2.5.1 Backpropagation

Backpropagation ([6]) é o algoritmo mais utilizado para treinar redes neurais. Primeiro, calcula-se os gradientes da função custo com respeito a todos os parâmetros do modelo (pesos e vieses), então realiza-se uma iteração do algoritmo do gradiente descendente para otimização da função custo da rede neural. Esse “passo para trás”, devido à regra da cadeia empregada no cálculo das derivadas com respeito aos parâmetros, é tipicamente realizado milhares ou milhões de vezes, usando vários lotes de treinamento (subconjuntos das observações que compõem a base de dados), até os parâmetros de modelo convergirem para valores que minimizem a função custo localmente.

De forma resumida, o algoritmo backpropagation nada mais é do que o gradiente descendente usando uma técnica eficiente para computar os gradientes de forma automática (ver adiante). Passando pela rede duas vezes, uma para frente (*forward*) e uma para trás (*backward*), o algoritmo computa o gradiente do erro da rede com respeito a cada parâmetro do modelo. Com isso, obtém-se a direção ótima segundo a qual os pesos e os vieses devem ser manipulados para que os erros de ajuste sejam reduzidos. Com os gradientes calculados, o algoritmo realiza uma iteração do gradiente descendente, e o processo se repete até a rede convergir para um ótimo local.

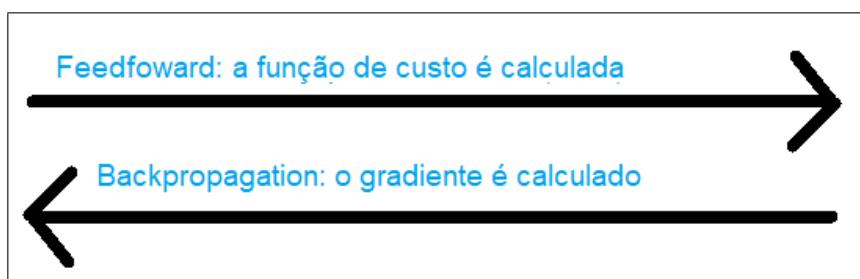


Figura 5: Etapas do Método dos Gradientes

Para computar os gradientes, o backpropagation usa diferenciação automática em modo reverso (*reverse-mode autodiff*), que realiza um passo *forward* computando cada valor dos perceptrons para o lote de treinamento da iteração corrente, e então realiza um passo reverso, computando todos os gradientes de uma vez. O cálculo do gradiente usa de forma recursiva a regra da cadeia, onde as derivadas das camadas superiores são

utilizadas para calcular as derivadas das camadas inferiores de maneira eficiente.

A seguir são apresentados os cálculos dos gradientes envolvidos no algoritmo back-propagation. Nas equações (3.4), (3.5) e (3.6):

- h_k^l representa o output do k -ésimo perceptron da camada l ;
- W_{ij}^l representa o peso ligado ao j -ésimo perceptron da camada $l - 1$ ao i -ésimo perceptron da camada l ;
- b_k^l representa o viés do k -ésimo perceptron da camada l ;
- $a_m^{l+1} = \sum_{k=1}^{N_l} h_k^l W_{km}^{l+1} + b_m^{l+1}$ é o input linear combinado para a camada l ;
- N_l representa o número de perceptrons na camada l .

A equação (3.4) representa a derivada da função custo C com respeito ao output h_k^l , a equação (3.5) representa a derivada da função custo C com respeito ao viés b_k^l e a equação (3.6) representa a derivada da função custo C com respeito aos pesos W_{pk}^l . O índice k representa o k -ésimo perceptron da camada e o índice p representa .

$$\frac{\partial C}{\partial h_k^l} = \sum_{m=1}^{N_{l+1}} \frac{\partial C}{\partial b_m^{l+1}} W_{km}^{l+1} \quad (2.4)$$

$$\frac{\partial C}{\partial b_k^l} = \frac{\partial C}{\partial h_k^l} \sigma'_l(a_k^l) \quad (2.5)$$

$$\frac{\partial C}{\partial W_{pk}^l} = \frac{\partial C}{\partial b_k^l} h_p^{l-1} \quad (2.6)$$

O termo $h_k^l = \sigma_l(a_k^l)$ é o termo de ativação não-linear. Aqui h é o output da unidade, a é o input combinado e σ é a função de ativação. O sub-índice l na função de ativação diz respeito a camada l , uma vez que podemos ter diferentes funções de ativação para cada camada.

Por fim, a verossimilhança, ou output, é dada por $C_l = \log(p(y_i|h_i^l, W^{L+1}, b^{L+1}))$, ou seja, o log da verossimilhança de y dado o último h , os pesos W e viés b da última camada $l + 1$.

Uma vez calculadas as derivadas pelo método de backpropagation, as derivadas são usadas nas iterações do método dos gradientes:

$$\begin{cases} w_{i,j}^l \leftarrow w_{i,j}^l - \gamma \frac{\partial C(y;x,\mathbf{W},\mathbf{b})}{\partial w_{i,j}^l} \\ b_i^l \leftarrow b_i^l - \gamma \frac{\partial C(y;x,\mathbf{W},\mathbf{b})}{\partial b_i^l} \end{cases}$$

onde $\mathbf{W} = (W_{i,j}^l)$ e $\mathbf{b} = (b_i^l)$ para todas as camadas l e pesos ligando unidade i na camada $l-1$ a unidade j na camada l .

Apesar da utilização do método dos gradientes ser uma estratégia de otimização muito popular, ele pode ser extremamente lento em alguns casos. Outros métodos, modificados a partir do método dos gradientes, são mais propensos a escapar de mínimos locais, dentre os quais destacamos os métodos de gradientes com momentos, RMSProp, AdaGrad e Adam nas seções a seguir.

2.5.2 Método de gradiente com momentos

A inclusão de momentos no método de gradientes é projetada para acelerar o aprendizado, especialmente em casos de superfícies muito *ingrimes* (com curvas de nível mais estreitas) em que os gradientes de iterações consecutivas tendem a apontar para direções aproximadamente opostas (ver Figura 6). O algoritmo dos momentos acumula uma média móvel que decai de forma exponencial (*me faltou vocabulário aqui também para "exponentially decaying moving average"*) e continua a se mover em sua direção.

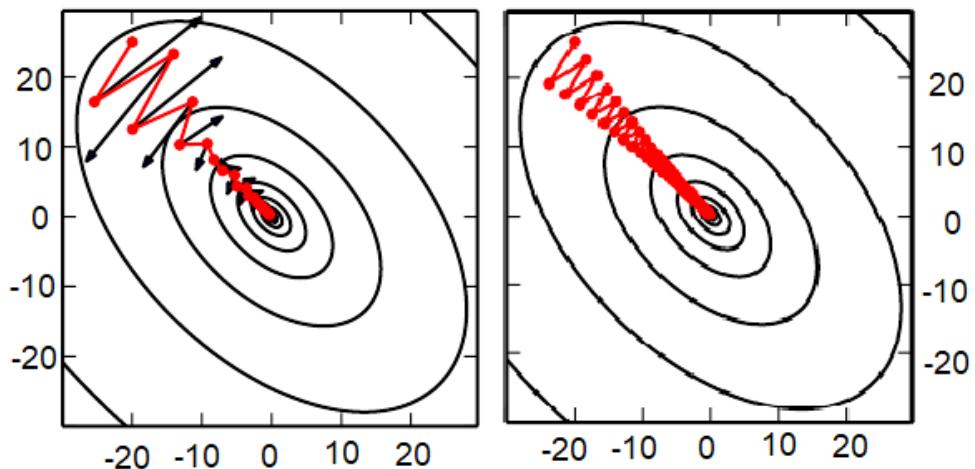


Figura 6: Gradiente descendente utilizando o método dos momentos x Gradiente descendente regular - Fonte: [7]

O lado esquerdo da Figura 6 ilustra como o momento ajuda na convergência. A cada iteração é desenhada uma seta indicando o passo que o gradiente descendente daria

nesse ponto. As linhas de contorno retratam a função de perda. A linha vermelha indica a trajetória percorrida pelo algoritmo no espaço dos parâmetros, combinando a direção dos momentos com a direção do gradiente. Pode-se ver que a utilização dos momentos diminui a tendência de idas e voltas na trajetória do método. O lado direito mostra como o gradiente falha em explorar as informações sobre a curvatura contidas na matriz Hessiana (matriz das derivadas parciais) quando não há momento envolvido.

A otimização pelo método dos momentos leva fortemente em consideração os gradientes anteriores. A cada iteração, ele subtrai o gradiente local do *vetor de momentos* \mathbf{m} , multiplicado por uma taxa de aprendizado η , e atualiza os pesos adicionando esse vetor de momentos. Para prevenir que o momento cresça muito rápido, introduz-se um parâmetro β , que varia entre 0 e 1, fazendo um papel análogo à força de atrito em aplicações da física. Além disso, trataremos $\boldsymbol{\theta}$ como a matriz dos pesos e C a função de custo. Portanto, o algoritmo do gradiente descendente via método dos momentos é definido pelas equações de atualização:

$$\begin{aligned}\mathbf{m} &\leftarrow \beta\mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \mathbf{m}.\end{aligned}$$

2.5.3 Adagrad

O gradiente adaptativo ou *AdaGrad* [8] recebe esse nome porque determina uma taxa de aprendizado, diferente para cada parâmetro, penalizando para dimensões mais íngremes do que para dimensões com curvas mais gentis. Ele ajuda a apontar os resultados das atualizações de forma mais direta para o ótimo global.

O algoritmo do AdaGrad é da seguinte forma:

$$\begin{aligned}s &\leftarrow s + \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \oslash \sqrt{s + \varepsilon}\end{aligned}$$

A primeira etapa acumula o quadrado dos gradientes no vetor $s = (s_1, \dots, s_d)$, onde d representa a dimensão de $\boldsymbol{\theta}$, o símbolo \odot representa a multiplicação elemento por elemento. Se a função custo é íngrime ao longo da i -ésima dimensão, então s_i aumentará a cada iteração. A segunda etapa é análoga ao gradiente descendente, com a diferença de que o vetor gradiente será alterado multiplicativamente pelo fator $\sqrt{s + \varepsilon}$, onde o símbolo \oslash representa a divisão elemento por elemento e ε é um termo de suavização para evitar divisão por zero.

O AdaGrad performance bem em problemas quadráticos simples, mas pode parar muito cedo quando é utilizado no treinamento de redes neurais. A taxa de aprendizado reduz tanto que o algoritmo acaba parando antes de atingir o ótimo global. Podemos ver uma comparação do seu desempenho com o gradiente descendente na Figura 7 a seguir.

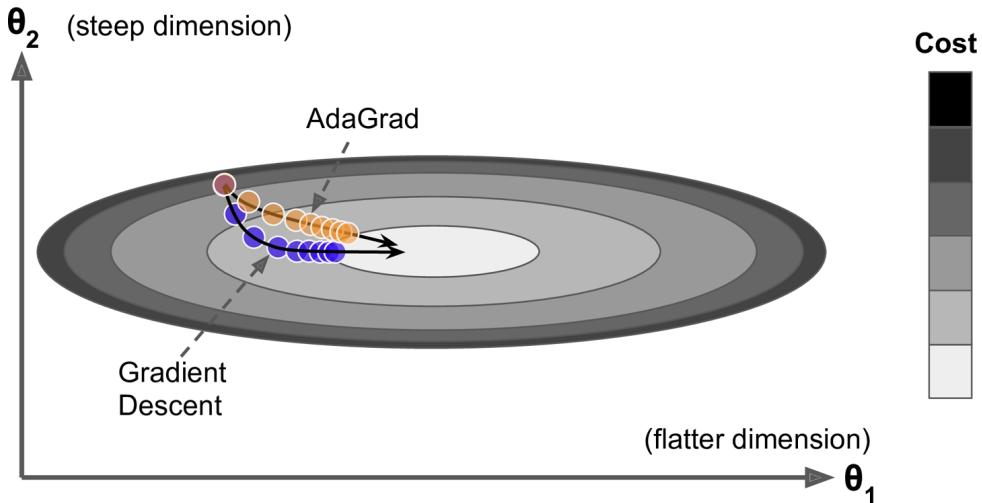


Figura 7: AdaGrad vs. Gradiente Descendente - Fonte: [5]

2.5.4 RMSProp

O *RMSProp* [9] é uma modificação do AdaGrad com o objetivo de resolver o problema a taxa de aprendizado se tornar muito pequena muito rápido e isso impedir que a convergência para o ótimo local ocorra.

O RMSProp modifica o AdaGrad para performar melhor em situações de não-convexidade, mudando a forma como o gradiente é acumulado em uma média móvel ponderada exponencialmente.

O RMSProp utiliza uma média exponencialmente decrescente para descartar informações antigas (primeira expressão do algoritmo), acumulando apenas os gradientes das iterações mais recentes (ao invés de todos os gradientes desde o começo do treinamento).

O algoritmo, onde β representa a taxa de decaimento, é dado pelas funções a seguir:

$$\begin{aligned} s &\leftarrow \beta_s + (1 - \beta) \nabla_{\theta} C(\theta) \odot \nabla_{\theta} C(\theta) \\ \theta &\leftarrow \theta - \eta \nabla_{\theta} C(\theta) \oslash \sqrt{s + \varepsilon} \end{aligned}$$

2.5.5 Adam

O método *adaptive moments* ou *Adam* [10] combina as ideias de otimização pelo método dos momentos e o RMSProp, com algumas distinções. Assim como no método dos momentos, o Adam adota o decaimento exponencial dos gradientes passados e, assim como no RMSProp, também ocorre penalização nas entradas do gradiente de acordo com as médias dos quadrados dos gradientes. Como há decaimento, esse otimizador não causa uma redução gradativa da taxa de aprendizado.

O algoritmo do Adam é dado por:

$$\begin{aligned} m &\leftarrow \beta_1 m + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \\ s &\leftarrow \beta_2 s + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \\ \hat{m} &\leftarrow \frac{m}{1 - \beta_1^t} \\ \hat{s} &\leftarrow \frac{s}{1 - \beta_2^t} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \eta \hat{m} \oslash \sqrt{\hat{s} + \varepsilon} \end{aligned}$$

Nesse algoritmo, temos que:

- t representa o número de iterações (começando em 1);
- β_1 e β_2 são os hiperparâmetros de decaimento dos momentos (tipicamente inicializados em 0.9 e 0.999, respectivamente);
- η é a taxa de aprendizado;
- m e s são inicializados em 0.

O Adam é geralmente visto como um método de otimização bastante robusto com respeito a escolha de hiperparâmetros, no entanto a taxa de aprendizado talvez precise ser manipulada para melhores resultados.

2.6 MLP para regressão e classificação

Os MLPs são usados para dois tipos de tarefas distintas: regressão e classificação. Nas tarefas de regressão, podemos estar interessados em prever um valor único (como o preço de um produto, dado um conjunto de características), onde teremos apenas um neurônio como saída, ou prever múltiplos valores, onde teremos um neurônio de saída para cada variável resposta. Em geral, nesse tipo de tarefa, quando as variáveis resposta variam

no conjunto dos números reais, sem restrições, como é o caso por exemplo dos modelos de regressão com resposta normal, não são utilizadas funções de ativação nos neurônios de saída. Isso impediria que fosse retornada uma faixa de valores qualquer.

Para os problemas de *classificação binária* teremos apenas um neurônio de saída que utilizará uma função de ativação logística. Neste caso, a saída do algoritmo será um número entre 0 e 1, que pode ser interpretado como a probabilidade estimada para a classe positiva. Para os casos em que temos atribuir múltiplas características a uma mesma entrada, precisamos de tantos neurônios na camada de saída quantos são as características de interesse, utilizando para cada um desses neurônios a função de ativação logística. Por exemplo, se quisesssemos prever se uma bebida é ou não alcóolica, e simultaneamente prever se ela é ou não doce, deveríamos ter na última camada dois neurônios com função de ativação logística. Esse tipo de classificação é chamado de *multilabel*.

Quando temos um caso em que a saída deve ser apenas uma entre várias classes possíveis, então a camada de saída da rede deve possuir um neurônio por classe, e devemos usar a função de ativação softmax (Figura 8). Esse tipo de classificação é chamado de *classificação multiclasse*.

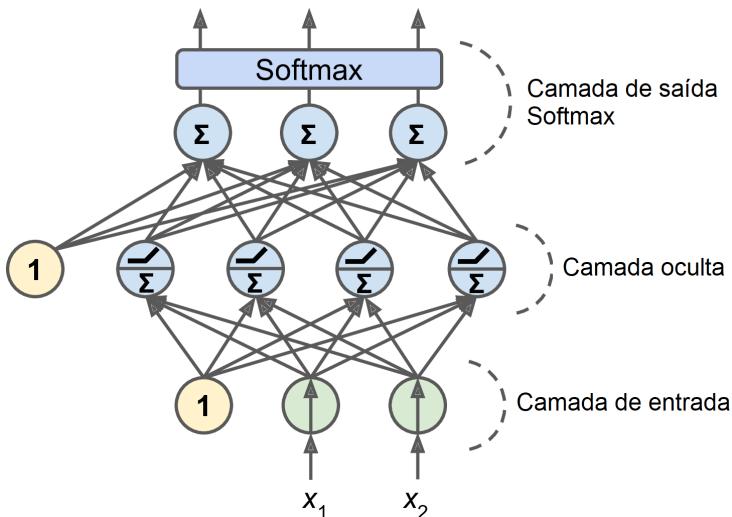


Figura 8: MLP de classificação com saída softmax - Fonte: [5]

2.7 Redes neurais convolucionais

Assim como uma rede neural profunda, a arquitetura de uma rede neural convolucional, ou CNN (*convolutional neural network*), consiste em empilhar neurônios em camadas ocultas com aplicação de funções de ativação. Contudo, ao invés de termos camadas com-

pletamente conectadas como nos MLPs, teremos *camadas convolucionais* e *camadas de pooling*.

A arquitetura de uma CNN consiste em camadas convolucionais recebendo uma entrada, geralmente seguidas de uma camada com função de ativação não-linear, por exemplo ReLU, e uma camada de *pooling*. Essa estrutura se repete diversas vezes conforme a rede vai se tornando mais profunda. Ao final dessa pilha de camadas é adicionada uma rede MLP a partir do output da última camada de convolução, que tem como objetivo de realizar a classificação, e uma última camada com função de ativação sigmoid ou softmax que retorna a previsão. Podemos ver um exemplo dessa arquitetura para a classificação de uma imagem contendo o número 5 na Figura 9.

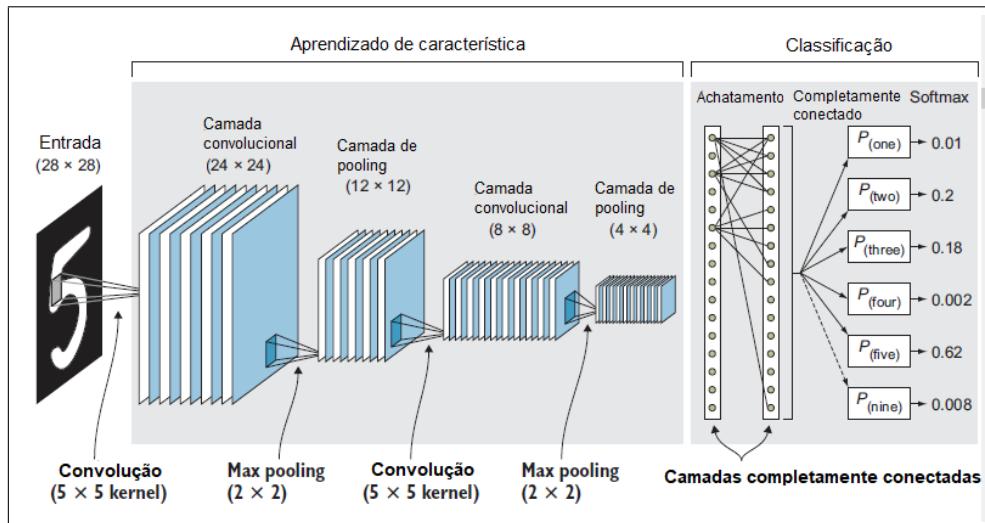


Figura 9: Arquitetura de uma CNN - Fonte: [11]

Neste capítulo será descrito o funcionamento das redes neurais convolucionais, como são caracterizadas cada uma das camadas que compõem a convolução e o passo a passo do processo de treinamento no contexto de classificação da rede.

2.7.1 Camadas convolucionais

O elemento mais importante de uma rede neural convolucional é a sua camada convolucional. Ela olha pixel por pixel da imagem buscando características que ajudem a identificar os objetos da imagem.

Os neurônios da camada convolucional não estão conectados a todos os pixels da imagem de entrada, como ocorre com a primeira camada de uma MLP. Ao invés disso, esses

nêurons possuem campos receptivos (*receptive fields*) que reagem apenas estímulos localizados em uma determinada região. Esses campos receptivos podem se sobrepor (ver Figura 10) para os diferentes nêurons e formar juntos todo o campo visual.

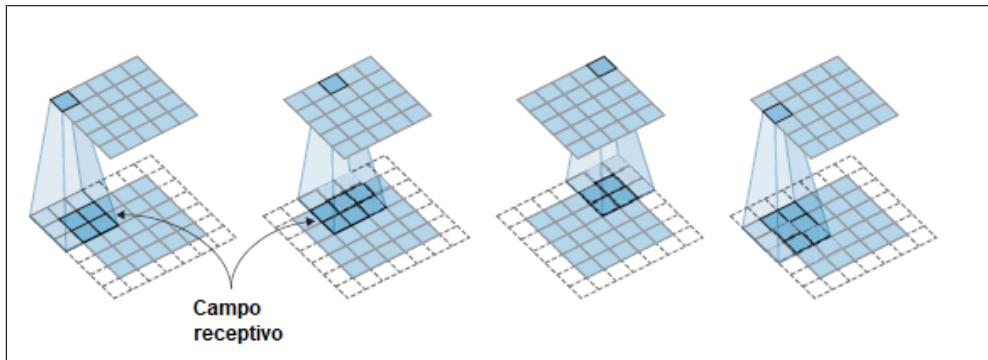


Figura 10: Campos receptivos de vários nêurons - Fonte: [11]

Os nêuros das camadas convolucionais seguintes seguem uma lógica semelhante, conectando-se apenas aos nêurons da camada anterior que estão localizados dentro dos seus campos receptivos. Isso faz com que as primeiras camadas da rede sejam especializadas em aprender características mais gerais da imagem, enquanto as camadas mais próximas do fim juntam essas informações e se tornam capazes de aprender características mais específicas.

2.7.2 Filtros

Assim como uma rede neural profunda tradicional, a convolução é uma operação linear que envolve a multiplicação de um conjunto de pesos com a entrada. Como no contexto de redes neurais convolucionais as entradas são bidimensionais (no caso de imagens em escala de cinza) ou tridimensionais (no caso de imagens em RGB), a multiplicação ocorre entre o vetor de entrada dos dados e uma matriz bidimensional (escala de cinza), ou array tridimensional (RGB), dos pesos, chamado de *filtro* ou *kernel* (ou núcleo).

O filtro utilizado é intencionalmente menor do que a entrada, uma vez que isso permite que o mesmo conjunto de pesos seja multiplicado pelo vetor de entrada múltiplas vezes em diferentes pontos da entrada. Isto é, o filtro é aplicado sistematicamente em cada parte sobreposta dos dados de entrada, e o tipo de multiplicação aplicada entre o filtro e o campo receptivo da entrada é o produto escalar, o que sempre resultará em um valor único numérico. A Figura 11 a seguir mostra a primeira aplicação de um filtro 3x3 sobre dados de entrada bidimensionais.

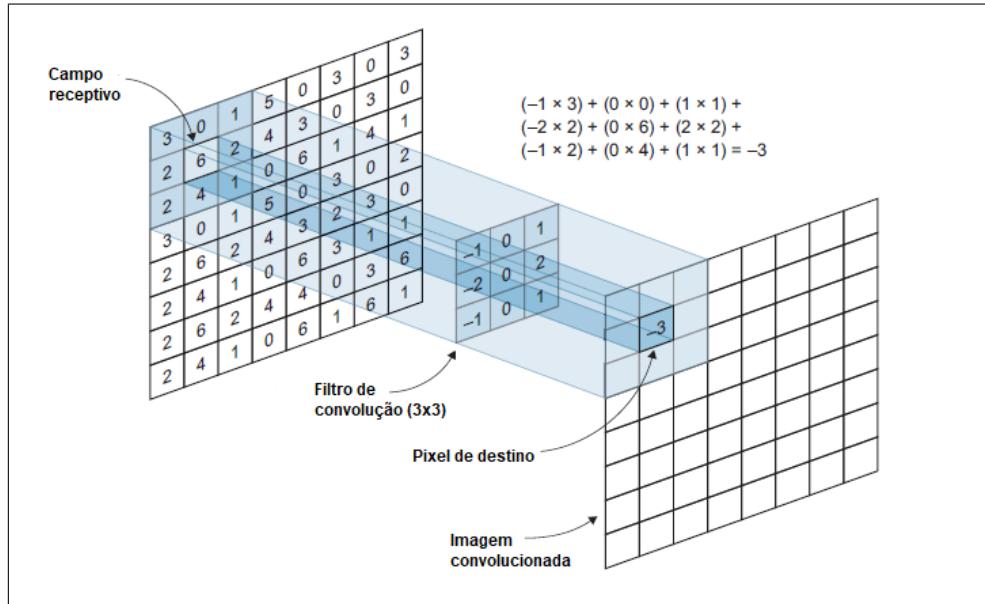


Figura 11: Filtro convolucional 3×3 deslizando sobre uma imagem de entrada - Fonte: [11]

Conforme o filtro é aplicado várias vezes ao vetor de entrada, o resultado obtido é uma matriz bidimensional de valores. A saída bidimensional dessa operação é chamada de *mapa de características* (ou *feature map*). Uma vez criado esse mapa aplicamos uma função de ativação não-linear assim como é feito nas camadas completamente conectadas do MLP.

Uma camada convolucional apresenta um número de filtros pré-determinado e retorna como saída um mapa de características por filtro, portanto é mais preciso representar camadas convolucionais em 3 dimensões (ver Figura 12). Ela possui um neurônio por pixel em cada mapa de características e todos os neurônios dentro de um mesmo mapa compartilham os mesmos pesos e viéses. Ou seja, a camada convolucional aplica múltiplos filtros aos inputs de forma simultânea, tornando-se capaz de detectar características diferentes em qualquer lugar das entradas.

Compartilhar os mesmos parâmetros faz com que o número total de parâmetros do modelo seja reduzido drasticamente. Uma vez que uma CNN aprendeu a reconhecer um padrão em um local, ela passa a ser capaz de reconhecer em qualquer outro local. Já um MLP seria capaz de reconhecer apenas nessa localização em particular.

É importante se atentar ao fato de que as imagens de entrada podem ser tanto coloridas quanto em preto e branco. No primeiro caso teremos uma canal para cada uma das

cores. Em geral, vermelho, verde e azul. No segundo caso teremos apenas um canal.

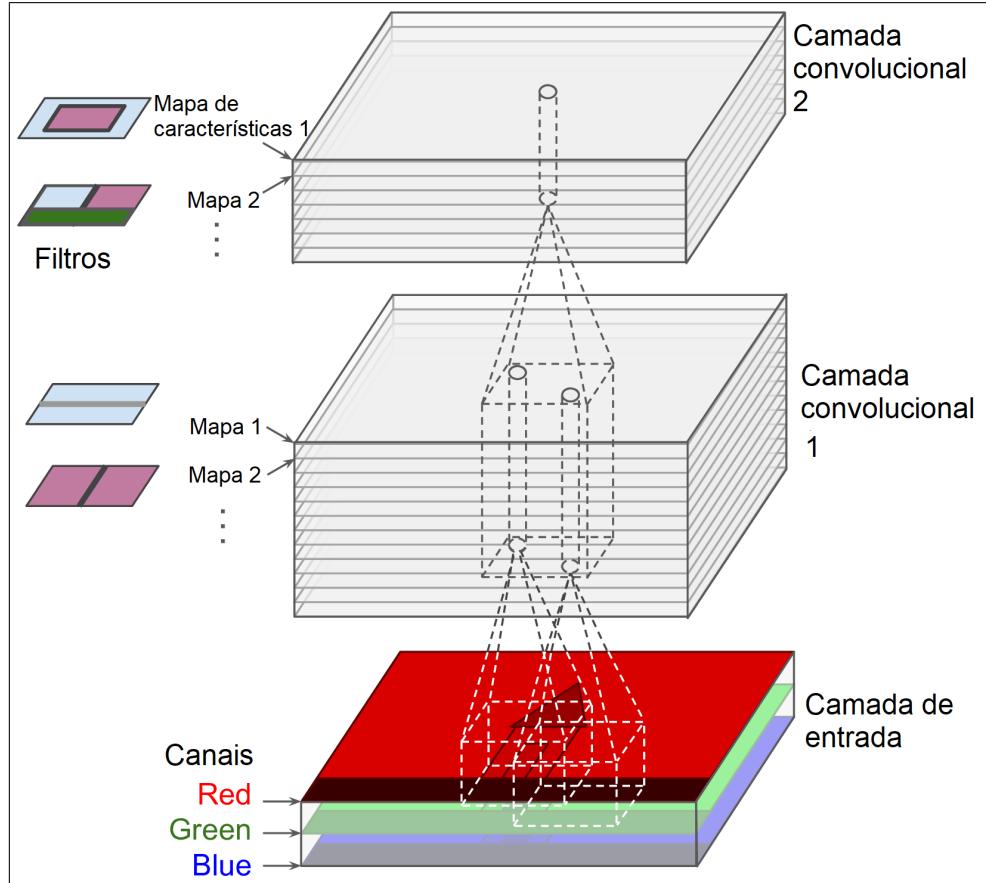


Figura 12: Camadas convolucionais com múltiplos mapas de características e imagem com três canais de cores - Fonte: [5]

2.7.3 Stride e padding

REESCREVER É possível conectar uma camada de entrada grande com uma camada menor espaçando os campos receptivos. Esse deslocamento de um campo perceptivo para o outro é chamado de *stride*. Também pode ser interpretado como a quantidade que o filtro desliza sobre a imagem. Esses saltos produzem um volume menor de saídas, o que reduz drasticamente a complexidade computacional do modelo. O stride é um valor numérico que representa o tamanho (em pixels) dos saltos em aplicações sucessivas de um filtro convolucional numa imagem de entrada.

Padding ou *zero padding* é utilizado quando queremos preservar a altura e a largura de uma entrada, para que tanto a entrada quanto a saída tenham as mesmas dimensões. Dessa forma, podemos utilizar camadas convolucionais sem precisar alterar as dimensões.

Esse método é importante, uma vez que, caso não fosse utilizado, as dimensões encolheriam conforme progredissemos para camadas mais profundas. Um exemplo tanto com *stride*, quanto com *zero padding* pode ser observado na Figura 13.

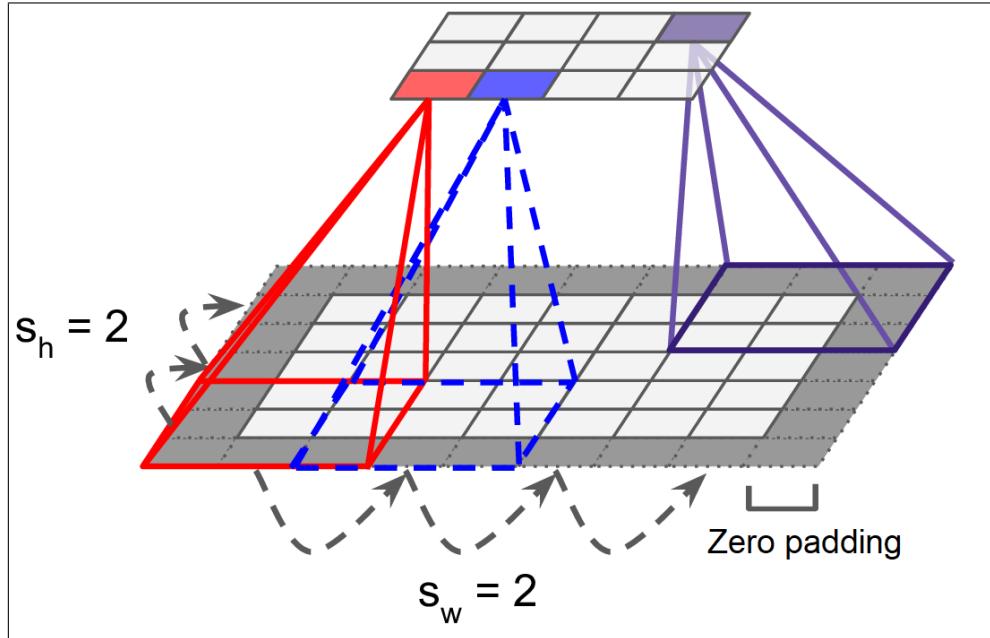


Figura 13: Zero padding e redução de dimensionalidade com stride de 2 - Fonte: [5]

Finalmente, podemos realizar o cálculo da saída de um neurônio específico em uma camada convolucional, representado na equação (3.7). Nela é calculada a soma ponderada de todos os inputs, mais o termo de viés.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \text{ com } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases} \quad (2.7)$$

Nesta equação, temos que:

- $z_{i,j,k}$ é a saída do neurônio da linha i , coluna j , no mapa k da camada convolucional (camada l);
- f_h e f_w são a altura e o largura do campo receptivo, e $f_{n'}$ é o número de mapas de características na camada anterior (camada $l - 1$)
- $x_{i',j',k'}$ é a saída do neurônio localizado na camada $l - 1$, linha i' , coluna j' , mapa k' (ou canal k' se a camada anterior é a camada de entrada);
- b_k é o viés para o mapa k (na camada l);

- $w_{u,v,k',k}$ é o peso de conexão entre qualquer neurônio no mapa k da camada l e sua entrada localizada na linha u , coluna v (relativos ao campo receptivo do neurônio) e mapa k' ;
- s_h e s_w são os *strides* vertical e horizontal.

2.7.4 Camadas de Pooling

Pooling (ou *subsampling*) tem como objetivo reduzir o custo computacional, o uso de memória e o número de parâmetros passado para a próxima camada. Essa redução do número de parâmetros reduz o risco de sobreajuste. A operação de pooling redimensiona a entrada aplicando uma medida estatística de resumo como por exemplo o maior valor da janela de pixels (*max pooling*), ou a média dos pixels (*average pooling*).

Assim como as camadas convolucionais, cada neurônio na camada de pooling possui um campo receptivo conectado a um número limitado de saídas da camada anterior. Além disso, também é possível definir *strides* e *padding* para essa camada.

Um neurônio da camada de *pooling* aplicam uma transformação no conjunto de pixels do campo receptivo e retorna um valor de acordo com o tipo de método que está sendo utilizado (ver Figura 14). O método de *average pooling* propaga para a próxima camada a média dos valores do campo receptivo. O método de *max pooling* é o tipo de pooling mais popular, onde apenas o valor máximo de input de cada pixel desse campo receptivo é propagado pela próxima camada.

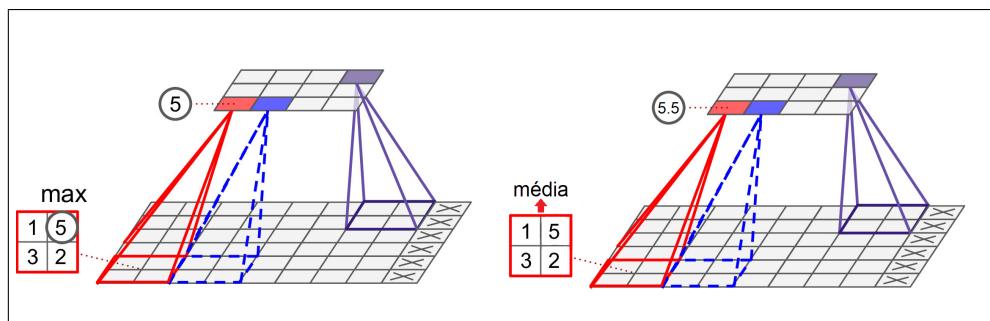


Figura 14: Camadas de max e average pooling, kernel 2x2, stride 2 e sem padding - Fonte: [5]

Além dos objetivos já citados, uma camada de max pooling também adiciona invariância à rede convolucional. A rede neural se torna capaz de identificar padrões espaciais em múltiplas áreas da imagem. Ao inserir uma camada de max pooling a cada certo número de camadas em uma rede neural convolucional, é possível obter algum nível

de invariância nas translações em larga escala. Apesar de pequena, essa invariância pode ser útil em casos onde a previsão não dependem desse tipo de detalhe, como é o caso em tarefas de classificação.

3 Classificação em imagens

Classificação de imagens no contexto de visão computacional se refere ao processo de categorizar e rotular imagens de acordo com um conjunto de regras determinados. Os métodos de classificação podem ser supervisionados, onde as imagens utilizadas no treinamento do modelo já possuem alguma forma de classificação, ou não-supervisionados, onde essa classificação não existe. Neste trabalho a forma de classificação será apenas supervisionada.

Os algoritmos de classificação possuem uma estrutura que se divide em alguma etapas:

- **Pré-processamento de imagem:** tem o objetivo de melhorar as características da imagem. Alguns passos dessa etapa são leitura da imagem, redimensionamento da imagem, e data augmentation;
- **Detecção na imagem:** se refere a localizar uma das classes na imagem;
- **Extração de características e treinamento:** etapa onde métodos estatísticos e de aprendizado profundo são implementados a fim de identificar padrões e características na imagem. Posteriormente essas características ajudarão a diferenciar entre as várias classes. Essa etapa em que o modelo aprende as características do conjunto de dados é chamado de treinamento do modelo;
- **Classificação da imagem:** se refere à etapa de categorização dos objetos em classes pré-definidas utilizando as características aprendidas pelo modelo na etapa anterior. Essa é a etapa final e onde o algoritmo retorna o resultado.

3.1 Arquiteturas

Os trabalhos referenciados neste capítulo propuseram arquiteturas inovadoras e ajudaram no desenvolvimento de redes neurais convolucionais profundas. Esses diferentes

designs compõem o estado da arte das tarefas de classificação de imagens e são importantes para entender as estruturas que ajudarão a desenvolver novos modelos.

Inception

A motivação principal da arquitetura Inception[12] é definido como uma estrutura local esparsa ótima em redes convolucionais[13]. Para isso são utilizados *módulos inception* (ou *inception modules*) empilhados, que permitem a utilização de parâmetros de forma eficiente.

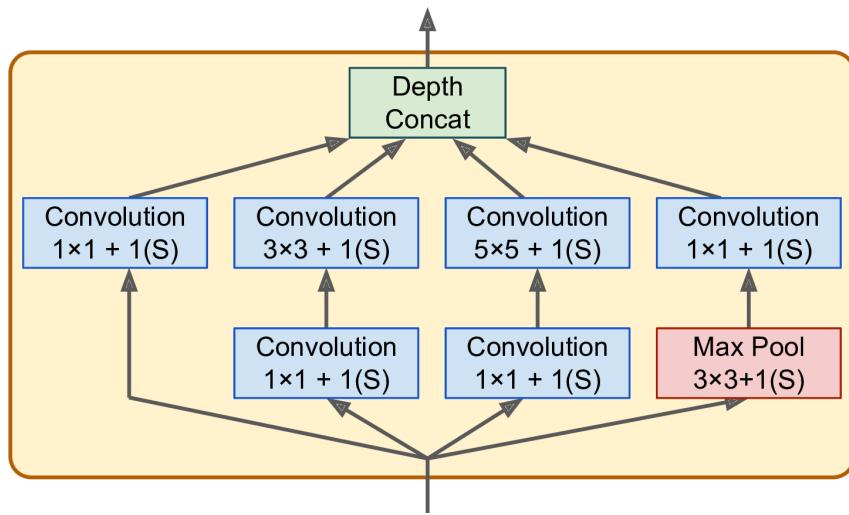


Figura 15: Módulo Inception com stride igual a 1 - Fonte: [5]

Os módulos são compostos de camadas convolucionais com tamanhos de kernel distintos, a fim de capturar padrões em escalas diferentes (no caso, utilizando kernels de dimensão 1x1, 3x3 e 5x5). As camadas 1x1 têm diferentes funções, e são centrais para o módulo inception

- Produzem, cada uma, uma combinação linear dos canais de input;
- Escolhendo um número de filtros 1x1 menor que a quantidade de canais de input temos como saída menos mapas de características do que entradas, reduzindo assim a dimensionalidade da rede;
- Os pares de camadas [1x1,3x3] e [1x1,5x5] agem como uma única camada convolucional mais poderosa, capaz de capturar padrões mais complexos.
- As camadas 1x1 ajudam a reduzir a quantidade de parâmetros dos filtros 3x3 e 5x5, uma vez que a quantidade de filtros 1x1 é menor que a quantidade de canais de input. Se não fossem esse filtros 1x1 a dimensão dos módulos inception obrigatoriamente aumentaria a cada camada.

Ao final de cada módulo todas as características extraídas são concatenadas antes de serem enviadas para a próxima camada. Durante o treinamento são decididas quais características são as mais importantes e os pesos dos kernels são determinados a partir disso.

A arquitetura do GoogLeNet[12] é um exemplo de arquitetura desenvolvida utilizando os módulos inception. É uma rede bastante longa, com 22 camadas contando apenas as camadas com parâmetros (ou 27 contando as camadas de pooling), contendo 9 módulos de inception, camadas de max pooling para a redução da imagem, uma camada dropout para regularização antes de uma camada completamente conectada e saída softmax. Todas as funções de ativação das camadas convolucionais são ReLU.

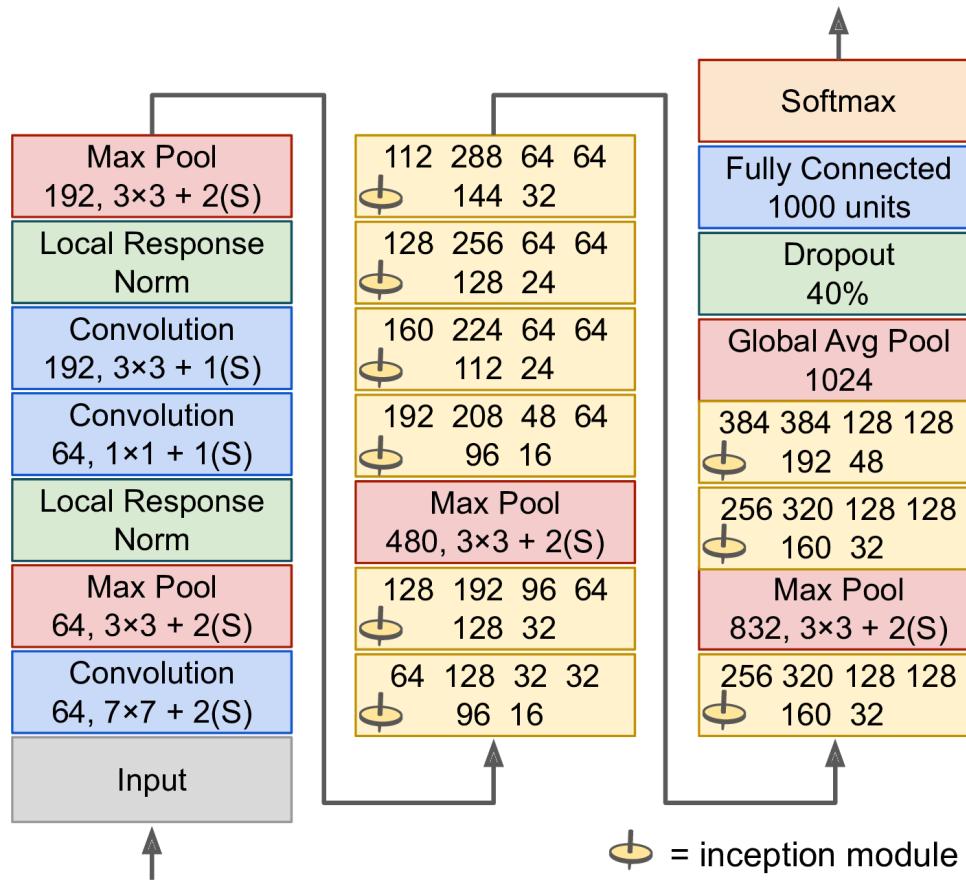


Figura 16: Arquitetura do GoogLeNet - Fonte: [5]

Algumas das arquiteturas apresentadas a seguir são variantes da arquitetura do GoogLeNet. Elas utilizam módulos inception com alguma alteração e apresentam melhora de performance em algum tipo de classificação.

VGGNet

Visual Geometry Group ou *VGG*[14] é uma arquitetura de redes neurais simples,

que utiliza filtros de convolução pequenos de 3x3 e apresenta duas versões com a mesma estrutura, tendo como diferença apenas o número de camadas: VGG-16, com 16, e VGG-19, com 19. A rede recebe imagens RGB com tamanho 224x224 e a única etapa de pré-processamento envolvida é a subtração da média dos valor RGB de cada pixel.

A imagem recebida passa por camadas convolucionais empilhadas, onde são aplicados filtros 3x3, com stride fixo de 1 pixel, o processo de pooling é realizado por 5 camadas de max-pooling em uma janela 2x2, com stride de 2 pixels. As camadas convolucionais empilhadas são seguidas por uma camada totalmente conectada, e a última camada é uma softmax (Figura 17). Todas as camadas possuem função de ativação ReLU. A primeira camada tem largura começando em 64 e aumenta em fator de 2 depois de cada camada de max-pooling até atingir 512.

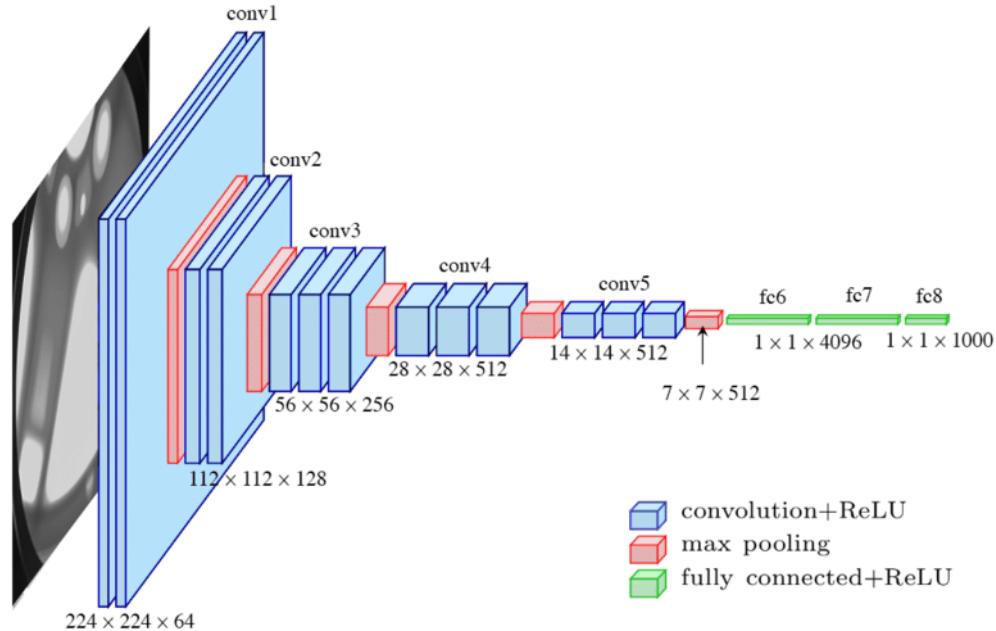


Figura 17: Arquitetura do VGG - Fonte: [15]

O VGG se vale do fato de duas camadas convolucionais 3x3 (sem pooling) terem um campo receptivo equivalente ao campo receptivo de uma camada convolucional 5x5, e três camadas convolucionais 3x3 terem um campo receptivo equivalente ao campo receptivo de uma camada convolucional 7x7. Incorporar múltiplas camadas em uma torna a função de decisão mais discriminativa a medida que mais funções não-lineares são aplicadas e, além disso, o número total de parâmetros é menor, tornando o processamento mais eficiente.

Inception v3

Assim como a versão original do Inception, essa versão utiliza módulos inception em

sua arquitetura, mas sugere algumas formas de fatoração das camadas convolucionais, a fim de tornar o trabalho computacionalmente mais eficiente.

Uma das fatorações sugeridas é a de camadas convolucionais grande, como 5x5 ou 7x7, em camadas convolucionais 3x3, similar ao que ocorre no VGG. Com isso é possível realizar a mesma computação com um número menor de parâmetros e, como múltiplas camadas convolucionais têm mais não-linealidade entre elas do que em uma camada maior, o poder discriminativo também é maior. A Figura 18 mostra uma comparação entre o módulo inception da versão original e da versão v3.

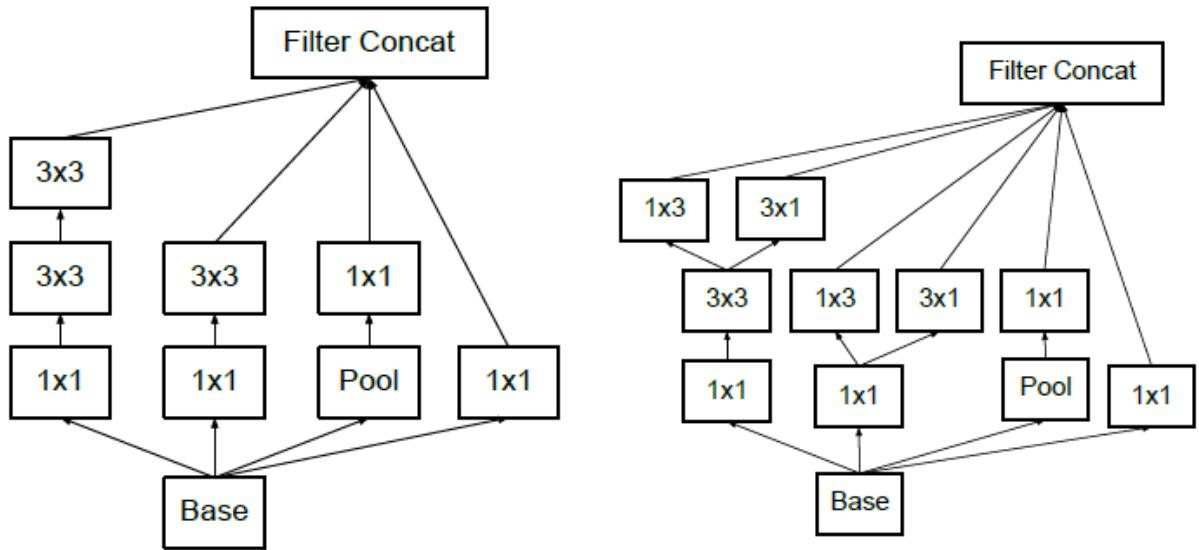


Figura 18: 1) Módulo inception com a convolução 5x5 dividida em duas 3x3; 2) Módulo inception após fatorização das convoluções em $1xn$ e $nx1$. - Fonte: [16]

Outra fatoração sugerida pelo Inception-v3 é a fatoração das camadas convolucionais de forma assimétrica. Por exemplo, utilizar uma convolução 3x1 seguida de uma convolução 1x3 é equivalente uma convolução 3x3 (ver Figura 18), ao mesmo tempo que torna o processo mais barato computacionalmente. Substituir uma convolução $n \times n$ por uma $1 \times n$ seguida de uma $n \times 1$ diminui o custo computacional conforme n aumenta, no entanto, na prática, esse tipo de fatorização não apresenta bons resultados quando aplicados nas primeiras camadas e são melhores quando aplicados em *grades* de tamanho médio.

Um classificador auxiliar é descrito como uma pequena rede neural convolucional inserida entre as camadas durante o treinamento, e a perda incorrida é adicionada a perda da rede principal. Originalmente, o Inception utiliza esses classificadores em redes muitos profundas com o objetivo de combater o problema de dissipação do gradiente (*van-*

(*nishing gradient*), melhorando a convergência da rede. Quando utilizados na arquitetura do Inception-v3 esses classificadores auxiliares funcionam como regularizadores. A redução do tamanho dos mapas de características é, tradicionalmente, feito por operações de pooling.

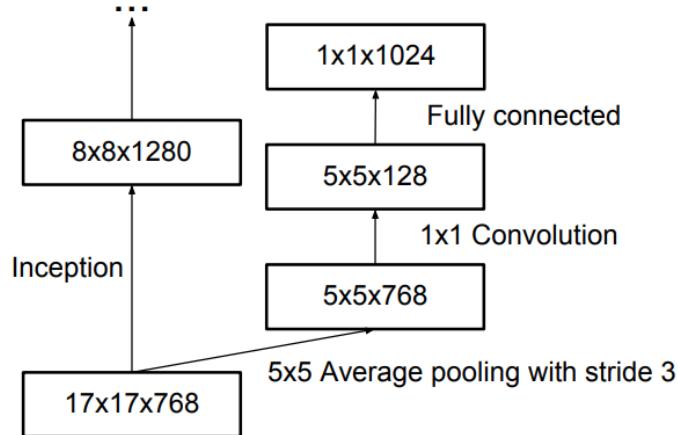


Figura 19: Classificador auxiliar - Fonte: [16]

Xception

Xception, que significa *Extreme Inception*, é uma outra variação da arquitetura Inception apresentada anteriormente. Essa arquitetura parte da hipótese fundamental por trás do Inception, que sugere que as correlações entre canais e as correlações espaciais são suficientemente dissociadas e, por isso, é preferível que não seja mapeadas conjuntamente. Nesse caso, assume-se que as correlações entre canais e as correlações espaciais podem ser mapeadas de forma completamente separada.

Essa versão do módulo Inception é essencialmente equivalente a uma operação existente conhecida como *depthwise separable convolution*, que consiste em uma convolução *depthwise* (uma convolução espacial realizada de forma independente sobre cada canal do input) seguida por uma convolução *pointwise* (uma convolução 1x1 em todos os canais). Uma forma fácil de entender é pensar as correlações em um espaço 2D primeiro e, em seguida, as correlações no espaço 1D. Essa separação é de aprendizado mais fácil do que pensar o espaço 3D inteiro.

A arquitetura Xception é formada por uma pilha de camadas convolucionais *depthwise* separáveis com conexões residuais. Mais especificamente, 36 camadas convolucionais estruturadas em 14 módulos (ver Figura 20)

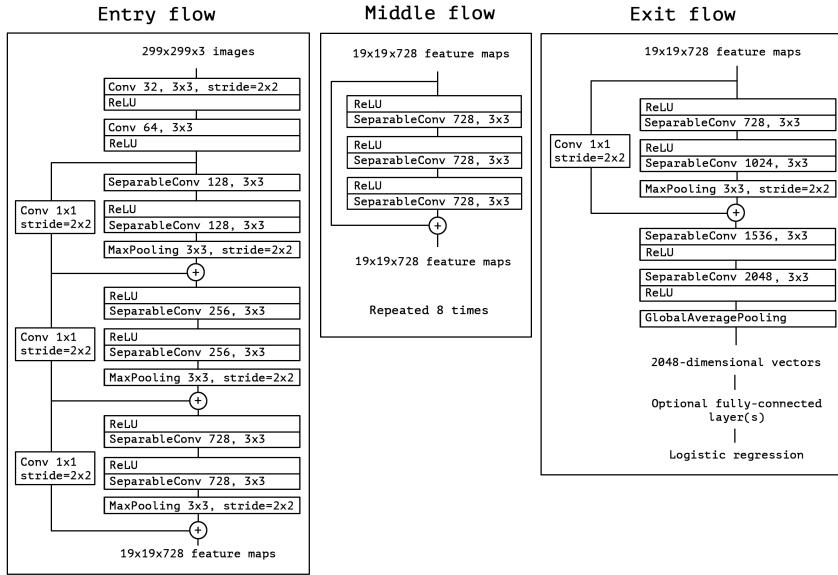


Figura 20: Arquitetura do Xception. Os dados entram, passando pela estrutura do lado esquerdo. Em seguida, passam pela estrutura do meio 8 vezes, e finalmente pela estrutura de saída na direita. - Fonte: [17]

3.2 Transfer Learning

Em um problema prático, em que desejamos treinar um classificador, é conveniente escolher algum modelo pré-treinado que apresente similaridades com a tarefa de interesse, para então reutilizar as camadas mais intermediárias dessa rede no nosso próprio modelo. Essa técnica é chamada *transfer learning* e torna o treinamento muito mais rápido, além de demandar menos dados, ao servir como método de inicialização dos pesos da rede que temos o interesse de treinar.

As primeiras camadas da rede neural são responsáveis por identificar as características mais fundamentais das imagens, como contornos, cores e curvas. Conforme a rede se torna mais profunda, mais características complexas são aprendidas. Tarefas que são similares tendem a ter essas características similares, o que torna interessante reutilizar essas camadas durante o treinamento de um novo modelo. Por exemplo, se temos uma rede neural profunda treinada para identificar imagens de diferentes categorias como animais, plantas e veículos, podemos reutilizar partes dessa rede para treinar uma nova rede que identifica um tipo específico de animal.

Em geral, as últimas camadas do modelo pré-treinado não costumam ser úteis como as camadas iniciais, uma vez que as características mais específicas, que são mais úteis para descrever as novas classes, podem ser muito diferentes das classes do modelo inicial.

Por esse motivo, congelamos as camadas iniciais do modelo pré-treinado, isto é, tornamos os seus pesos não-treináveis, e treinamos apenas uma ou duas camadas finais (ver Figura 21).

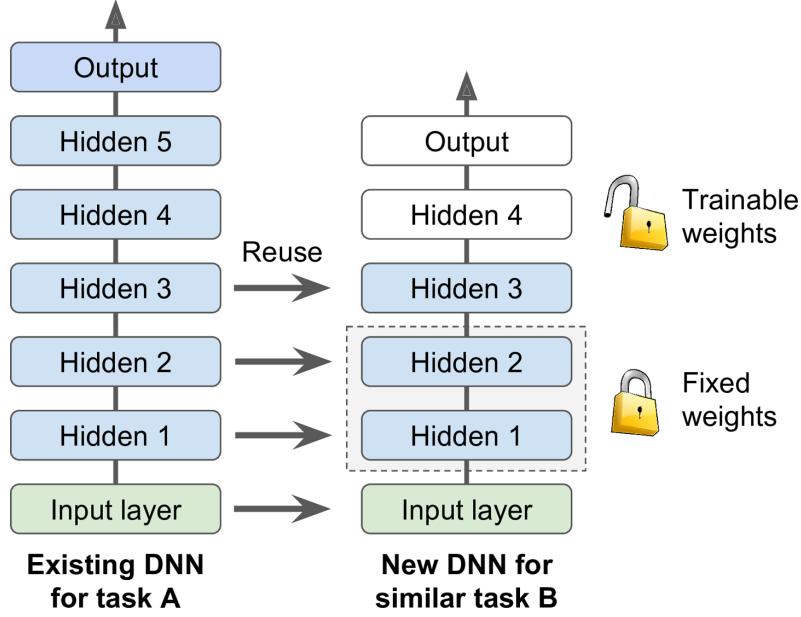


Figura 21: Utilizando camadas pré-treinadas. - Fonte: [5]

As arquiteturas apresentadas no capítulo anterior são popularmente utilizadas em tarefas de classificação que envolvem transfer learning por já terem sido testadas em competições e performado extremamente bem utilizando o conjunto de dados do *ImageNet* [1] (esse conjunto será descrito formalmente no capítulo seguinte). A Tabela 1 a seguir mostra uma comparação entre a acurácia de modelos implementados com as arquiteturas citadas nesse capítulo.

Tabela 1: Classificação de performance no ImageNet [17]

Modelo	Top-1 Acurácia	Top-5 Acurácia
VGG-16	0.715	0.901
ResNet-152	0.770	0.933
Inception-v3	0.782	0.941
Xception	0.790	0.945

A tarefa de classificação realizada neste trabalho utilizará o Xception como base da transferência de aprendizado, uma vez que se mostrou o mais eficiente dentre os modelos apresentados.

Imagenet

ImageNet é uma base de dados grande com imagens anotadas de diversas classes

diferentes. Tem o objetivo ser uma referência que possa ser utilizada para comparar o desempenho de diferentes modelos de classificação de imagens e ser uma conjunto de dados em larga escala tanto para treino, quanto para teste, capaz de tornar mais prático qualquer tipo de trabalho envolvendo visão computacional. Atualmente, é a principal referência para avaliar algoritmos de classificação.

As arquiteturas mais populares, como as apresentadas anteriormente neste capítulo, permitem que o usuário carregue versões da rede pré-treinada no conjunto de dados do ImageNet. Como as redes neurais obtidas dessas forma já são capazes de classificar um conjunto grande de objetos, elas se tornam ideais para ajudar a treinar novos modelos que reconheçam classes fora do ImageNet, utilizando transferência de aprendizado.

Neste trabalho serão testados três modelos diferentes para classificar duas classes não pertencentes ao ImageNet, a fim de concluir se a transferência de aprendizado é realmente mais eficiente do que treinar o modelo do zero. O primeiro modelo utilizará a arquitetura Xception com os pesos pré-treinados no ImageNet, o segundo modelo utilizará a arquitetura Xception sem os pesos pré-treinados, e o terceiro modelo será apenas um modelo MLP treinado do zero.

3.3 Aplicação

Essa análise tem como interesse comparar os resultados de modelos e mostrar a validade do método de Transfer Learning em tarefas de classificação. Temos como alvo a classificação de imagens de capivaras e equinos.

Foram utilizadas 1004 imagens para treinamento, sendo 502 imagens de capivaras e 502 de equinos, e 249 para validação, sendo 144 imagens de capivaras e 105 de equinos, coloridas e com dimensão 224x224. Como o volume de dados não é muito grande, utilizou-se um método de *data augmentation* para aumentar a quantidade de imagens disponíveis para treino a partir das imagens originais. Esse método consiste em incluir no conjunto de treino várias versões de imagens já existente, após ser rotacionada em diferentes direções. Um exemplo dessa aplicação nas imagens que foram utilizadas pode ser visto na Figura 22. Todas as imagens de treinamento e validação foram coletadas no Google Imagens utilizando um algoritmo de *scraping* no Python.

A linguagem de programação escolhida para os experimentos foi o Python, dada a praticidade na hora de trabalhar com machine learning oferecido pelas suas bibliotecas. Foram utilizados o Keras e Tensorflow para o treinamento das redes neurais, NumPy para

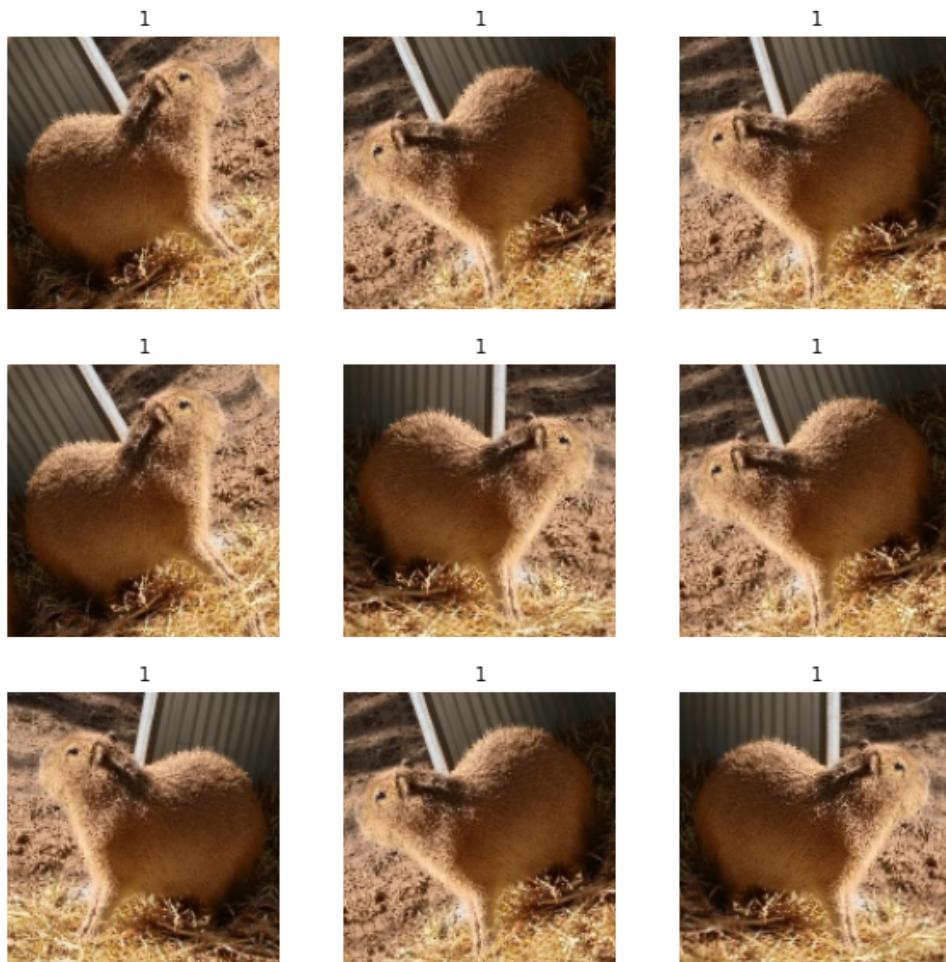


Figura 22: Exemplo de data augmentation utilizando uma das imagens de capivara do conjunto de teste.

o processamento de matrizes e geração de números aleatórios, scikit-learn para a análise de dados, e matplotlib para a criação de gráfico e processamento de imagens.

Foram comparados três modelo para classificação de imagens. O primeiro modelo utiliza transfer learning com pesos treinados no Xception para as primeiras camadas, seguido por um MLP com saída softmax para a classificação. O segundo modelo utiliza a mesma arquitetura do primeiro, porém sem os pesos pré-treinados. O terceiro modelo é apenas um MLP treinado como inicialização aleatória (sem transfer learning). Os modelos foram treinado para 25 épocas e com tamanho de batch 64.

Xception utilizando Transfer Learning

O primeiro modelo testado foi um modelo com arquitetura Xception, utilizando pesos treinados ImageNet, seguido por uma estrutura MLP formada por duas camadas densas com 1024 neurônios com função de ativação ReLU, uma camada densa com 512 neurônios também com camada de ativação ReLU, e uma camada de saída com 2 neurônios e função de ativação softmax.

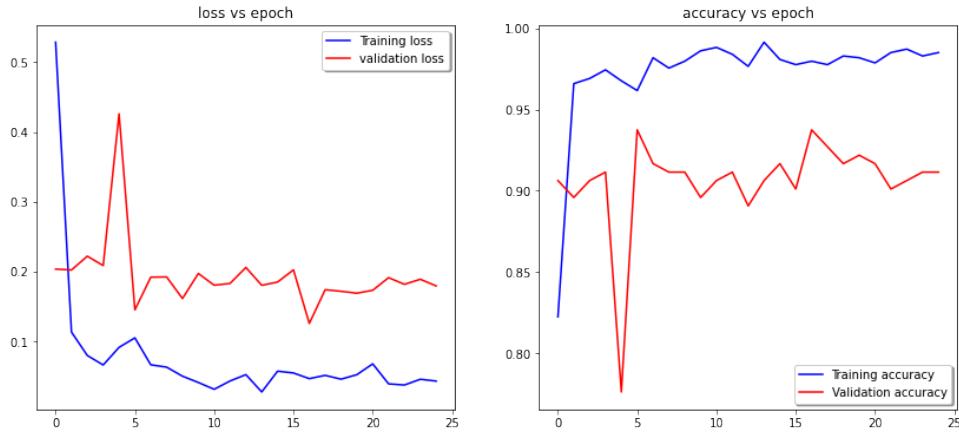


Figura 23: Gráficos de perda e acurácia para o modelo Xception usando transfer learning.

Podemos observar na Figura 23 que a perda do modelo para o conjunto de treinamento cai rapidamente, atingindo um valor inferior a 10% nas primeiras 3 épocas, aumenta sutilmente até a quinta época, e depois volta a cair para as épocas seguintes e se torna de certa forma constante. Para o conjunto de validação, a perda fica estável um pouco abaixo dos 20% após a época 5, e não apresenta grande melhora até o fim do treinamento. A acurácia se comporta de forma semelhante, o conjunto de treino apresenta altos valores já nas primeiras épocas, se mantendo estável após a época 5, e o conjunto de validação apresenta acurácia um pouco acima dos 90% após a época 5 e mantém uma variação pequena até o fim.

Tabela 2: Classificação de performance no ImageNet

	Precisão	Recall	F1	Suporte
Equino	0.83	1.00	0.90	114
Capivara	1.00	0.82	0.90	135
Acurácia			0.90	249
Macro avg	0.91	0.91	0.90	249
Weighted avg	0.92	0.90	0.90	249

Os valores da matriz de confusão contidos na Tabela 2 mostram a capacidade do

modelo detectar imagens da classe “Equino” com 83% de precisão e imagens da classe “Capivara” com 82% de precisão, além de ser capaz de detectar falsos positivos e falsos negativos de forma extremamente eficiente. Este modelo apresentou acurácia de 90%, o que pode ser considerado alta, após poucas épocas de treinamento. A coluna “Suporte” indica o número de ocorrências de cada classe nos dados reais. O score $F1$ pode ser interpretado como a média harmônica entre a precisão e o recall, onde 1 é o melhor valor e 0 é o pior. A fórmula para o score $F1$ é $F1 = 2 * (precision * recall) / (precision + recall)$.

Xception modificada sem Transfer Learning

O segundo modelo, assim com o primeiro, também utilizou a arquitetura Xception, porém sem os pesos pré-treinados no ImageNet. As camadas superiores são compostas pela mesma estrutura MLP apresentada no modelo anterior: uma camada de GlobalAveragePooling2D, seguida por duas camadas densas com 1024 neurônios com função de ativação ReLU, uma camada densa com 512 neurônios também com camada de ativação ReLU, e uma camada de saída com 2 neurônios e função de ativação softmax.

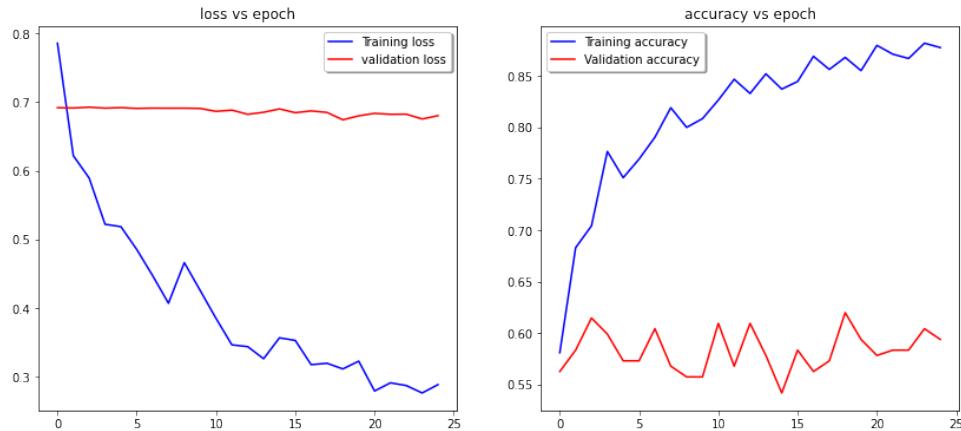


Figura 24: Gráficos de perda e acurácia para o modelo Xception sem usar transfer learning.

O gráfico de perda por época da Figura 24 mostra que, ao final das 25 épocas, apenas o conjunto de treinamento apresentou uma redução significativa, enquanto o conjunto de validação permaneceu aproximadamente constante em 70%. O comportamento da acurácia é semelhante, apresentando melhora significativa apenas no conjunto de treinamento, atingindo valores superiores a 85%. O conjunto de validação parece variar em torno de 55%. O comportamento das curvas parece indicar que há um sobreajuste no treinamento do modelo. Isso provavelmente está ocorrendo devido ao pequeno número de amostras de treino, sobretudo levando em consideração o elevado grau de complexidade do modelo Xception, que provavelmente demanda um alto número de imagens de treino

para evitar sobreajuste.

Tabela 3: Classificação de performance no ImageNet

	Precisão	Recall	F1	Suporte
Equino	0.00	0.00	0.00	123
Capivara	0.51	1.00	0.67	126
Acurácia			0.51	249
Macro avg	0.25	0.50	0.34	249
Weighted avg	0.26	0.51	0.34	249

A Tabela 3 apresenta um resultado interessante. Aparentemente, esse modelo não é capaz de detectar a classe “Equino”, e mesmo a classe “Capivara” aparece com precisão baixa. Esse modelo tem acurácia total de 51%. Modelos com valores de acurácia próximos ou inferiores a 50% não podem ser considerados úteis, uma vez que ter 50% de acurácia indica que não há diferença entre selecionar o modelo e fazer um sorteio aleatório.

MLP

O terceiro modelo é simplesmente um MLP sem camadas convolucionais. Ele possui a mesma estrutura das camadas superiores dos modelos anteriores: duas camadas densas com 1024 neurônios com função de ativação ReLU, uma camada densa com 512 neurônios também com camada de ativação ReLU, e uma camada de saída com 2 neurônios e função de ativação softmax.

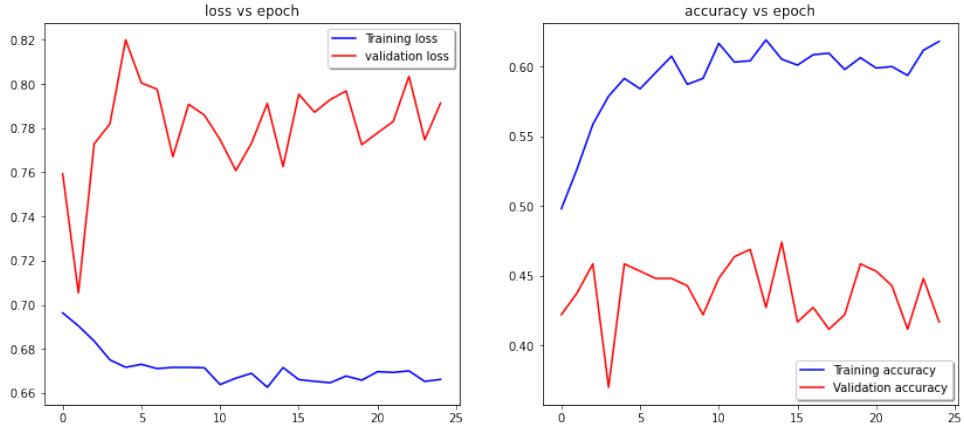


Figura 25: Gráficos de perda e acurácia para o modelo com apenas camadas MLP.

Neste caso, pode-se ver na Figura 25 que a perda no conjunto de treino tende a se estabilizar em um alto valor próximo a 67%, enquanto o conjunto de validação apresenta perdas próximas de 80%. A acurácia de treinamento se estabiliza em torno dos 60%, enquanto a de validação fica consistentemente abaixo dos 50%, próxima dos 45%.

Tabela 4: Classificação de performance no ImageNet

	Precisão	Recall	F1	Suporte
Equino	0.47	0.46	0.46	140
Capivara	0.33	0.34	0.33	109
Acurácia			0.41	249
Macro avg	0.40	0.40	0.40	249
Weighted avg	0.41	0.41	0.41	249

A Tabela 4 apresenta valores inferiores a 50% para a detecção de ambas as classes, indicando que esse modelo não é ideal para esse tipo de análise. A acurácia total do modelo foi de 41%, indicando que é pior do que escolher aleatoriamente a classe da imagem de entrada.

Conclusão

O terceiro modelo foi o que performou pior entre todos, apresentado apenas 40% de acurácia. Esse caso serve para ilustrar a necessidade da inclusão de camadas convolucionais para o processo de classificação de imagens, onde as camadas MLP não se mostram suficiente em reconhecer os padrões.

O segundo modelo também não foi capaz de performar bem. Os gráficos mostram que houve um sobreajuste no modelo, uma vez que não há redução na perda, nem melhora na acurácia, para o conjunto de validação, enquanto o conjunto de treino progride com o passar das épocas. Além disso, o modelo só foi capaz de detectar uma das classes ao final do período de treinamento. Talvez fosse necessária uma base de dados maior e um número maior de épocas para que esse tipo de modelo fosse capaz de obter bons resultados. Entre todos os três modelos esse foi o que demandou o maior tempo para treinamento, levando mais de 5 vezes o tempo necessário para rodar o modelo com transfer learning, pela mesma quantidade de épocas.

O primeiro modelo apresentou resultados superiores e seu custo computacional foi inferior ao segundo modelo (treinado do zero). Foi observado 90% de acurácia de validação e valores acima 80% para precisão e recall para o modelo base. Após a etapa de fine-tuning, os valores apresentaram uma melhora considerável de 4%, atingindo uma acurácia final de 94%. Percebe-se que a utilização do transfer learning de fato implica em maior eficiência computacional e melhores resultados preditivos.

4 Detecção em imagens

Problemas de classificação de imagem podem ser considerados aplicações simples, uma vez que não incluem localização dos objetos de interesse. Para isso, é necessário adicionar outras camadas de complexidade às redes, tornando elas capazes de reconhecer múltiplos objetos e suas localizações na imagem. Esse tipo de tarefa é chamada de *detecção de objetos*.

As tarefas que envolvem detecção de objetos consistem em localizar um ou mais objetos na imagem e classificar esses objetos. Isso é feito utilizando *caixas delimitadoras* (ou *bounding boxes*) em volta de um objeto que corresponda a uma das classes de interesse. Para cada objeto na imagem, uma caixa será utilizada para determinar sua posição. A caixa delimitadora é retangular e é determinada pela coordenada (x, y) do centro da caixa, juntamente com sua altura e largura. Logo, diferentemente do que foi visto nos capítulos anteriores, há a necessidade de prever as coordenadas da caixa, e não apenas a classificação da imagem.

Resumidamente,

- Classificação de imagens: prevê a classe de um objeto em uma imagem. Recebe como entrada a imagem de um objeto único e retorna a probabilidade de classe. A performance de um modelo de classificação de imagens é avaliado usando a média do erro de classificação da classe predita.
- Detecção de objetos: localiza objetos em uma imagem e indica suas classes e posições utilizando caixas. Recebe uma imagem com um ou mais objetos e retorna caixas ao redor dos objetos encontrados, juntamente com a classe mais provável. A performance para um modelo de detecção de objetos é avaliada usando a precisão e o recall entre cada uma das melhores caixas delimitadoras para os objetos conhecidos na imagem.

4.1 Estrutura

Neste capítulo serão apresentadas as quatro principais componentes que formam a estrutura de redes neurais para uma tarefa de detecção de objetos. O objetivo das seções a seguir é apresentar um resumo das etapas e gerar uma intuição sobre como os algoritmos funcionam. As duas primeiras seções “Região proposta” e “Previsões da rede” descrevem metodologias comuns aos algoritmos da família R-CNN [18] e SSD [19], que não serão utilizados nesse trabalho, mas serão descritas a título de contextualização. As outras seções apresentam mecanismos de filtragem e avaliação, e não são específicas de nenhum algoritmo.

Alguns modelos de detecção de objetos são divididos em duas partes. Primeiro, são identificadas as regiões de interesse (caixas delimitadoras) onde o objeto está presente e, em seguida, essa região de interesse é classificada. Esse tipo de abordagem é comum aos modelos da família R-CNN[18]. No entanto, neste trabalho, foi escolhida a utilização de um modelo da família YOLO[20]. Este tipo de detector trata a identificação e classificação do objeto em uma única etapa e, em geral, performa melhor em termos de velocidade e acurácia. A metodologia do YOLO será apresentada posteriormente.

Região proposta

O sistema olha a imagem e propõe *regiões de interesse* (RoIs[21]) através de clusterização hierárquica de pixels para análises posteriores. As regiões de interesse são regiões da imagem em que o algoritmo acredita que haja alta probabilidade de conter um objeto. Um score é atribuído a essas regiões, a rede analisa e classifica cada região como objeto ou não-objeto baseada nesses scores, e apenas aquelas com valor acima de um certo limiar pré-selecionado são passadas para a próxima etapa.

O valor do limiar de score que define quais regiões serão ou não passadas para a próxima etapa depende do tipo de problema que está sendo tratado. Se o limiar for muito baixo, a rede gerará todas as regiões possíveis e isso tornará a chance de detectar todos os objetos na imagem maior, no entanto, resultando em um custo computacional muito grande, o que torna a escolha do valor de limiar de uma tarefa específica para cada problema.

A Figura 26 mostra algumas regiões de interesse em uma imagem real. As regiões em rosa são áreas em que há grande chance de ter um objeto da classe ”Cachorro” e as áreas em verde um objeto da classe ”Capivara”.

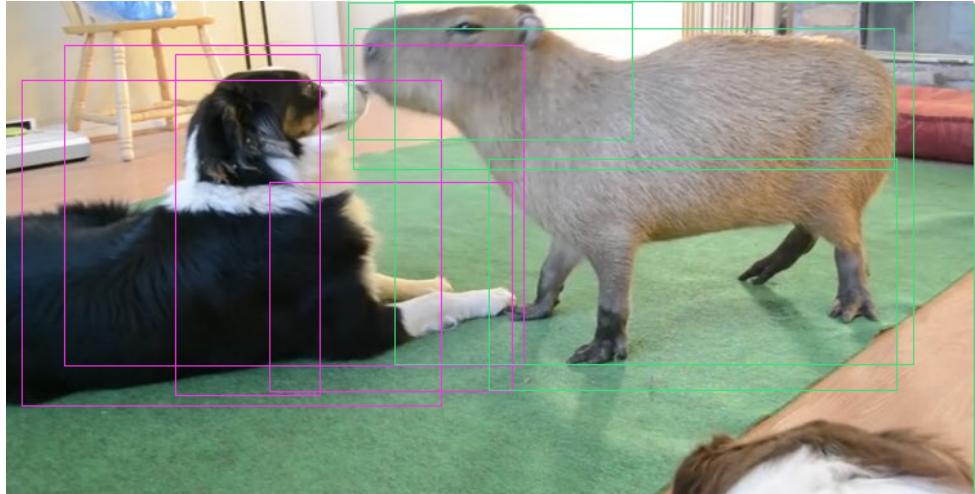


Figura 26: Regiões de interesse com alto score representam regiões com grandes chances de conterem os objetos de interesse.

Previsões da rede

Esta etapa consistem em uma a rede neural convolucional pré-treinada para classificação a partir da qual extraem-se características representativas da imagem de entrada. Essas características nada mais são que outputs de alguma camada intermediária da rede e são utilizadas para indicar a classe de novas imagens. Assim como em tarefas de classificação de imagens, também costuma-se utilizar transfer learning em modelos treinados para reconhecer características de imagens de classes relacionadas com o que temos interesse em classificar. Em problemas de detecção de objetos, modelos treinados no ImageNet ou no MS COCO são bastante populares.

Todas as regiões com alta probabilidade de conter um objeto são analizadas e duas previsões para cada região são feitas pelo modelo. Uma das previsões diz respeito às coordenadas das caixas delimitadoras, representadas por um vetor (x, y, w, h) , onde x e y são as coordenadas do centro da caixa e w e h são o comprimento e a largura da caixa. A outra previsão é realizada por uma saída softmax e diz respeito às probabilidades de cada classe para aquele objeto.

Assim como na seleção das regiões de interesse, sempre teremos várias caixas para um mesmo objeto, como pode ser visto na Figura 27. Apesar do objetivo de detecção ser atendido por esse método, não é interessante ter várias caixas sobre o mesmo objeto para que não haja confusão sobre quantos objetos da mesma classe estão presentes na imagem. Por isso, a seguir será apresentada uma forma de reduzir o output final a apenas uma caixa por objeto.

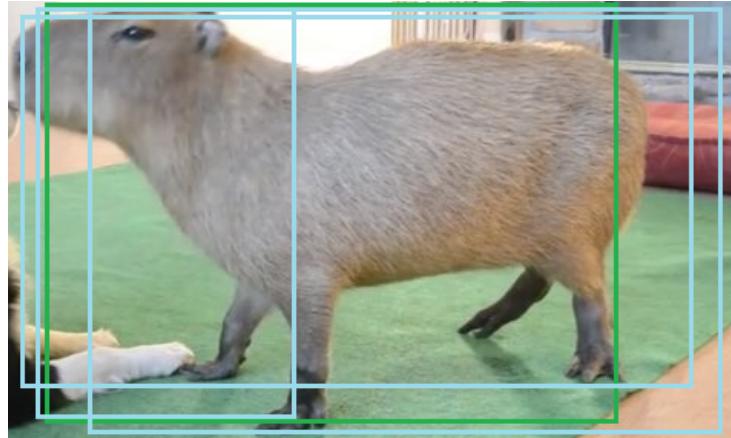


Figura 27: O detector produz várias caixas delimitadoras para um objeto.

Non-maximum suppression

Para que múltiplas detecções não sejam realizadas para o mesmo objeto aplica-se uma técnica chamada *Non-maximum suppression* ou *NMS*. Essa técnica olha para todas as caixas ao redor do objeto, encontra aquela com a probabilidade máxima de previsão e elimina as outras caixas. O lado esquerdo da Figura 28 mostra a imagem com as 4 caixas geradas na etapa anterior e do lado direito a melhor caixa mais significante selecionada após a aplicação do NMS.

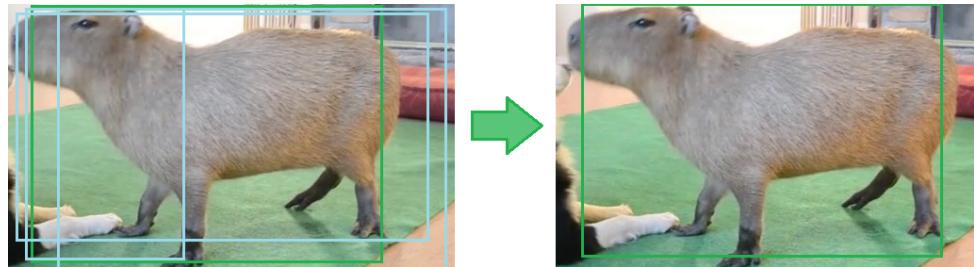


Figura 28: Após a aplicação do NMS, apenas a caixa que melhor reconhece o objeto permanece sendo utilizada.

O algoritmo do NMS primeiro descarta todas as caixas delimitadoras que tenham previsões inferiores a um certo valor do limiar para detecção. Essas caixas serão excluídas do output do modelo e nunca serão usadas como caixas detectoras de objetos. Em seguida, a partir das caixas que não foram excluídas, selecciona-se aquela com maior probabilidade de conter objeto. A caixa selecionada é retornada como detecção do objeto e as outras caixas que apresentem interseção alta com a caixa selecionada serão excluídas desde que estejam prevendo a mesma classe. Repete-se o processo até que não seja possível excluir mais caixas. Essa métrica de sobreposição de caixas utilizada no NMS é chamada de

interseção sobre a união ou IoU.

Métricas de avaliação

São utilizadas, de forma geral, duas principais métricas de avaliação de desempenho de modelos de detecção de objetos. *Quadros por segundo* (frames per second ou fps), que se refere à quantidade de vezes que o computador envia, no intervalo de um segundo, um “quadro” para a tela do monitor. Também pode ser interpretado como a velocidade de detecção do modelo e está altamente relacionada ao nível de complexidade da rede. A outra medida é o *mAP* (mean average precision), que diz respeito a precisão média das precisões do modelo na detecção de objetos pertencentes a cada uma das classes de interesse do modelo. Os valores do mAP são uma porcentagem, variando de 0 até 1, onde quanto maior o valor, mais precisas as detecções.

4.2 Redes totalmente convolucionais

Um aspecto importante das redes de detecção de objetos é que elas devem ser totalmente convolucionais [22]. Esse tipo de rede neural não contém nenhuma camada MLP, tipicamente encontrada após as camadas convolucionais e antes de retornar a previsão. As redes totalmente convolucionais têm duas vantagens principais:

- É mais rápido, uma vez que contém apenas operações convolucionais;
- Diferente de uma camada densa que espera uma entrada com tamanho específico, uma camada totalmente convolucional é capaz de aceitar imagens de qualquer resolução como input, desde que a imagem caiba na memória.

4.3 Caixas de ancoragem

Os algoritmos de detecção, como apresentado anteriormente, usualmente amostram um grande número de regiões da imagem de entrada, determina se essas regiões contêm objetos de interesse, e ajusta as fronteiras das regiões a fim de prever as caixas delimitadoras dos objetos de forma mais acurada. Diferentes modelos podem adotar diferentes esquemas de amostragem. Neste capítulo será introduzido um método capaz de gerar múltiplas caixas com escalas e proporções variáveis centradas em cada pixel. Esse tipo de caixas delimitadoras são chamadas de *caixas de ancoragem* (ou *bounding boxes*).

Sobreposição de objetos

Pode-se observar na Figura 29 que os centros do cachorro e do cavalo estão quase no mesmo lugar e ambos caem na mesma célula da grade. Então, para essa célula, y será um vetor $y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$, onde p_c descreve a existência ou não do objeto, utilizando valores 1 ou 0 respectivamente, b_x, b_y, b_h, b_w descrevem a caixa ao redor do objeto, sendo b_x, b_y os pontos que definem o centro da caixa e b_h, b_w as dimensões da caixas, e c_1, c_2, c_3 definem a classe do objeto. No exemplo a seguir estamos considerando 3 classes: capirava, cachorro e equino, e por isso temos c_i , para $i = 1, 2, 3$.

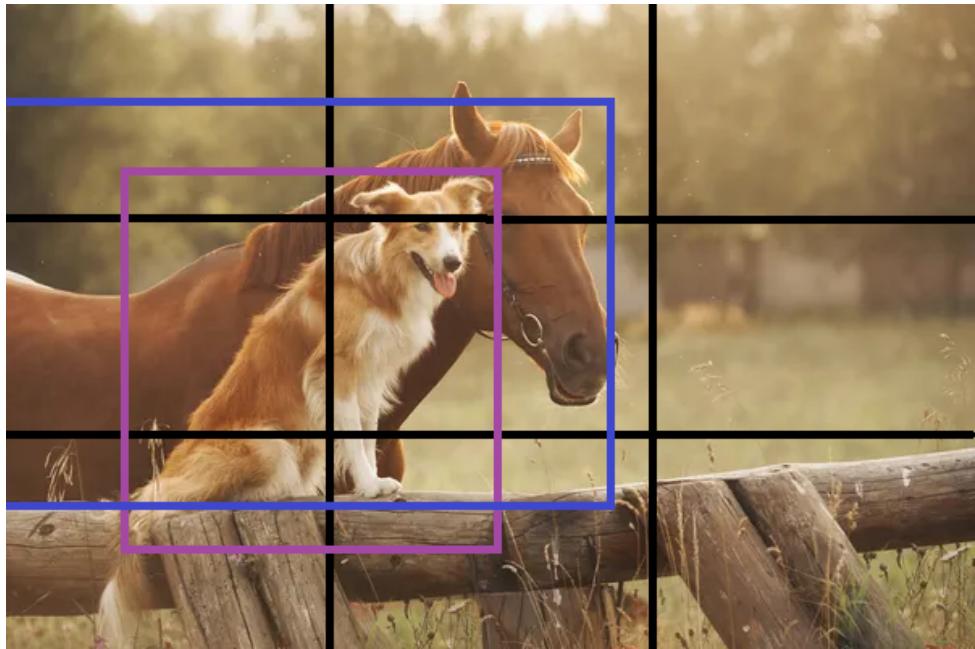


Figura 29: Dois objetos se sobrepondo na mesma imagem

O algoritmo normalmente não seria capaz de gerar duas detecções, ele teria que escolher e retornar a classe mais provável. Para ser capaz de realizar múltiplas previsões para os objetos na imagem, são definidos dois formatos diferentes para as caixas, que serão chamadas de *caixas de ancoragem*. O número de caixas de ancoragem dependerá da quantidade de objetos detectados na imagem. De forma semelhante à forma como um único objeto é detectado na célula, também define-se um vetor que fornece informações sobre a posição e classe da objeto. Neste caso, o vetor será igual ao vetor y descrito anteriormente, mas com o dobro de entradas: $y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$, onde as primeiras 8 entradas estão associadas com a primeira caixa de ancoragem, e as outras 8 entradas estão associadas com a segunda caixa.

De forma resumida, como o formato do cachorro é mais similar ao formato da caixa de ancoragem roxa, usamos os primeiros 8 valores do vetor para descrever essa classe.

Da mesma forma, como a caixa do cavalo é mais similar ao formato da caixa de ancoragem azul, utilizamos os outros 8 valores para descrever esse objeto. O vetor neste exemplo teria os valores $y = [1, b_x, b_y, b_h, b_w, 1, 0, 0, 1, b_x, b_y, b_h, b_w, 0, 1, 0]$. Caso só houvesse um dos objetos, por exemplo se só existisse o cachorro na imagem, o vetor seria $y = [1, b_x, b_y, b_h, b_w, 1, 0, 0, 0, ?, ?, ?, ?, ?, ?, ?, ?]$. As entradas marcadas com ponto de interrogação (?) na prática recebem algum valor numérico arbitrário. Tais valores não são utilizados no cálculo da função de perda. Em outras palavras, a função de perda é calculada de uma forma quando a rede produz caixa para um objeto que existe e de uma segunda forma quando produz previsões para objetos que não existem.

Algoritmo

No caso de uma caixa de ancoragem com apenas um objeto, ao objeto na imagem de treino é associado uma grade de células que contém o centro do objeto. O output y terá dimensão 3x3x8 (no caso de uma grade com 9 células). Para duas caixas de ancoragem, a imagem de treino será associada ao mesmo formato de grade anterior contendo os pontos centrais dos objetos e à caixa de ancoragem para a célula da grade com a maior interseção sobre a união (IoU) com o formato do objeto. Esse objeto é associado a um par (célula da grade, caixa de ancoragem). Neste caso, o output y terá o formato 3x3x16 (ou 3x3x2x8).

4.4 You Only Look Once (YOLO)

YOLO[23] é uma classe de modelos de aprendizado profundo com o objetivo de realizar detecção de objetos de forma rápida, sendo inclusive capaz de performar previsões em tempo real. Apesar da sua acurácia, em geral, não ser tão alta quanto a de modelos da família R-CNN [18] (modelos que seguem a estrutura apresentada no começo do capítulo), tornou-se bastante popular pela sua velocidade de detecção advinda do fato de que a rede produz previsões com apenas uma leitura da imagem.

Diferente da ideia de detecção apresentada inicialmente, o YOLO não passa por uma etapa em que regiões propostas são filtradas nas imagens. Ao invés disso, a imagem é dividida em uma grade regular separando a imagem em células menores (painel à esquerda da Figura 30), onde cada uma prevê uma classe e um número fixo de caixas de detecção (duas no caso da primeira versão do YOLO). Isso resulta em um grande número de caixas delimitadoras, que resultam em uma previsão final utilizando non-maximum suppression.

Durante a etapa de treinamento, utiliza-se o mesmo conceito das caixas de ancoragem. Dado um certo número de classes de interesse e um número de caixas de ancoragem

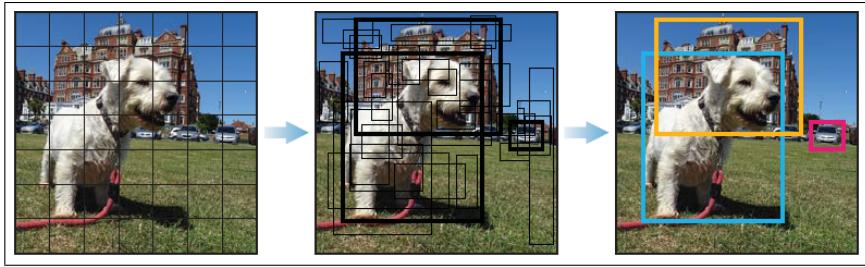


Figura 30: O YOLO divide a imagem, prevê os objetos para cada uma dessas divisões e usa o NMS na última etapa. [11]

arbitrário, a imagem de entrada é dividida em uma grade (no exemplo apresentado anteriormente foi sugerida uma grade 3×3 , com 3 classes de interesse e 2 caixas de ancoragem), e um vetor y é associado a cada uma das células geradas por essa grade (no exemplo esse vetor tem dimensão $3 \times 3 \times 2 \times 8$, sendo 3×3 a dimensão da grade, 2 o número de caixas de ancoragem, e 8 os elementos do vetor para cada caixa). O algoritmo receberá uma imagem de qualquer dimensão, essa imagem passará por uma rede neural convolucional, e retornará um vetor nas dimensões de interesse (no exemplo, a entrada são imagens de dimensão $100 \times 100 \times 3$, e o algoritmo retornaria um vetor $3 \times 3 \times 2 \times 8$).

A parte final do treinamento utiliza o mecanismo de non-maximum suppression. Caso estejam sendo utilizados 2 caixas de ancoragem, como no exemplo, então para cada uma das 9 células da grade teremos duas caixas de ancoragem preditas para cada célula. Algumas com baixa e algumas com alta probabilidade. A seguir, serão eliminadas as caixas com menor probabilidade de previsão. Finalmente, para um número qualquer de classes que estamos tentando detectar, neste caso 3, roda-se de forma independente (uma vez para cada classe) o non-maximum suppression para gerar as previsões finais. A saída será a detecção das classes na imagem.

Versões do YOLO

A primeira versão do YOLO[23] foi escrita em uma estrutura flexível chamada *Darknet*[24]. O YOLO original foi a primeira rede de detecção que combinou o problema de desenhar as caixas delimitadoras e identificar as classes em uma rede diferenciável de ponta a ponta numa única passagem pela rede. A segunda versão YOLOv2[25] incluiu melhorias como normalização de lote (*BatchNorm*[26]), maiores resoluções, e caixas de ancoragem. A terceira versão YOLOv3[27], construída sobre os modelos anteriores, adicionou um score sobre as previsões das caixas delimitadoras, adicionou conexões às camadas gerais da rede, e passou a fazer previsões em três níveis separados de granulação para que a performance em objetos pequenos fosse melhorada. A quarta versão YOLOv4[20] é a versão utilizada

neste trabalho. Ela introduz novos conceitos como a agregação de características, *Bag-of-Freebies* e *Bag-of-Specials*.

YOLOv4

A arquitetura do YOLOv4 é composta de 4 blocos distintos como pode ser visto na Figura 31). *espinha dorsal* (ou backbone), o *pESCOÇO* (ou neck), a *previsão densa* (ou dense prediction), e a *previsão esparsa* (ou sparse prediction).

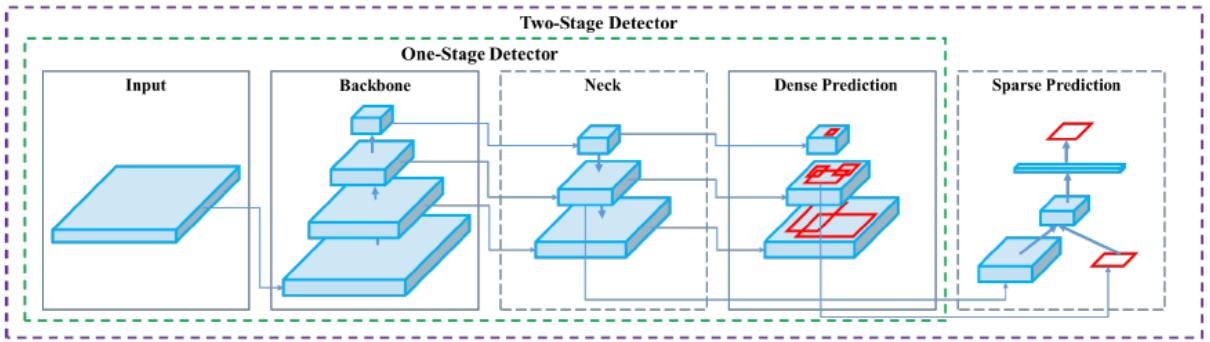


Figura 31: Detector do YOLOv4 - Fonte: [20]

A *espinha dorsal* é uma rede tipicamente pré-treinada nas classificações do ImageNet. Pré-treinar essa rede significa que os pesos da rede já foram adaptados para identificar características relevantes na imagem, no entanto, eles serão alterados para a tarefa de detecção. É a parte da arquitetura responsável pela extração de características e utiliza o CSPDarknet53[28]. O CSPDarknet53 utiliza conexões Cross-Spatial-Partial, que é utilizado para dividir a camada em duas partes, uma passa pelas camadas convolucionais e a outra que não passa pelas convoluções, e ao final os resultados são agregados por concatenação.

O *pESCOÇO* da arquitetura ajuda a adicionar camadas entre a *espinha dorsal* e a *cabeça* (bloco de *previsão densa*). Neste etapa as características extraídas nas camadas convolucionais da *espinha dorsal* são combinadas para preparar para a etapa de detecção.

O bloco de *previsão denso*, ou a *cabeça* da arquitetura, é responsável por localizar as caixas delimitadoras e pela classificação. As coordenadas x, y das caixas, assim como sua altura e largura, são detectadas e um score é atribuído. A saída será um vetor contendo as coordenadas da caixa e a probabilidade de cada classe.

Para além da arquitetura do YOLOv4, o algoritmo também utiliza outras técnicas para melhorar a performance e acurácia do treinamento. Mais especificamente o *Bag of Freebies*, que usa técnicas de data augmentation e dropout, e o *Bag of Specials*, que envolve métodos como o non-maximum suppression.

O bag of freebies ajuda no treinamento sem aumentar o tempo de inferência. Possui duas técnicas, o primeiro sendo o bag of freebies na espinha dorsal, que utiliza data augmentation e regularização (ver Figura 32), e o segundo o bag of freebies para detecção, que adiciona a espinha dorsal outras formas de treinamento.

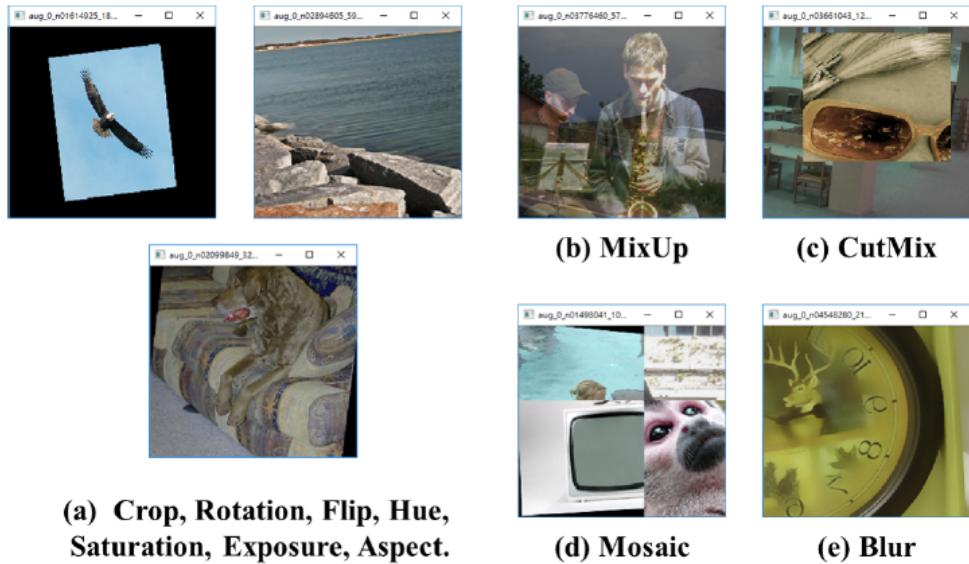


Figura 32: Diferentes métodos de data augmentation - Fonte: [20]

O bag of specials muda a arquitetura e aumenta o tempo de inferência em uma pequena parcela. Também possui duas técnicas, a primeira sendo o bag of specials para a espinha dorsal, que usa função de ativação mish [29], conexões cross-stage partial, e a segunda é a bag of specials para detecção, que usa alguns blocos de path-aggregation.

4.5 Aplicação

Para o treinamento do modelo, foram utilizadas 3237 imagens de capivaras, cachorros e equinos (cavalos, burros, mulas e asnos), cada classe com 1079 imagens. O conjunto de validação foi composto de 618 imagens das mesmas classes, também dividido de forma balanceada. Para a obtenção das imagens foram utilizados Open Images[30] do Google e um método de scapping do Google Imagens utilizando o pacote Selenium. O software LabelImg[31] foi utilizado para marcar a posição dos animais nas imagens. Todas as anotações das posições das capivaras nas imagens foram feitas manualmente.

O YOLOv4 foi escolhido como a rede neural convolucional para o treinamento do modelo e os pesos pré-treinados foram treinados no conjunto de dados MSCOCO[2]. A estrutura usada para o treinamento foi o Darknet[24]. O YOLOv4 foi treinado em nuvem

utilizando o Google Colaboratory (GPU Nvidia Tesla k80 24GB), e o modelo foi treinado para 6000 épocas.

Obtenção e manipulação das imagens

As imagens de equinos e cachorros foram facilmente obtidas utilizando o Open Images. Esse banco de dados disponibiliza imagens já anotadas com as classes e os objetos já marcados com caixas delimitadoras (Figura 33). No entanto, as classes disponíveis são limitadas e as imagens das capivaras tiveram que ser coletadas a partir de um algoritmo desenvolvido no Python: o algoritmo acessa o Google Imagens, realiza a busca que estamos interessados e baixa uma quantidade determinada de imagens resultantes da busca.



Figura 33: Imagens de cachorro já marcadas do Open Images

As imagens obtidas pelo Google Imagens não possuem marcações indicando as posições das capivaras. O LabelImg permite que o usuário crie múltiplas caixas sobre as imagens e, para cada objeto marcado com uma dessas caixas (Figura 34), as coordenadas e classes de cada objeto são atribuídos a uma linha de um arquivo de texto com o mesmo nome da imagem original.

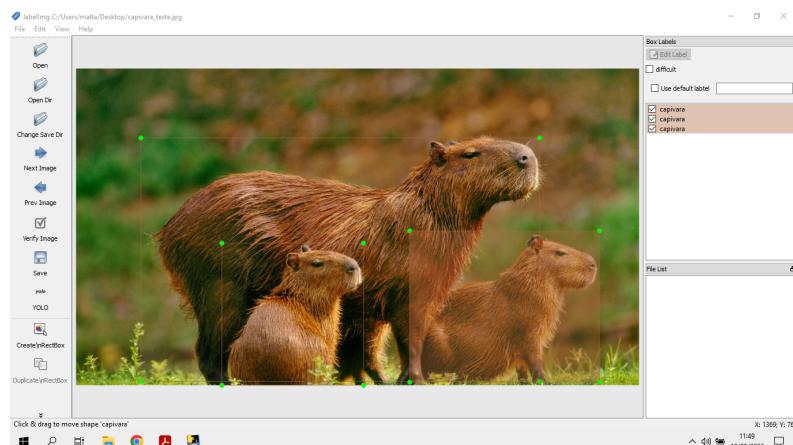


Figura 34: Marcando objetos na imagem com LabelImg

Após o processo de obtenção das imagens foram necessárias algumas modificações no formato das imagens. As dimensões foram alteradas para 416x416, apesar de não ser uma necessidade, dado que o próprio algoritmo de treinamento do YOLOv4 redimensiona as imagens para esse mesmo valor. Além disso, imagens em formato diferente de “.JPG” não são aceitas e, por isso, todas as imagens do conjunto de dados foram transformadas para esse formato.

Google Colab e Darknet

O Google Colab é um serviço da Google que permite escrever, rodar e compartilhar códigos do Python pelo navegador, e garante acesso gratuito a uma GPU do Google. Durante o período de realização desse trabalho o Google Colab disponibiliza uma GPU NVIDIA Tesla K80 com 12GB de memória, e que pode ser usada continuamente por até 12 horas. Durante esse período de utilização, caso o fluxo de usuários no servidor esteja muito elevado, o tempo de utilização pode ser reduzido e, ao fim dele, o usuário é removido e todos os dados são perdidos. Apesar disso, esse serviço é extremamente necessário para a realização do trabalho, dado que a máquina pessoal disponível tem uma capacidade de processamento inferior ao necessário para completar o treinamento em tempo hábil.

O Darknet é uma estrutura de redes neurais em código aberto, escrito em C e CUDA. Ele permite utilizar tanto computações com GPU quanto com CPU, e pode ser obtido no repositório <https://github.com/AlexeyAB/darknet>.

Resultados

A precisão média do modelo de 1000 até 6000 épocas pode ser visto na Tabela 5. Para as primeiras 1000 épocas as precisões para as classes *Capivara* e *Cachorro* já são superiores a 85%, indicando um bom desempenho. A classe *Equino* possui as piores precisões durante todas as épocas, alcançando um máximo de 82.47% na época 4000 e 81.41% em 6000 iterações. Ao final das épocas as precisões de *Capivara* e *Cachorro* chegam a 91.74% e 93.81% respectivamente.

Tabela 5: Precisão média (AP) por classe

	1000	1500	2000	3000	4000	5000	6000
Capivara	88.78%	90.96%	91.39%	93.16%	92.11%	91.23%	91.74%
Equino	69.99%	78.89%	81.13%	76.04%	82.47%	81.32%	81.41%
Cachorro	86.72%	93.63%	94.01%	91.49%	90.37%	93.60%	93.81%

A Tabela 6 mostra o desempenho geral do modelo utilizando como critério de avaliação a medida de precisão geral (mAP), que indica quão acertivo é o modelo. O mAP para

1000 as primeiras 1000 épocas é 81.83% e atinge seu valor mais ao final das 6000 iterações com 88.99%. Pode-se perceber também um aumento no número de verdadeiros positivos (TP), e redução dos falsos positivos (FP) e falsos negativos (FN) conforme o número de épocas aumenta. A Figura 35 mostra graficamente o comportamento dos verdadeiros positivos (TP) e falsos positivos (FP) de cada classe para as diferentes épocas.

Tabela 6: Performance geral do modelo

	Precisão	Recall	F1	TP	FP	FN	mAP
1000 épocas	0.70	0.84	0.77	711	299	132	81.83%
1500 épocas	0.79	0.86	0.82	727	194	116	87.83%
2000 épocas	0.79	0.87	0.82	732	200	111	88.84 %
3000 épocas	0.81	0.86	0.83	726	175	117	86.90%
4000 épocas	0.83	0.87	0.85	732	150	111	88.32%
5000 épocas	0.83	0.88	0.85	741	154	102	88.72%
6000 épocas	0.84	0.88	0.86	742	144	101	88.99%

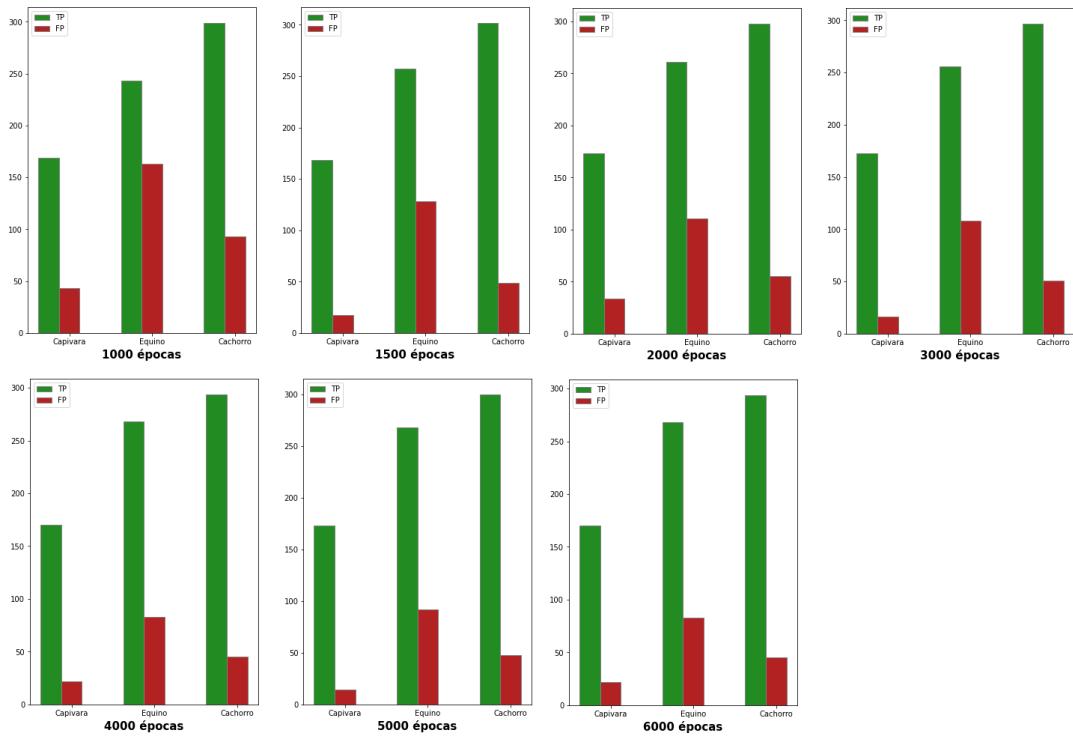


Figura 35: Verdadeiras positivas (TP) e falsos positivos (FP) em diferentes épocas

Quando aplicado em imagens, o modelo apresenta alta precisão na detecção dos animais nos casos em que há apenas uma classe na imagem. Isso inclui casos em que há múltiplos cachorros, capivaras ou equinos na mesma imagem.

A Figura 36 mostra casos em que as detecções são aplicadas para a classe dos cachorros. O modelo é capaz de detectar os objetos tanto em casos simples, como os da primeira

linha, quanto em casos com vários objetos como os da segunda linha. É interessante notar também que o modelo é capaz de identificar diferentes raças como a classe cachorro. Como os pesos pré-treinados utilizados no treinamento do modelo foram obtidos em um conjunto de dados que utilizava fotos de cachorro, é possível que o bom desempenho esteja associado a esse fato.

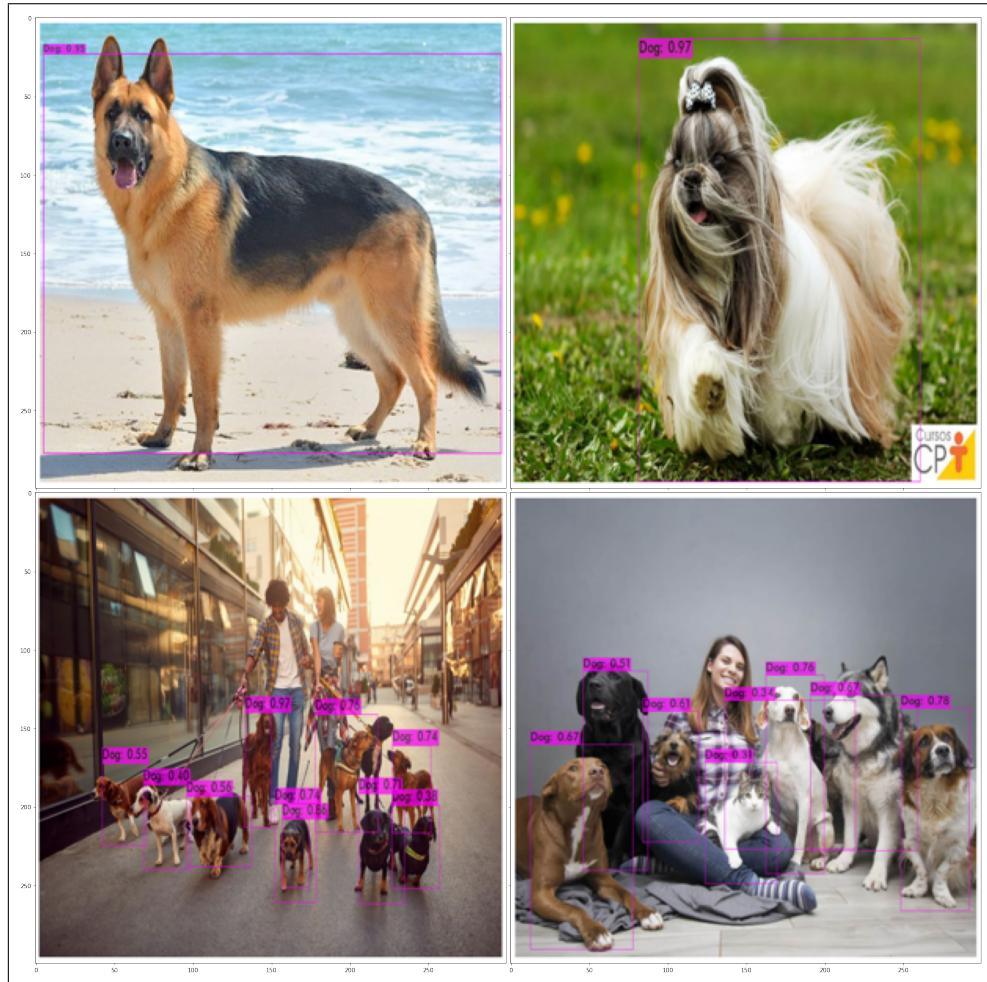


Figura 36: YOLOv4 sendo utilizado para classificar imagens de cachorros.

A classe dos equinos, apesar de ter tido a pior performance utilizando o critério da precisão média (AP), tem um bom desempenho para a maioria das imagens em que o corpo do animal parece completo na imagem, como pode ser visto na Figura 37. Para alguns casos, mesmo imagens descentralizadas e com o objeto pequeno, o modelo desempenha de forma satisfatória. O modelo passa a desempenhar pior quando temos imagens que mostram apenas parte do animal, como por exemplo: imagens onde aparece apenas a cabeça do animal, ou imagens onde as patas aparecem encoberta por algum outro objeto (detectável ou não detectável). Uma possível explicação para o primeiro problema talvez seja o fato do conjunto de dados incluir diferentes animais para a classe *equino*. Essa classe

incluia tanto cavalos, quanto mulas, burros, e animais similares. Já o segundo problema talvez esteja sendo causado pela natureza do conjunto de treinamento, que é composto, em sua maioria por imagens de corpo inteiro dos animais.

É importante apontar que como as imagens utilizadas na composição do conjunto de treinamento foram obtida através de uma extração de dados indiscriminado do Google, é possível que a inclusão de abstrações, como quadros e desenhos, estejam prejudicando o desempenho do modelo para alguns casos.



Figura 37: YOLOv4 sendo utilizado para classificar imagens de equinos.

Para os casos em que o modelo tenta detectar capivaras, o modelo se comporta de forma irregular. Ele apresenta um bom desempenho em imagens onde o formato dos corpos das capivaras aparecem bem definido e imagens onde elas aparecem de frente. Nos casos em que os animais estão longe, não centralizados, ou a imagem não está bem focada, a classificação atribuída varia entre *capivara* e *cachorro*, e em raros casos *equino*. A Figura 38 mostra um dos problemas apresentados: enquanto a maior parte das capivaras

é detectada com precisão no meio da imagem, a capivara mais a direita, e em menor foco, é classificada de forma errada.

A classe das capivaras apresenta a particularidade de ser a única que não está presente nos pesos pré-treinados do MSCOCO. Isso pode ser uma explicação para o confundimento da detecção em alguns casos. Além disso, assim como no caso dos equinos, a natureza da obtenção das imagens permitiu que desenhos e figuras estivessem incluídas no conjunto final, o que pode ter diminuído a eficiência da detecção.



Figura 38: YOLOv4 sendo utilizado para classificar imagens de capivaras.

Para os casos em que mais de uma classe aparece na mesma imagem o modelo também parece funcionar bem, porém apresentando as mesmas deficiências das imagens com apenas uma classe. Apesar de incomum, o modelo falhou em classificar casos em que os animais aparecem sobrepostos, desenhando a caixa ao redor de apenas um deles. A Figura 39 mostra uma classificação com alta precisão de duas das classes.

O modelo também foi testado em vídeos. Alguns desses teste estão disponíveis nos links <https://youtu.be/T019RpEq44>, https://youtu.be/yS5NTN_Hd0g e <https://youtu.be/v8HjYRd96XA>. O primeiro vídeo mostra capivaras em várias posições e profundidades, e o modelo é capaz de realizar a detecção na maior parte do tempo. Ele apresenta dificuldade em realizar a detecção quando as capivaras estão longe do centro do

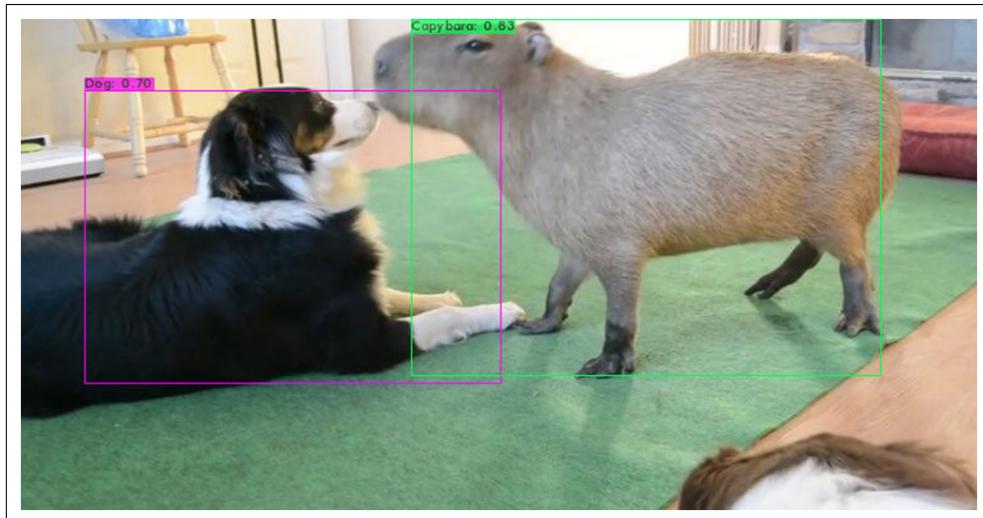


Figura 39: Duas classes distintas em uma mesma imagem: o algoritmo é capaz de determinar caixas ao redor dos diferentes animais, atribuindo a cada um deles uma probabilidade para a classe mais provável.

video ou não aparecem totalmente na filmagem. As caixas delimitadoras compreendem apenas a parte das capivaras que está acima da água e a parte abaixo da agua, ainda que visível, não é capturada na detecção. Isso parece indicar que marcações desse tipo são raras no conjunto de treino. No segundo vídeo o desempenho do modelo também foi muito bom. Ele é capaz de detectar tanto os cachorros quanto o cavalo de forma correta durante quase toda duração. No último vídeo o modelo falha na tarefa de classificação, mas executa bem a tarefa de detecção dos animais. Isto é, apesar de apontar a classe errada do animal, ele ainda é capaz de detectar o objeto como uma das classes e gerar uma caixa para ele. As capivaras são detectadas consistentemente apenas nos quadros em que os animais estão estáticos. Nos outros quadros o modelo classifica a capivara com quase 100% de certeza como a classe cachorro. O cachorro parece ser bem detectado na maior parte do video, mas quando se distancia do ponto da filmagem passa a ser classificado como equino. O mesmo ocorre com a capivara nesta situação.

5 Conclusão e trabalhos futuros

Esse trabalho estudou a aplicação de transferência de aprendizado no contexto de classificação de imagens, utilizando a arquitetura Xception com os pesos treinados no ImageNet para demonstrar os benefícios na performance e resultado quando comparado com um modelo treinado do zero.

Foram estudados os modelos YOLO para detecção de objetos e, assim como para classificação, a transferência de aprendizado foi utilizada para otimizar a performance durante o treinamento. O modelo apresentou alta precisão na detecção dos objetos testados, atingindo mAP de 88.99% ao final do treinamento e precisão média de 91.74% para Capivaras, 81.41% para Equinos e 93.81% para Cachorros. Testes deste modelo foram realizados em imagens e vídeos, apresentando resultados bons na maioria dos casos.

No futuro, seria interessante incluir no treinamento imagens que não contenham nenhuma das classes para reduzir o número de falsos positivos, incluir imagens com mais de uma classe no conjunto de treinamento, e realizar formas de data augmentation mais elaboradas para aumentar a chance do modelo detectar animais em posições diferentes e em profundidades diferentes da imagem. Investigar outras versões mais recentes do YOLO ou outros modelos também é uma tarefa interessante.

Lista de Figuras

1	Threshold Logic Unit - Fonte: [5].	p. 6
2	Perceptron com 2 neuróns de input, 1 neurón de viéis, e 3 neuróns de output - Fonte: [5]	p. 7
3	Multilayer Perceptron - Fonte: [5]	p. 9
4	Funções de ativação - Fonte: [5]	p. 10
5	Etapas do Método dos Gradientes	p. 12
6	Gradiente descendente utilizando o método dos momentos x Gradiente descendente regular - Fonte: [7]	p. 14
7	AdaGrad vs. Gradiente Descendente - Fonte: [5]	p. 16
8	MLP de classificação com saída softmax - Fonte: [5]	p. 18
9	Arquitetura de uma CNN - Fonte: [11]	p. 19
10	Campos receptivos de vários neuróns - Fonte: [11]	p. 20
11	Filtro convolucional 3x3 deslizando sobre uma imagem de entrada - Fonte: [11]	p. 21
12	Camadas convolucionais com múltiplos mapas de características e imagem com três canais de cores - Fonte: [5]	p. 22
13	Zero padding e redução de dimensionalidade com stride de 2 - Fonte: [5]	p. 23
14	Camadas de max e average pooling, kernel 2x2, stride 2 e sem padding - Fonte: [5]	p. 24
15	Módulo Inception com stride igual a 1 - Fonte: [5]	p. 27
16	Arquitetura do GoogLeNet - Fonte: [5]	p. 28
17	Arquitetura do VGG - Fonte: [15]	p. 29

18	1) Módulo inception com a convolução 5x5 dividida em duas 3x3; 2) Módulo inception após fatorização das convoluções em $1 \times n$ e $n \times 1$. - Fonte: [16]	p. 30
19	Classificador auxiliar - Fonte: [16]	p. 31
20	Arquitetura do Xception. Os dados entram, passando pela estrutura do lado esquerdo. Em seguida, passam pela estrutura do meio 8 vezes, e finalmente pela estrutura de saída na direita. - Fonte: [17]	p. 32
21	Utilizando camadas pré-treinadas. - Fonte: [5]	p. 33
22	Exemplo de data augmentation utilizando uma das imagens de capivara do conjunto de teste.	p. 35
23	Gráficos de perda e acurácia para o modelo Xception usando transfer learning.	p. 36
24	Gráficos de perda e acurácia para o modelo Xception sem usar transfer learning.	p. 37
25	Gráficos de perda e acurácia para o modelo com apenas camadas MLP.	p. 38
26	Regiões de interesse com alto score representam regiões com grandes chances de conterem os objetos de interesse.	p. 42
27	O detector produz várias caixas delimitadoras para um objeto.	p. 43
28	Após a aplicação do NMS, apenas a caixa que melhor reconhece o objeto permanece sendo utilizada.	p. 43
29	Dois objetos se sobrepondo na mesma imagem	p. 45
30	O YOLO divide a imagem, prevê os objetos para cada uma dessas divisões e usa o NMS na última etapa. [11]	p. 47
31	Detector do YOLOv4 - Fonte: [20]	p. 48
32	Diferentes métodos de data augmentation - Fonte: [20]	p. 49
33	Imagens de cachorro já marcadas do Open Images	p. 50
34	Marcando objetos na imagem com LabelImg	p. 50
35	Verdadeiras positivas (TP) e falsos positivos (FP) em diferentes épocas	p. 52
36	YOLOv4 sendo utilizado para classificar imagens de cachorros.	p. 53

37	YOLOv4 sendo utilizado para classificar imagens de equinos.	p. 54
38	YOLOv4 sendo utilizado para classificar imagens de capivaras.	p. 55
39	Duas classes distintas em uma mesma imagem: o algoritmo é capaz de determinar caixas ao redor dos diferentes animais, atribuindo a cada um deles uma probabilidade para a classe mais provável.	p. 56

Lista de Tabelas

1	Classificação de performance no ImageNet [17]	p. 33
2	Classificação de performance no ImageNet	p. 36
3	Classificação de performance no ImageNet	p. 38
4	Classificação de performance no ImageNet	p. 39
5	Precisão média (AP) por classe	p. 51
6	Performance geral do modelo	p. 52

Referências

- [1] 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.
- [2] LIN, T.-Y. et al. Microsoft coco: Common objects in context. In: SPRINGER. *European conference on computer vision*. [S.l.], 2014. p. 740–755.
- [3] LECUN, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, Ieee, v. 86, n. 11, p. 2278–2324, 1998.
- [4] KRIZHEVSKY, A.; HINTON, G. et al. Learning multiple layers of features from tiny images. Citeseer, 2009.
- [5] GÉRON, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. [S.l.]: "O'Reilly Media, Inc.", 2019.
- [6] WERBOS, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, IEEE, v. 78, n. 10, p. 1550–1560, 1990.
- [7] GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep learning*. [S.l.]: MIT press, 2016.
- [8] DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, v. 12, n. 7, 2011.
- [9] TIELEMANS, T.; HINTON, G. Rmsprop adaptive learning. *Neural Networks for Machine Learning, COURSERA*, 2012.
- [10] KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [11] ELGENDY, M. *Deep learning for vision systems*. [S.l.]: Simon and Schuster, 2020.
- [12] SZEGEDY, C. et al. Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 1–9.
- [13] ARORA, S. et al. Provable bounds for learning some deep representations. In: PMLR. *International conference on machine learning*. [S.l.], 2014. p. 584–592.
- [14] SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [15] FERGUSON, M. et al. Automatic localization of casting defects with convolutional neural networks. In: . [S.l.: s.n.], 2017. p. 1726–1735.

- [16] SZEGEDY, C. et al. Rethinking the inception architecture for computer vision. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2016. p. 2818–2826.
- [17] CHOLLET, F. Xception: Deep learning with depthwise separable convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2017. p. 1251–1258.
- [18] GIRSHICK, R. et al. Rich feature hierarchies for accurate object detection and semantic segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2014. p. 580–587.
- [19] LIU, W. et al. Ssd: Single shot multibox detector. In: SPRINGER. *European conference on computer vision*. [S.l.], 2016. p. 21–37.
- [20] BOCHKOVSKIY, A.; WANG, C.-Y.; LIAO, H.-Y. M. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.
- [21] UIJLINGS, J. R. et al. Selective search for object recognition. *International journal of computer vision*, Springer, v. 104, n. 2, p. 154–171, 2013.
- [22] LONG, J.; SHELHAMER, E.; DARRELL, T. Fully convolutional networks for semantic segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 3431–3440.
- [23] REDMON, J. et al. You only look once: Unified, real-time object detection. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2016. p. 779–788.
- [24] REDMON, J. *Darknet: Open Source Neural Networks in C*. 2013–2016. <http://pjreddie.com/darknet/>.
- [25] REDMON, J.; FARHADI, A. Yolo9000: better, faster, stronger. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2017. p. 7263–7271.
- [26] IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: PMLR. *International conference on machine learning*. [S.l.], 2015. p. 448–456.
- [27] REDMON, J.; FARHADI, A. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.
- [28] WANG, C.-Y. et al. CspNet: A new backbone that can enhance learning capability of cnn. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. [S.l.: s.n.], 2020. p. 390–391.
- [29] MISRA, D. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, CoRR, v. 4, n. 2, p. 10–48550, 2019.
- [30] KUZNETSOVA, A. et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

- [31] TZUTALIN. *LabelImg*. 2015. Free Software: MIT License. Disponível em: <<https://github.com/tzutalin/labelImg>>.