

# Notas de Aula da Disciplina de Visão Computacional

Cesar Henrique Comin

Departamento de Ciência da Computação

Universidade Federal de São Carlos

Neste documento estão descritos conceitos importantes para o entendimento dos códigos da disciplina. Cada seção possui o conceito apresentado e entre parênteses o módulo (códigos) ao qual o conceito está associado. Seções e notas de rodapé começando com \* são conceitos mais avançados ou que podem ser ignorados em uma primeira leitura. A Seção 1.1 pode ser lida de forma independente da Seção 1.2. Várias figuras utilizadas foram obtidas do livro *Understanding Deep Learning* do Simon J.D. Prince.

# Sumário

<b>1</b>	<b>Conceitos de aprendizado de máquina (módulos 2 e 3)</b>	<b>2</b>
1.1	Conceitos de modelagem probabilística e otimização . . . . .	2
1.2	Conceitos de cálculo diferencial . . . . .	12
1.3	*Regularização e weight decay (módulo 5) . . . . .	23
1.4	Figuras sobre Regressão Linear e Logística . . . . .	25
<b>2</b>	<b>Figuras sobre redes neurais convolucionais (módulos 4 a 8)</b>	<b>28</b>
2.1	Convolução . . . . .	28
2.2	Arquiteturas de CNNs . . . . .	30
2.3	Redes Residuais (ResNets) . . . . .	33
2.4	Decodificadores . . . . .	35
<b>3</b>	<b>Mecanismo de atenção e transformers (módulo 9)</b>	<b>37</b>
3.1	Figuras sobre implementação de atenção . . . . .	42
<b>4</b>	<b>Figuras sobre processamento multimodal (módulo 10)</b>	<b>45</b>
<b>A</b>	<b>Apêndices</b>	<b>48</b>
<b>A</b>	<b>Símbolos</b>	<b>48</b>

# 1 Conceitos de aprendizado de máquina (módulos 2 e 3)

## 1.1 Conceitos de modelagem probabilística e otimização

### 1.1.1 Função de verossimilhança

Suponha que temos um dado de seis lados (Figura 1.1) que parece estar enviesado. Como verificamos matematicamente se o dado está de fato enviesado?



Figura 1.1: Um dado enviesado?

Nosso modelo inicial, que chamaremos de  $\mathcal{M}$  será que o dado não está enviesado, ou seja, cada lado possui probabilidade  $p = 1/6$  de ocorrer. Com isso, podemos calcular o seguinte: dado o modelo  $\mathcal{M}$ , qual a probabilidade  $\mathcal{P}$  de obter uma sequência específica de valores? Vamos supor que jogamos o dado  $n$  vezes e tivemos como resultado a sequência  $s = \{2, 1, 1, 3, 5, 6, 1, 3, \dots\}$ . Podemos obter a probabilidade que essa sequência seja obtida pelo nosso modelo. Ela é dada pelo produto das probabilidades  $p_i$  de obter cada valor

$$\mathcal{P}(s) = \prod_i^n p_i \tag{1.1}$$

Segundo o nosso modelo, a probabilidade de obter cada valor é igual a  $1/6$ . Portanto, temos que

$$\mathcal{P}(s) = \prod_i^n 1/6 = \frac{1}{6^n} \tag{1.2}$$

Nesse caso o nosso modelo não possui nenhum parâmetro e não há nada interessante que podemos fazer com  $\mathcal{P}$ . Casos mais interessantes surgem quando o modelo possui parâmetros. Nesses casos,  $\mathcal{P}$  possibilita compararmos diferentes versões de um modelo, e encontrarmos o parâmetro que leva ao modelo mais provável. Para analisar essa situação, vamos considerar

que temos um novo modelo. Nesse modelo, supomos que o valor 1 ocorre com uma probabilidade diferente dos demais valores, ou seja, o valor 1 ocorre com probabilidade  $p_1$  ao invés de  $1/6$ , e os demais valores ocorrem com probabilidade  $p$ . Note que as probabilidades precisam somar 1, o que implica  $p_1 + 5p = 1$ . Portanto o modelo possui apenas 1 parâmetro, já que dado  $p_1$  temos  $p = (1 - p_1)/5$ .

Dada a mesma sequência de valores acima, temos que

$$s = \{2, 1, 1, 3, 5, 6, 1, 3, \dots\} \quad (1.3)$$

$$\mathcal{P}(s; p_1) = p \times p_1 \times p_1 \times p \times p \times p_1 \times p \dots \quad (1.4)$$

onde  $\mathcal{P}(s; p_1)$  representa a probabilidade de obter a sequência  $s$  para um modelo com parâmetro  $p_1^1$ . A probabilidade  $p_1$  aparece quando o valor 1 é obtido, e  $p$  aparece quando os valores 2 a 6 são obtidos. Podemos escrever  $\mathcal{P}(s; p_1)$  de forma mais sucinta como

$$\mathcal{P}(s; p_1) = p_1^{n_1} p^{n-n_1} \quad (1.5)$$

onde  $n_1$  representa o número de vezes que o valor 1 foi obtido.

Agora temos uma situação mais interessante! Podemos supor diferentes valores de  $p_1$  para o modelo e verificar qual valor de  $p_1$  leva ao maior valor de  $\mathcal{P}$ . Se  $\mathcal{P}$  for máximo para  $p_1 = 1/6$ , quer dizer que o dado provavelmente não é enviesado para esse valor.

Suponha que jogamos o dado  $n = 40$  vezes e obtivemos a seguinte sequência de valores:

$$s = \{2, 1, 1, 3, 5, 6, 1, 3, 3, 3, 1, 6, 4, 5, 4, 6, 6, 1, 4, 3, 5, 1, 4, 3, 2, 5, 1, 1, 4, 3, 1, 5, 2, 4, 1, 4, 1, 1, 2, 5\}$$

O valor 1 ocorre  $n_1 = 12$  vezes nessa sequência. Considerando o nosso modelo de que o valor 1 ocorre com uma probabilidade diferente dos demais, qual a probabilidade que o modelo gere essa sequência de valores? Caso o modelo esteja correto, qual o valor ideal de  $p_1$  que melhor explica os dados? Para responder essas perguntas, na Figura 1.2 é mostrado o valor da Equação 1.5 para diferentes valores de  $p_1$ .

Vemos na figura que o máximo ocorre para  $p_1 = 0.3$ . Ou seja, o modelo que melhor se ajusta aos dados obtidos é o modelo para o qual  $p_1 = 0.3$  e  $p = (1 - 0.3)/5 = 0.14$ . Portanto, o modelo indica que o dado está enviesado. Note que o valor absoluto de  $\mathcal{P}(s; p_1)$  não é relevante para a discussão, estamos interessados apenas na comparação entre valores de  $\mathcal{P}(s; p_1)$  para diferentes  $p_1$ .

Podemos fazer a mesma análise supondo diferentes probabilidades  $p_1, p_2, \dots, p_6$  para cada

<sup>1</sup>É comum a notação  $\mathcal{P}(s|p_1)$  também ser utilizada.

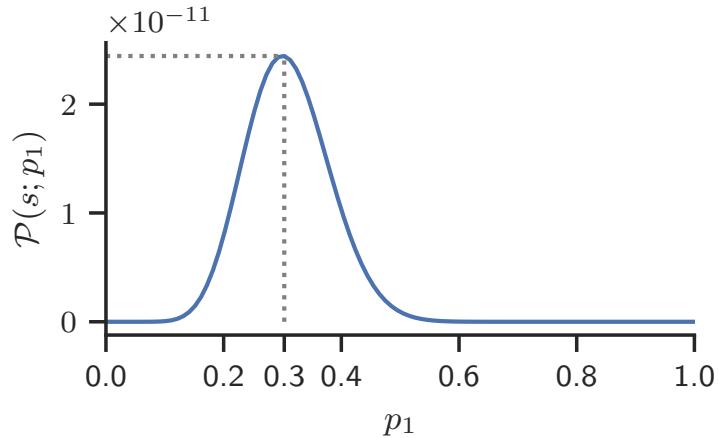


Figura 1.2: Probabilidade de obter a sequência  $s$  para diferentes probabilidades  $p_1$  do valor 1 de um dado de seis lados.

valor do dado. Nesse caso, o modelo passará a depender de 6 variáveis. É comum representarmos essas variáveis como um vetor  $\mathbf{p} = (p_1, p_2, \dots, p_6)$ . Podemos então calcular  $\mathcal{P}(s; \mathbf{p})$  para diferentes combinações de  $\mathbf{p}$  e verificar qual valor de  $\mathbf{p}$  leva ao modelo mais provável. Para um dado não enviesado o máximo de  $\mathbf{p}$  deve ocorrer para  $\mathbf{p} = (1/6, 1/6, \dots, 1/6)$ .

De forma geral, a função  $\mathcal{P}$  é chamada de função de verossimilhança, do inglês *likelihood function*. É comum essa função ser na verdade representada por  $L$  ao invés de  $\mathcal{P}$ , e os parâmetros da função são em geral representados por  $\theta$ . Um modelo  $\mathcal{M}$  é matematicamente representado por uma função  $f(x; \theta)$ <sup>2</sup> que atribui uma probabilidade para cada elemento de um conjunto de dados<sup>3</sup>. Portanto, dada uma função  $f(x; \theta)$  representando um modelo, a chance (*likelihood*) de obter um conjunto específico de valores  $s = \{x_1, x_2, \dots, x_n\}$  é dada por

$$L(s; \theta) = \prod_i^n f(x_i; \theta) \quad (1.6)$$

Em problemas de aprendizado de máquina, queremos sempre encontrar os parâmetros do modelo  $\theta$  que maximizam essa função. Note que o valor de  $n$ , a quantidade de dados, está relacionado com o nível de confiabilidade do modelo. Idealmente queremos o maior número  $n$  de exemplos possível para testar o modelo, mas em problemas reais  $n$  tende a ser limitado.

Um detalhe importante é que  $L$  é o produto de valores pequenos (entre 0 e 1) e, portanto, tende a possuir um valor extremamente baixo. Por exemplo, no caso do modelo não enviesado do dado de seis lados calculamos uma probabilidade de  $1/6^n$  (Equação 1.2). Valores pequenos

<sup>2</sup>\*Essa função pode ser uma rede neural. Nesse caso,  $\theta$  são os parâmetros da rede. Mas para a discussão desta seção não é necessário pensar em redes neurais.

<sup>3</sup>No nosso exemplo do dado de seis lados,  $f(1) = p_1, f(2) = p_2, \dots$

são problemáticos para computadores e também para interpretação dos resultados. Portanto, é extremamente comum que seja utilizado o logaritmo<sup>4</sup> da função de verossimilhança (do inglês *log-likelihood*). Como o logaritmo de produtos se torna uma soma, temos que

$$\log(L(s; \theta)) = \sum_i^n \log(f(x_i; \theta)) \quad (1.7)$$

No caso do dado de seis lados, temos que  $\log(L) = -n \log(6)$ . Outro detalhe importante é que  $f(x; \theta)$  é sempre menor ou igual a 1 (pois é uma probabilidade), mas o logaritmo de um valor entre 0 e 1 é sempre negativo (ou zero). Por isso, é comum que seja utilizado o negativo da função de verossimilhança. Com isso, finalmente podemos definir a função que usualmente é utilizada para otimizar algoritmos de aprendizado de máquina, a chamada *negative log-likelihood*. No decorrer do texto utilizaremos a notação  $\mathcal{L}$  para representar o negativo do logaritmo da função de verossimilhança, ou seja

$$\mathcal{L}(s; \theta) = - \sum_i^n \log(f(x_i; \theta)) \quad (1.8)$$

O menor valor possível para essa função é 0, que consiste no caso em que o modelo retorna o valor 1 para todos os dados.

O objetivo de algoritmos de aprendizado de máquina é encontrar um modelo que minimize o valor de  $\mathcal{L}(s; \theta)$ , ou seja, um modelo que represente bem o processo real que gerou os dados observados. Em redes neurais,  $f(x; \theta)$  é representado por uma rede com parâmetros  $\theta$  que recebe uma entrada  $x$  e retorna uma probabilidade.

Para vermos um exemplo mais concreto, vamos supor que nos foi dado um conjunto de 100 valores  $s = \{x_1, x_2, \dots, x_{100}\}$ . Ao calcular o histograma desses valores, observamos a distribuição mostrada na Figura 1.3(a)<sup>6</sup>. Essa distribuição é muito parecida com a distribuição normal, também chamada de Gaussiana, que ocorre em muitas situações em problemas reais. Essa distribuição possui a seguinte equação

$$f(x; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \quad (1.9)$$

o termo  $\theta$  representa os parâmetros da função, que nesse caso são dados por  $\theta = (\mu, \sigma)$ .  $\mu$  é a

<sup>4</sup>O logaritmo natural (na base  $e$ , que é o número Euler) é comumente utilizado, pois  $e^{\log(x)} = x$ .

<sup>5</sup>Note que dado um valor de  $\mathcal{L}(s; \theta)$ , podemos encontrar a probabilidade de obter a sequência de dados como  $\mathcal{P} = e^{-\mathcal{L}(s; \theta)}$ .

<sup>6</sup>\*Esse histograma é parecido com a curva mostrada na Figura 1.2, mas os gráficos representam conceitos distintos. Acontece que a quantia  $\mathcal{P}(s; p_1)$  pode ser modelada como uma variável aleatória, e para  $n$  suficientemente grande a distribuição de  $\mathcal{P}(s; p_1)$  tende a uma distribuição normal, o que ressalta a importância dessa distribuição.

posição do pico da distribuição e  $\sigma$  está relacionado com a largura do pico. Na Figura 1.3(b) são mostrados dois exemplos para diferentes valores de  $\mu$  e  $\sigma$ .

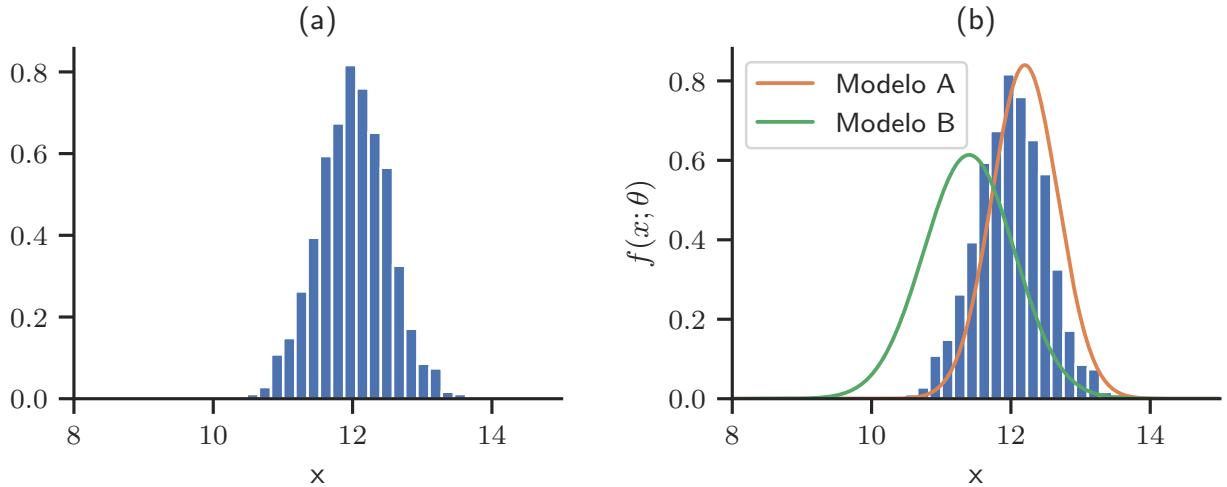


Figura 1.3: (a) Histograma de um conjunto de dados. (b) Dois possíveis modelos para a distribuição de probabilidades dos dados.

Queremos encontrar quais os valores de  $\theta$  que levam ao melhor modelo possível dos dados, ou seja, à distribuição que de fato gerou esses dados. Para isso, podemos realizar o seguinte procedimento:

1. Definimos um valor específico para  $\theta$ , que chamaremos de  $\theta^* = (\mu^*, \sigma^*)$ ;
2. Para cada valor de  $x_i$  calculamos o valor de  $f(x_i; \theta^*)$  utilizando a Equação 1.9;
3. Calculamos o valor  $\mathcal{L}$  da função de verossimilhança dado pela Equação 1.8;
4. Repetimos os itens 1 a 3 para diferentes parâmetros  $\theta^*$ , e armazenamos os parâmetros que levaram ao menor valor de  $\mathcal{L}$ .

Esse processo não é muito eficiente, teremos que testar diversos valores de  $\theta$  para encontrar um modelo apropriado. Uma opção mais eficiente é *otimizar* a função de verossimilhança para encontrar os melhores parâmetros. Um método que pode ser utilizado para isso é o *gradiente descendente*, que consiste em alterar os parâmetros de acordo com o gradiente da função. Portanto, podemos realizar o seguinte procedimento:

1. Definimos um valor inicial para  $\theta$ , por exemplo,  $\theta^* = (\mu^*, \sigma^*) = (0, 1)$ ;
2. Para cada valor de  $x_i$  calculamos o valor de  $f(x_i; \theta^*)$  utilizando a Equação 1.9;
3. Calculamos o valor da função de verossimilhança  $\mathcal{L}(s; \theta)$  (Equação 1.8);

4. Calculamos o gradiente  $\nabla_{\theta^*} \mathcal{L} = (\frac{\partial \mathcal{L}}{\partial \mu^*}, \frac{\partial \mathcal{L}}{\partial \sigma^*})$ <sup>7</sup> da função de verossimilhança;
5. Usando o método de gradiente descendente, atualizamos os parâmetros do modelo como  $\theta^* = \theta^* - lr \nabla_{\theta^*} \mathcal{L}$ , onde  $lr$  é a taxa de aprendizado;
6. Repetimos os itens 1 a 5 por um número  $E$  de passos ou até que o valor de  $\mathcal{L}$  não modifique mais.

Esse procedimento é conhecido em alguns contextos como *expectation-maximization*, pois no passo 3 calculamos o valor da função de verossimilhança (*expectation*), e nos passos 4 e 5 otimizamos a função (*maximization*, mas nesse caso estamos minimizando).

No caso da distribuição normal, é possível calcular o gradiente de forma analítica, ou seja, encontrar a equação de  $\nabla_{\theta^*} \mathcal{L}$  e utilizar ela diretamente na otimização, o que torna o processo mais rápido. Para um modelo geral, como é o caso de redes neurais, calculamos o gradiente computacionalmente.

Ainda mais interessante é o fato de que para distribuições normais o resultado do algoritmo acima também é conhecido analiticamente. É possível mostrar que os melhores parâmetros possíveis para o modelo são

$$\mu = \frac{1}{n} \sum x_i \quad (1.10)$$

$$\sigma^2 = \frac{1}{n} \sum (x_i - \mu)^2 \quad (1.11)$$

As equações 1.10 e 1.11 simplesmente representam a média e a variância dos dados.

### 1.1.2 Verossimilhança no contexto de redes neurais

Em redes neurais, a função  $f(x; \theta)$  é representada por uma sequência de camadas. Cada camada  $l$  mapeia a entrada  $x^{l-1}$  advinda da camada anterior para um novo valor  $x^l$ . A última camada da rede é especial, ela mapeia a entrada em um valor que é garantido de estar entre 0 e 1<sup>8</sup>. Portanto, a rede se comporta exatamente como a função que vimos nesta seção, ela recebe uma entrada e retorna uma probabilidade. O grande desafio está em calcular o gradiente da função. Mas desde que seja possível calcular o gradiente de cada camada, podemos utilizar a regra da cadeia (Seção 1.2.5) para calcular o gradiente de toda a rede! Com isso, temos um procedimento geral para encontrar qualquer modelo  $f(x; \theta)$  que represente um conjunto de dados  $s = \{x_1, x_2, \dots, x_n\}$ , descrito pelo Algoritmo 1.

<sup>7</sup>A Seção 1.2 explica como fazer isso.

<sup>8</sup>Utilizando uma função sigmoide ou softmax.

---

Algoritmo 1: Otimização da função de verossimilhança.

---

1. Criamos uma rede neural com  $L$  camadas. Essa rede representa uma função  $f(x; \theta)$ . Os parâmetros  $\theta$  da rede são inicializados de forma aleatória com valores  $\theta = \theta^*$ ;
  2. Aplicamos a rede em cada item  $x_i$ , obtendo uma probabilidade para cada item;
  3. Calculamos o valor da função de verossimilhança  $\mathcal{L}(s; \theta^*)$  (Equação 1.8) utilizando os resultados da rede;
  4. Calculamos o gradiente  $\nabla_{\theta^*} \mathcal{L}$  da função de verossimilhança;
  5. Usando o método de gradiente descendente, atualizamos os parâmetros do modelo como  $\theta^* = \theta^* - lr \nabla_{\theta^*} \mathcal{L}$ ;
  6. Repetimos os itens 1 a 5 por um número  $E$  de passos ou até que o valor de  $\mathcal{L}$  não modifique mais.
- 

O passo 4 é o mais custoso no processo de otimização de redes neurais. Ele é feito pelo método `.backward()` da biblioteca Pytorch.

### 1.1.3 Entropia cruzada

A seção anterior considerou a situação na qual queremos criar um modelo que represente de forma fiel o processo de criação dos dados no mundo real. Por exemplo, no caso do dado de seis lados que gera valores entre 1 e 6, o modelo consiste em criar uma função  $f(x; \theta)$  que represente o processo de geração dos valores realizado pelo dado. Focaremos agora em tarefas de classificação de dados, isto é, o processo de associar uma classe  $y_i$  para cada item  $x_i$ , sendo que a classe deve ser um valor entre 1 e  $C$ , onde  $C$  é o número de classes do problema. No caso de classificação de imagens, queremos associar uma classe  $y_i$  para cada imagem  $x_i$  de entrada. As classes podem ser, por exemplo, “cachorro”, “gato”, “cavalo”, etc, representadas pelos valores  $1, 2, 3, \dots, C$ .

Suponha que temos um conjunto de treinamento composto por uma sequência de  $n$  imagens e os respectivos rótulos  $s = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ . Nesse caso, podemos considerar que para cada imagem estamos afirmando: com probabilidade 1 a imagem  $x_i$  é da classe  $y_i$ , ou seja, temos absoluta certeza que a imagem  $x_i$  deve ser classificada pelo modelo na classe  $y_i$ <sup>9</sup>. Portanto, queremos desenvolver um modelo que receba como entrada uma

---

<sup>9</sup>\*Pode parecer exagero pensar no problema dessa forma, mas esse conceito é importante porque é possível também atribuirmos probabilidades diferentes de 1 para as classes. Por exemplo, seria possível considerarmos que a imagem  $x_i$  possui probabilidade 0.8 de ser da classe 1, 0.15 de ser da classe 2 e 0.05 de ser da classe 3. Esse é um problema de classificação multirrótulo. A forma mais geral da entropia cruzada inclui esse caso,

imagem e retorne como saída um vetor de  $C$  valores. Quando a imagem for da classe 1, queremos que o modelo retorne o vetor  $\mathbf{q} = (1, 0, 0, \dots, 0)$ . Quando a imagem for da classe 2, queremos que o modelo retorne  $\mathbf{q} = (0, 1, 0, \dots, 0)$ . Quando a imagem for da classe  $C$ , queremos que o modelo retorne  $\mathbf{q} = (0, 0, 0, \dots, 1)$ .

Como iremos utilizar uma rede neural para representar esse modelo, que consiste numa sequência de multiplicações e somas, o modelo não será perfeito. Por exemplo, ao invés de retornar o vetor  $\mathbf{q} = (1, 0, 0, \dots, 0)$  para uma imagem da classe 1 o modelo irá retornar o vetor  $\mathbf{r} = (7, 2.1, 1.2, \dots, 2.4)$ , ou seja, um valor alto para a classe 1 e baixo para as demais classes. Esse vetor é útil para classificar a imagem, afinal, basta encontrarmos o índice do elemento possuindo o maior valor<sup>10</sup>, mas é interessante transformarmos esse vetor em probabilidades para que seja possível interpretar seus valores como sendo a probabilidade de imagem pertencer a cada classe. Isso também possibilitará utilizarmos a função de verossimilhança para encontrar o melhor modelo possível para os dados.

Uma operação simples que podemos fazer é normalizar os valores pela soma, ou seja, definir

$$\mathbf{p}(i) = \frac{\mathbf{r}(i)}{\sum_{i=1}^C \mathbf{r}(i)} \quad (1.12)$$

Nesse caso, garantimos que os valores de  $\mathbf{p}$  são entre 0 e 1, e também garantimos que a soma dos valores é 1<sup>11</sup>. Isso permite interpretar  $\mathbf{p}$  como um vetor contendo as probabilidades da imagem pertencer a cada classe.

Curiosamente, um outro tipo de normalização costuma ser mais utilizada, a chamada *função softmax*, dada por

$$\mathbf{p}(i) = \frac{e^{\mathbf{r}(i)}}{\sum_{i=1}^C e^{\mathbf{r}(i)}} \quad (1.13)$$

mas o propósito dela é o mesmo da Equação 1.12, transformar os valores em probabilidades. Por isso é comum que a saída da rede seja normalizada pela função softmax para que seja possível interpretar os valores como probabilidades.

Um *addendum*, em tarefas de classificação em duas classes, chamada de classificação binária, podemos usar o mesmo procedimento, ou seja, ter um modelo cuja saída é um vetor

mas para simplificar estamos considerando apenas uma única classe para cada imagem.

<sup>10</sup>Operação `argmax()` do Pytorch.

<sup>11</sup>Utilizaremos a notação  $\mathbf{r}(i)$  para representar um elemento do vetor  $\mathbf{r}$ . Isso porque a notação  $\mathbf{r}_i$  já está sendo utilizada para representar a saída associada à imagem  $i$ .

de dois valores  $\mathbf{r} = (r_1, r_2)$  e aplicar a função softmax. Por outro lado, também é possível, e muito comum, utilizar um modelo que possui como saída apenas um valor  $r_1$ , e considerar que se o valor for positivo a imagem é classificada na classe 1, e se o valor for negativo a imagem é classificada na classe 2. Nesse caso utilizamos a função sigmoide para normalizar os valores entre 0 e 1

$$p_1 = \frac{1}{1 + e^{-r_1}} \quad (1.14)$$

Segundo essa equação, quando  $r_1 < 0$  temos que  $0 \leq p_1 < 0.5$  e quando  $r_1 > 0$  temos que  $0.5 < p_1 \leq 1$ . Portanto, o resultado da função pode ser interpretado como sendo a probabilidade de que a imagem seja da classe 1. Se essa probabilidade for maior que 0.5, classificamos a imagem na classe 1. Portanto, em um problema de classificação binária basta que o modelo retorne um único valor.

Tendo os valores normalizados podemos retornar ao nosso problema original do começo desta seção. Lembrando que temos um conjunto de treinamento composto por uma sequência de  $n$  imagens e os respectivos rótulos  $s = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ . O nosso modelo é representado por uma função  $\mathbf{f}(x; \theta)$ , e consideraremos que ele inclui a função softmax e portanto retorna um vetor  $\mathbf{p}$  de tamanho  $C$  contendo probabilidades para cada classe.

Para cada item  $x_i$  pertencente à classe  $y_i$  queremos que o vetor  $\mathbf{p}_i$  retornado pelo modelo seja o mais próximo possível do “vetor ideal”  $\mathbf{q}_i$  dado por um vetor de zeros exceto na posição  $y_i$  que possui o valor 1. Para isso, precisamos definir um critério de similaridade entre os vetores. Se tivermos esse critério, será possível otimizarmos o modelo para minimizar o critério.

De longe o critério mais comum utilizado em classificação é a entropia cruzada, do inglês *cross entropy*. Ela é definida para cada imagem  $i$  como

$$E_i = -\log(\mathbf{p}_i(y_i)) \quad (1.15)$$

Na equação,  $\mathbf{p}_i(y_i)$  representa o valor na posição  $y_i$  do vetor  $\mathbf{p}_i$ . Em outras palavras, a entropia cruzada para a imagem  $i$  é o valor na  $y_i$ -ésima posição do vetor de probabilidades retornado pelo modelo. Quanto mais próximo de 1 for o valor de  $\mathbf{p}_i(y_i)$ , mais próximo de zero serão os demais valores, pois a soma deve ser 1. Portanto, quanto maior for o valor de  $\mathbf{p}_i(y_i)$ , mais próximo o vetor  $\mathbf{p}_i$  estará do vetor ideal  $\mathbf{q}_i$ !

A entropia cruzada para todas as imagens da base  $s$  é então calculada como

$$E(s) = -\sum_i^n \log(\mathbf{p}_i(y_i)) \quad (1.16)$$

Lembrando que  $\mathbf{p}$  é a saída do nosso modelo, podemos reescrever a equação como

$$E(s; \theta) = - \sum_i^n \log(\mathbf{f}(x_i; \theta)(y_i)) \quad (1.17)$$

Note que essa equação é basicamente a Equação 1.8. Na seção anterior vimos que a função de verossimilhança é dada pelo negativo do logaritmo da probabilidade. Portanto, a entropia cruzada nada mais é do que a função de verossimilhança calculada para as probabilidades das classes “corretas” da base. Portanto, o procedimento de otimização para encontrar o modelo ideal  $\mathbf{f}(x; \theta)$  é o mesmo que o visto no Algoritmo 1 da Seção 1.1.2. Também é importante notar que  $E(s; \theta)$  representa um único valor para toda a base de dados.

### \*Equação geral da entropia cruzada

Na discussão acima consideramos que a distribuição que queremos modelar consiste no vetor  $\mathbf{q}$  possuindo apenas valores 0 e 1. Mas em geral é possível considerar que o vetor pode possuir qualquer valor de probabilidade para as classes. Isso ocorre em tarefas de classificação multirrótulo, na qual queremos associar diversas possíveis classes às imagens, e também na técnica chamada de *label smoothing*, que consiste em associar um valor menor do que 1 à classe correta, com a ideia de que o modelo não precisa ser perfeito, basta que a classe correta seja a mais provável.

Nesse caso, a entropia cruzada de uma imagem  $i$  é dada por

$$E_i = - \sum_i^C \mathbf{q}(y_i) \log(\mathbf{p}_i(y_i)) \quad (1.18)$$

A entropia cruzada pode ser entendida como uma medida de dissimilaridade entre as distribuições. Quando  $\mathbf{p}_i(y_i) \approx 0$ , o termo  $-\log(\mathbf{p}_i(y_i))$  tenderá a um valor muito alto. Mas se  $\mathbf{q}(y_i)$  também for próximo de zero, os termos “se cancelam” e a entropia resulta em um valor baixo.

## 1.2 Conceitos de cálculo diferencial

Nesta seção desenvolveremos os conceitos necessários para calcular o gradiente da função de verossimilhança  $\mathcal{L}(s; \theta)$ .

### 1.2.1 Derivada de funções univariadas

A derivada de uma função  $f(x)$ <sup>12</sup> indica a *taxa de variação* da função. Por exemplo, suponha que estamos enchendo uma garrafa de água em um bebedouro, e a função  $f(t)$  representa o volume de água na garrafa ao longo do tempo. A derivada de  $f(t)$ , representada por  $df/dt$ , indica a taxa de variação da quantidade de água na garrafa, ou seja, quão rápido a garrafa está enchendo. Essa taxa de variação também representa o fluxo de água do bebedouro.

Considere a função  $f(x) = 3x$ . Para cada unidade 1 que a variável  $x$  é aumentada, a função  $f(x)$  cresce por uma quantia 3. Por exemplo, se  $x_1 = 3$  temos que  $f(x_1) = 9$ . Se  $x_2 = x_1 + 1 = 4$ , temos que  $f(x_2) = 12$  e portanto  $f(x_2) = f(x_1) + 3$ . Por isso, a derivada de  $f(x)$  é dada por  $df/dx = 3$ . A Tabela 1.1 apresenta algumas derivadas relevantes.

Função	Derivada
$f(x) = a$	$df/dx = 0$
$f(x) = x$	$df/dx = 1$
$f(x) = ax + b$	$df/dx = a$
$f(x) = x^2$	$df/dx = 2x$
$f(x) = e^x$	$df/dx = e^x$

Tabela 1.1: Algumas derivadas relevantes.  $a$  e  $b$  são constantes.

Em geral as derivadas que encontramos ao trabalhar com redes neurais são simples. A dificuldade aparece no fato de que em geral as funções envolvidas possuem várias variáveis e são compostas, isto é, funções são aplicadas no resultado de outras funções sucessivamente.

### 1.2.2 Derivada de funções univariadas compostas

A saída de uma função  $f$  pode ser utilizada como entrada de outra função  $g$ , isso é chamado de composição de funções. Nesse caso, podemos escrever  $h(x) = g(f(x))$ . Mais importante, dado uma função  $f$ , podemos *decompor* essa função em funções intermediárias. Por exemplo, a função  $f(x) = (x - 5)^2 + 8$  pode ser escrita como  $f(g) = g^2 + 8$ , sendo que  $g(x) = x - 5$ . Podemos ir além e escrever  $f(x)$  como  $f(h) = h + 8$ , onde  $h(g) = g^2$ .

---

<sup>12</sup>A função  $f(x)$  é chamada de univariada porque possui apenas uma variável.

Para calcular a derivada de uma função complexa, basta escrevermos cada parte da função como uma nova função, e derivar cada função à parte. O produto de todas as derivadas calculadas resulta na derivada da função original. Essa é a chamada *regra da cadeia*. Por exemplo, se quisermos encontrar a derivada da função  $f(x) = (x - 5)^2 + 8$ , podemos escrever  $f(h) = h + 8$ ,  $h(x) = (x - 5)^2$ . Nesse caso, a derivada de  $f$  em relação à  $x$  será dada por

$$\frac{df}{dx} = \frac{df}{dh} \frac{dh}{dx} \quad (1.19)$$

Calculamos cada derivada à parte. No cálculo de  $df/dh$ , tratamos  $h$  como uma variável de  $f$ , o que resulta em  $df/dh = 1$ . Para calcular  $dh/dx$  podemos escrever  $h$  como  $h(g) = g^2$ ,  $g(x) = x - 5$ , e portanto

$$\frac{dh}{dx} = \frac{dh}{dg} \frac{dg}{dx} \quad (1.20)$$

Para calcular  $dh/dg$  usamos a regra da potência, obtendo  $dh/dg = 2g$ . Temos também que  $dg/dx = 1$ . Com isso, conseguimos calcular o valor de  $df/dx$ . Em resumo, temos a seguinte decomposição da função  $f(x) = (x - 5)^2 + 8$ :

$$\begin{aligned} g(x) &= x - 5 &\rightarrow dg/dx &= 1 \\ h(g) &= g^2 &\rightarrow dh/dg &= 2g \\ f(h) &= h + 8 &\rightarrow df/dh &= 1 \end{aligned}$$

A derivada de  $f(x)$  é dada por

$$\begin{aligned} \frac{df}{dx} &= \frac{df}{dh} \frac{dh}{dg} \frac{dg}{dx} \\ &= 1 \times 2g \times 1 \\ &= 2g \\ &= 2(x - 5) \\ &= 2x - 10 \end{aligned} \quad (1.21)$$

Um exemplo mais complexo, podemos derivar a função  $f(x) = a(x - b)^2 - c$ , onde  $a$ ,  $b$  e  $c$  são constantes. Nesse caso temos

$$\begin{aligned} l(x) &= x - b &\rightarrow dl/dx &= 1 \\ g(l) &= l^2 &\rightarrow dg/dl &= 2l \\ h(g) &= ag &\rightarrow dh/dg &= a \\ f(h) &= h - c &\rightarrow df/dh &= 1 \end{aligned}$$

A derivada é dada por

$$\begin{aligned}
\frac{df}{dx} &= \frac{df}{dh} \frac{dh}{dg} \frac{dg}{dl} \frac{dl}{dx} \\
&= 1 \times a \times 2l \times 1 \\
&= a2(x - b)
\end{aligned} \tag{1.22}$$

Portanto, qualquer função, por mais complexa que seja, pode ser derivada se soubermos derivar cada operação individual realizada pela função. No exemplo acima, basta sabermos derivar uma função linear do tipo  $f(x) = ax + b$  e uma função quadrática do tipo  $f(x) = x^2$ .

Agora vamos ver um exemplo no qual temos uma sequência de funções aplicadas em uma entrada. Uma rede neural funciona dessa forma. Vamos considerar as seguintes operações:

$$\begin{aligned}
a_1(x) &= w_1x + b_1 \\
a_2(a_1) &= w_2a_1 + b_2 \\
a_3(a_2) &= w_3a_2 + b_3 \\
a_4(a_3) &= (a_3 - w_4)^2
\end{aligned} \tag{1.23}$$

onde  $w_i$  e  $b_i$  são constantes e  $a_i$  são funções que recebem uma entrada e geram uma saída<sup>13</sup>. Queremos encontrar o valor de  $da_4/dx$ , ou seja, a derivada da saída em relação à entrada. Para isso, decomponemos a função em operações atômicas. Iremos definir alguns valores intermediários  $q_i$  para as operações:

$$\begin{aligned}
q_1 &= w_1x & \rightarrow dq_1/dx &= w_1 \\
a_1 &= q_1 + b_1 & \rightarrow da_1/dq_1 &= 1 \\
q_2 &= w_2a_1 & \rightarrow dq_2/da_1 &= w_2 \\
a_2 &= q_2 + b_2 & \rightarrow da_2/dq_2 &= 1 \\
q_3 &= w_3a_2 & \rightarrow dq_3/da_2 &= w_3 \\
a_3 &= q_3 + b_3 & \rightarrow da_3/dq_3 &= 1 \\
q_4 &= a_3 - w_4 & \rightarrow dq_4/da_3 &= 1 \\
a_4 &= q_4^2 & \rightarrow da_4/dq_4 &= 2q_4
\end{aligned}$$

A derivada é calculada como

<sup>13</sup>Essas funções serão as camadas de uma rede neural.

$$\begin{aligned}\frac{da_4}{dx} &= \frac{da_4}{dq_4} \frac{dq_4}{da_3} \frac{da_3}{dq_3} \frac{dq_3}{da_2} \frac{da_2}{dq_2} \frac{dq_2}{da_1} \frac{da_1}{dq_1} \frac{dq_1}{dx} \\ &= 2q_4 w_3 w_2 w_1\end{aligned}\tag{1.24}$$

Note uma questão interessante. Para calcular a saída da função  $a_4$  aplicamos as funções  $a_1$ ,  $a_2$ ,  $a_3$  e  $a_4$  em ordem. Para calcular a derivada da saída em relação à entrada multiplicamos as derivadas na *ordem contrária*. Por isso esse algoritmo é conhecido como *backpropagation*. Temos a operação *forward* que calcula o resultado da função, e a operação *backward* que faz o caminho contrário e calcula a derivada da saída em relação à entrada.

As bibliotecas Pytorch, Tensorflow e Jax implementam a chamada *diferenciação automática*. Essa técnica consiste em dividir qualquer operação complexa em operações atômicas simples de derivar. Para calcular a derivada de uma expressão complexa, basta multiplicar todas as derivadas das operações atômicas.

### 1.2.3 Derivada de funções multivariadas

Uma função de duas variáveis  $f(x, y)$  possui duas derivadas: em relação às variáveis  $x$  e  $y$ . Chamamos essas derivadas de *parciais*, pois cada derivada descreve parcialmente a taxa de variação da função. Representamos as derivadas parciais por  $\partial f / \partial x$  e  $\partial f / \partial y$ . Por exemplo, para a função  $f(x, y) = xy + x$  temos

$$\frac{\partial f}{\partial x} = y + 1\tag{1.25}$$

$$\frac{\partial f}{\partial y} = x\tag{1.26}$$

Em geral, uma função pode depender de  $m$  variáveis. Nesse caso, representamos as variáveis por um vetor  $\mathbf{x} = (x_1, x_2, \dots, x_m)$ <sup>14</sup>, e escrevemos a função como  $f(\mathbf{x})$ . Essa função admite  $m$  derivadas parciais. O conjunto de derivadas parciais de uma função é chamado de gradiente, e é representado por  $\nabla f$ :

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_m} \right)\tag{1.27}$$

O gradiente é sempre um vetor, e indica a direção de maior crescimento de uma função. A magnitude do gradiente (tamanho do vetor) indica a taxa de crescimento da função na

<sup>14</sup>Vetores sempre serão representados utilizando a versão em negrito de um caractere. Note a diferença entre  $x$  e  $\mathbf{x}$ . Caso tenha dúvidas se um caractere representa um vetor ou não, veja o Apêndice A

direção do gradiente. Por exemplo, considere a função  $f(\mathbf{x}) = 6x_1 + 3x_2$ . Essa função varia a uma taxa igual a 6 em relação à  $x_1$  e a uma taxa igual a 3 em relação à  $x_2$ . Com isso, dizemos que a função varia a uma taxa de 6 na direção de  $x_1$  e uma taxa de 3 na direção de  $x_2$ . Isso dito, qual a direção na qual a função varia na maior taxa possível? Essa direção é representada pelo vetor  $(6, 3)$ , e a taxa de variação nessa direção é dada pela magnitude do vetor, que é  $\sqrt{6^2 + 3^2} \approx 6.7$ .

É importante salientar, enquanto que a entrada de uma função pode ser múltiplas variáveis (um vetor), a saída de uma função escalar sempre é um único valor.

No restante desta seção analisaremos um tipo particular de função que é relevante para redes neurais, que são funções do tipo

$$f(\mathbf{x}, \mathbf{w}) = w_1x_1 + w_2x_2 + \cdots + w_mx_m \quad (1.28)$$

Essa função possui  $m$  parâmetros  $\mathbf{w} = (w_1, w_2, \dots, w_m)$  e recebe como entrada um vetor de  $m$  variáveis  $\mathbf{x} = (x_1, x_2, \dots, x_m)$ . A saída da função é uma *combinação linear* dos valores  $\mathbf{x}$  ponderados pelos valores  $\mathbf{w}$ . Essa função pode ser escrita de duas outras formas equivalentes

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^m w_i x_i \quad (1.29)$$

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}\mathbf{x} \quad (1.30)$$

Na Equação 1.30 escrevemos a função como um *produto escalar* dos vetores  $\mathbf{x}$  e  $\mathbf{w}$ . Essa é uma notação muito mais compacta que vai facilitar as contas quando trabalharmos com matrizes.

Como vimos, o gradiente de  $f(\mathbf{x}, \mathbf{w})$  em relação à  $\mathbf{x}$ <sup>15</sup> consiste nas derivadas parciais  $\partial f / \partial x_i$  em relação a cada variável  $x_i$ . Portanto temos

$$\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) = \left( \frac{\partial f(\mathbf{x}, \mathbf{w})}{\partial x_1}, \frac{\partial f(\mathbf{x}, \mathbf{w})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x}, \mathbf{w})}{\partial x_m} \right) \quad (1.31)$$

$$= (w_1, w_2, \dots, w_m) \quad (1.32)$$

$$= \mathbf{w} \quad (1.33)$$

É interessante que o uso de vetores permite escrever o cálculo do gradiente de forma bem similar ao caso de uma função de apenas uma variável, isto é, o gradiente “elimina” a variável

<sup>15</sup>Para deixar explícito a variável que está sendo usada para calcular o gradiente, em alguns casos utilizaremos a notação  $\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$ .

**x.**

Note que também podemos tratar o vetor  $\mathbf{w}$  como sendo as variáveis da função, e considerar a entrada  $\mathbf{x}$  como uma constante. Isso possibilita calcular o gradiente em relação aos parâmetros  $\mathbf{w}$ :

$$\nabla_{\mathbf{w}} f(\mathbf{x}, \mathbf{w}) = (x_1, x_2, \dots, x_m) \quad (1.34)$$

$$= \mathbf{x} \quad (1.35)$$

A notação em vetores facilita a representação de cálculos quando temos múltiplas variáveis, mas em redes neurais as operações em geral envolvem o produto de uma matriz por um vetor. Por isso teremos que aprimorar um pouco a nossa notação para incluir o uso de matrizes. Vamos definir dois tipos de vetores: vetores linha e vetores coluna. Um vetor linha é exatamente o vetor que usamos até agora, os seus valores são representados um ao lado do outro na direção horizontal. No vetor coluna, representamos o vetor com um valor abaixo do outro, na direção vertical. Por exemplo, podemos transformar o vetor  $\mathbf{x}$  em um vetor coluna fazendo

$$\mathbf{x}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad (1.36)$$

O símbolo  $T$  representa o transposto do vetor  $\mathbf{x}$ . A ideia é que um vetor linha irá representar uma *linha* de uma matriz e um vetor coluna irá representar uma *coluna*. De forma alternativa, podemos pensar que um vetor linha é uma matriz de uma linha, e um vetor coluna uma matriz de uma coluna. Isso irá facilitar a construção de matrizes com os resultados dos nossos cálculos. No restante desta seção, todos os vetores serão linha ou coluna, e teremos que tomar cuidado para que as operações que fazemos com esses vetores seja válida. Por exemplo, *o produto  $\mathbf{w}\mathbf{x}$  não é mais válido!* Esse produto representa multiplicar uma matriz de tamanho  $1 \times m$  por outra matriz de tamanho  $1 \times m$ . Pela regra de multiplicação matricial, o número de colunas da primeira matriz deve ser igual ao número de linhas da segunda matriz.

Para calcular o valor de  $f(\mathbf{x}, \mathbf{w})$  agora temos que utilizar

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}\mathbf{x}^T \quad (1.37)$$

ou de forma equivalente podemos utilizar

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{x}\mathbf{w}^T \quad (1.38)$$

Na Equação 1.37 temos o produto de uma matriz  $1 \times m$  por outra matriz  $m \times 1$ . Então está tudo certo, e o resultado é dado por uma matriz de tamanho  $1 \times 1$ , que é simplesmente um valor escalar.

### 1.2.4 \*Camada linear de uma rede neural

A camada linear de uma rede neural consiste em um *conjunto* de  $n$  funções, cada uma aplicada em um mesmo vetor de  $m$  variáveis de entrada. Novamente, as variáveis de entrada são representadas por um vetor linha  $\mathbf{x} = (x_1, x_2, \dots, x_m)$ . Cada função realiza a combinação linear das variáveis de entrada e possui parâmetros próprios. Iremos representar uma função específica  $i$  como  $f_i(\mathbf{x}, \mathbf{w}_i)$ . Portanto podemos escrever as operações realizadas por uma camada linear<sup>16</sup> como

$$\begin{aligned} f_1(\mathbf{x}, \mathbf{w}_1) &= w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m \\ f_2(\mathbf{x}, \mathbf{w}_2) &= w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m \\ &\vdots \\ f_n(\mathbf{x}, \mathbf{w}_n) &= w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m \end{aligned} \tag{1.39}$$

Na equação utilizamos  $w_{ij}$  para representar os parâmetros das funções.  $w_{ij}$  representa o  $j$ -ésimo parâmetro da  $i$ -ésima função.

Podemos representar esse sistema de  $n$  equações como um produto matriz-vetor. Para isso, basta definirmos uma matriz  $W$ <sup>17</sup>

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} \tag{1.40}$$

Tendo a matriz  $W$ , podemos escrever o conjunto de equações 1.39 como

$$\mathbf{f}(\mathbf{x}, W) = W\mathbf{x}^T \tag{1.41}$$

A Equação 1.41 representa uma operação bem complexa. Ela representa o resultado da aplicação de um conjunto de  $n$  funções, cada uma com  $m$  parâmetros em um vetor de  $m$

<sup>16</sup>Nesta discussão desconsideraremos o chamado *bias* da camada, que consiste em somar um valor constante  $b_i$  em cada equação  $f_i(\mathbf{x}, \mathbf{w}_i)$ .

<sup>17</sup>Matrizes serão sempre representadas por letras maiúsculas.

variáveis de entrada. É importante salientar que, como temos várias funções, o resultado da equação será um vetor, pois temos o produto de uma matriz de tamanho  $n \times m$  por um vetor de tamanho  $m \times 1$ . O resultado terá tamanho  $n \times 1$  e será um vetor coluna.

Uma forma equivalente da Equação 1.41 é

$$\mathbf{f}(\mathbf{x}, W) = \mathbf{x}W^T \quad (1.42)$$

Note que nesse caso o resultado será um vetor linha.<sup>18</sup>

### \*Derivada de uma camada linear de uma rede neural<sup>19</sup>

Considere o conjunto de funções  $\mathbf{f}(\mathbf{x}, W) = \mathbf{x}W^T$  representando uma camada de uma rede neural. Queremos calcular a derivada das funções em relação a cada parâmetro  $w_{ij}$ . Isso permitirá otimizar a rede neural para realizar qualquer tarefa (como descrito na Seção 1.1). Utilizando as equações 1.39 vemos por exemplo que

$$\frac{df_1(\mathbf{x}, \mathbf{w}_1)}{\partial w_{11}} = x_1 \quad (1.43)$$

De maneira geral, a derivada da função  $f_i(\mathbf{x}, \mathbf{w}_i)$  em relação ao parâmetro  $w_{ij}$  é dada por

$$\frac{df_i(\mathbf{x}, \mathbf{w}_i)}{\partial w_{ij}} = x_j \quad (1.44)$$

A derivada de uma função  $f_i(\mathbf{x}, \mathbf{w}_i)$  em relação a um parâmetro  $w_{kj}$  que não pertence à função ( $k \neq i$ ) é igual a 0. Com isso, a derivada das funções pode ser escrita como

$$\frac{d\mathbf{f}(\mathbf{x}, W)}{dw_{ij}} = \left( \frac{df_1(\mathbf{x}, \mathbf{w}_1)}{dw_{ij}}, \frac{df_2(\mathbf{x}, \mathbf{w}_2)}{dw_{ij}}, \dots, \frac{df_i(\mathbf{x}, \mathbf{w}_i)}{dw_{ij}}, \dots, \frac{df_n(\mathbf{x}, \mathbf{w}_n)}{dw_{ij}} \right) \quad (1.45)$$

$$= (0, 0, \dots, x_j, \dots, 0) \quad (1.46)$$

$$= \boldsymbol{\delta}_i x_j \quad (1.47)$$

O vetor  $\boldsymbol{\delta}_i$  é formado por  $n$  valores 0 exceto na posição  $i$ , na qual ele possui o valor 1<sup>20</sup>.

<sup>18</sup>\*Essa forma de multiplicação é comum em bibliotecas como o Pytorch pois a entrada da camada é formada por diversos vetores  $\mathbf{x}$  (um batch). Manter a entrada no lado esquerdo da equação garante que as dimensões das matrizes continuam válidas para o produto.

<sup>19</sup>Esta seção é bem complexa e envolve conceitos avançados de cálculo multivariado. Mas o entendimento das equações possibilita entender todo o processo de treinamento de uma rede neural. Portanto, é importante que seja feito um esforço para entender o conteúdo. Para isso, também é importante que o conteúdo das seções anteriores esteja bem entendido.

<sup>20</sup>Uma forma de interpretar a Equação 1.47 é que os índices  $ij$  de  $w_{ij}$  representam a seleção de um valor na posição  $j$  de  $\mathbf{x}$  e a alocação desse valor na posição  $i$  de um novo vetor. Utilizando numpy essa operação

Também será necessário calcularmos a derivada de cada função em  $\mathbf{f}(\mathbf{x}, W)$  em relação aos valores de  $\mathbf{x}$ . Podemos fazer o processo passo-a-passo utilizando as equações 1.39. Por exemplo, a derivada da função  $f_1(\mathbf{x}, \mathbf{w}_1)$  em relação à variável  $x_1$  é dada por

$$\frac{\partial f_1(\mathbf{x}, \mathbf{w}_1)}{\partial x_1} = w_{11} \quad (1.48)$$

De forma geral, a derivada da função  $f_i(\mathbf{x}, \mathbf{w}_i)$  em relação à variável  $x_j$  é dada por

$$\frac{\partial f_i(\mathbf{x}, \mathbf{w}_i)}{\partial x_j} = w_{ij} \quad (1.49)$$

Agrupando todas as derivadas em uma matriz temos

$$\frac{d\mathbf{f}(\mathbf{x}, W)}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f_1(\mathbf{x}, \mathbf{w}_1)}{\partial x_1} & \frac{\partial f_1(\mathbf{x}, \mathbf{w}_1)}{\partial x_2} & \dots & \frac{\partial f_1(\mathbf{x}, \mathbf{w}_1)}{\partial x_m} \\ \frac{\partial f_2(\mathbf{x}, \mathbf{w}_2)}{\partial x_1} & \frac{\partial f_2(\mathbf{x}, \mathbf{w}_2)}{\partial x_2} & \dots & \frac{\partial f_2(\mathbf{x}, \mathbf{w}_2)}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(\mathbf{x}, \mathbf{w}_n)}{\partial x_1} & \frac{\partial f_n(\mathbf{x}, \mathbf{w}_n)}{\partial x_2} & \dots & \frac{\partial f_n(\mathbf{x}, \mathbf{w}_n)}{\partial x_m} \end{bmatrix} \quad (1.50)$$

Essa matriz é chamada de *matriz Jacobiana*. Ela é usada sempre que temos a derivada de um conjunto de valores (os resultados das funções  $\mathbf{f}(\mathbf{x}, W)$ ) em relação a um outro conjunto de valores (o vetor de entrada  $\mathbf{x}$ ). Em outras palavras, ela representa a derivada de um vetor em relação a outro vetor<sup>21</sup>. Mas calculando cada derivada dessa matriz vemos que a matriz é exatamente igual a  $W$ . Portanto temos que

$$\frac{d\mathbf{f}(\mathbf{x}, W)}{d\mathbf{x}} = W \quad (1.51)$$

Resumindo o que derivamos nesta seção, dado um conjunto de funções lineares  $\mathbf{f}(\mathbf{x}, W)$  representando uma camada linear de uma rede neural, temos que

$$\mathbf{f}(\mathbf{x}, W) = \mathbf{x}W^T \quad (1.52)$$

$$\frac{d\mathbf{f}(\mathbf{x}, W)}{d\mathbf{x}} = W \quad (1.53)$$

$$\frac{d\mathbf{f}(\mathbf{x}, W)}{dw_{ij}} = \delta_i x_j \quad (1.54)$$

Uma propriedade interessante, e muito importante, de redes neurais é que dado os parâmetros  $W$  de uma camada, o resultado da camada é computado pela matriz  $W^T$  e a derivada

---

seria `v=np.zeros(n); v[i]=x[j]`.

<sup>21</sup>Usualmente a matriz Jacobiana é representada pelo símbolo  $J$  em artigos e livros.

da saída em relação à entrada é simplesmente o transposto dessa matriz. Dessa forma, o cálculo da saída da camada e da derivada são praticamente idênticos.

### 1.2.5 \*Derivada de funções multivariadas compostas

Na seção anterior consideramos uma camada linear de uma rede neural. Tipicamente, uma rede neural é composta por  $L$  camadas. Representaremos por  $\mathbf{f}^l(\mathbf{x}, W_l)$  a  $l$ -ésima camada de uma rede neural. A rede neural em si será representada por  $\mathcal{N}(x; \mathbb{W})$ . O símbolo  $\mathbb{W}$  representa o conjunto de todos os parâmetros de uma rede neural, ou seja, o conjunto de valores de todas as matrizes  $W_1, W_2, \dots, W_L$ .

Uma rede neural consiste em aplicar sucessivamente cada camada em dados de entrada. A  $l$ -ésima camada recebe a saída da camada  $l-1$ . Por exemplo, a aplicação da primeira camada da rede dará um resultado  $\mathbf{a}_1 = \mathbf{f}^1(\mathbf{x}, W_1)$ . Os valores  $\mathbf{a}_1$  são chamados de *ativações* da rede. A aplicação da segunda camada nas ativações da primeira dará um resultado  $\mathbf{a}_2 = \mathbf{f}^2(\mathbf{a}_1, W_2)$ , e assim por diante, até a última camada.

Em qualquer tarefa de treinamento de rede neural, o objetivo é otimizar os parâmetros  $\mathbb{W}$  de forma que a rede dê um resultado esperado de acordo com o conjunto de treinamento. Para isso é necessário encontrar a derivada da saída da rede  $\mathcal{N}(x; \mathbb{W})$  em relação aos parâmetros  $\mathbb{W}$ . Essas derivadas são calculadas pela regra da cadeia.

A derivada da saída da rede em relação a um parâmetro  $w_{ij}^l$ <sup>22</sup> é calculada como

$$\frac{\partial \mathcal{N}(x; \mathbb{W})}{\partial w_{ij}^l} = \frac{d\mathbf{a}_L}{d\mathbf{a}_{L-1}} \frac{d\mathbf{a}_{L-1}}{d\mathbf{a}_{L-2}} \cdots \frac{d\mathbf{a}_{l+1}}{d\mathbf{a}_l} \frac{d\mathbf{a}_l}{dw_{ij}^l} \quad (1.55)$$

Cada termo  $d\mathbf{a}_i/d\mathbf{a}_{i-1}$  é uma matriz Jacobiana que descreve a variação da saída da camada em relação à entrada, ou seja,

$$\frac{\partial \mathcal{N}(x; \mathbb{W})}{\partial w_{ij}^l} = J_L J_{L-1} \cdots J_{l+1} \frac{d\mathbf{a}_l}{dw_{ij}^l} \quad (1.56)$$

onde  $J_i$  são as matrizes Jacobianas das camadas. Para camadas lineares, cada elemento do gradiente pode ser calculado pelas equações 1.53 e 1.54, começando da derivada mais à esquerda.

Como descrito na Seção 1.1, a partir da saída da rede é calculada a função de verossimilhança  $\mathcal{L}(x; \mathbb{W})$  para medir a performance da rede, e o processo de otimização consiste em calcular o gradiente da função em relação aos parâmetros. O gradiente é formado pela derivada em relação à cada parâmetro  $w_{ij}^l$ . Portanto temos que

<sup>22</sup>Estamos ficando sem espaço para símbolos! Para deixar claro,  $l$  representa uma camada e  $ij$  representa a posição na linha  $i$  e coluna  $j$  da matriz  $W_l$  contendo os parâmetros da camada.

$$\frac{\partial \mathcal{L}(x; \mathbb{W})}{\partial w_{ij}^l} = \frac{\partial \mathcal{L}(x; \mathbb{W})}{d\mathbf{a}_L} \frac{\partial \mathcal{N}(x; \mathbb{W})}{\partial w_{ij}^l} \quad (1.57)$$

Note que  $\partial \mathcal{L}(x; \mathbb{W})/\partial w_{ij}^l$  é um valor, pois estamos calculando a derivada de um valor em relação a um outro valor. Para interpretar essa equação, no caso de camadas lineares podemos reescrevê-la como

$$\frac{\partial \mathcal{L}(x; \mathbb{W})}{\partial w_{ij}^l} = \nabla_{\mathbf{a}_L} \mathcal{L}(x; \mathbb{W}) J_L J_{L-1} \dots J_{l+1} \frac{d\mathbf{a}_l}{dw_{ij}^l} \quad (1.58)$$

$$= \nabla_{\mathbf{a}_L} \mathcal{L}(x; \mathbb{W}) W_L W_{L-1} \dots W_{l+1} \boldsymbol{\delta}_i^T a^{l-1} \quad (1.59)$$

Portanto, para encontrar as derivadas, primeiramente é calculado o gradiente da função de verossimilhança em relação à saída da rede, o que resulta em um vetor. Em sequência, é realizada uma série de multiplicações entre vetores e matrizes. Por fim, o vetor  $\boldsymbol{\delta}_i$  seleciona uma posição específica do resultado, que é multiplicado pelo valor de entrada da camada  $a_j^{l-1}$ <sup>23</sup><sup>24</sup>.

Calculamos a Jacobiana apenas de camadas lineares. Em geral, após cada camada linear de uma rede há uma camada não linear. Mas todo o processo de calcular a saída da rede e as derivadas é o mesmo. Basta que a matriz Jacobiana da camada seja adicionada à Equação 1.56.

É importante salientar que todo esse processo é feito de forma totalmente automatizada por bibliotecas como Pytorch, Tensorflow e Jax. Mas conhecer o funcionamento do processo ajuda a entender alguns conceitos importantes. Por exemplo, como todo o processo envolve multiplicações matriciais, ele é altamente paralelizável. Vemos também que para calcular o gradiente de uma camada linear é preciso armazenar a entrada da camada. Isso implica que todas as ativações de entrada de uma camada linear são armazenadas quando uma rede é aplicada em um dado de entrada, o que causa um grande uso de memória. Mas após o cálculo do gradiente da camada  $l$  as ativações das camadas posteriores não são mais necessárias e é possível desalocar os valores.

---

<sup>23</sup>A primeira camada possui índice  $l = 1$ , nesse caso  $a_j^{l-1} = x_j$ , que é a entrada da rede.

<sup>24</sup>Uma implementação da Equação 1.59 pode ser encontrada em [este endereço](#).

### 1.3 \*Regularização e weight decay (módulo 5)

A regularização L2 consiste em adicionar uma penalidade na função de loss para que os parâmetros do modelo não tenham valores muito altos. A ideia é que valores altos de parâmetros possibilitem derivadas altas na função, o que facilita que a função tenha um sobreajuste nos dados de treinamento. Para entender isso, considere uma função típica de uma camada linear  $f(x) = wx + b$ . A derivada dessa função em relação à entrada é dada por  $df(x)/dx = w$ . Portanto, a taxa de variação da função em relação a  $x$  é dada por  $w$ . Quanto maior o valor de  $w$ , mais rápido a função pode variar. Ao forçar que os parâmetros tenham valores pequenos, a função tenderá a ser mais suave, o que tende a levar a uma melhor generalização.

A regularização L2 consiste em definir a loss

$$L = E_{ce} + \lambda \sum_i w_i^2 \quad (1.60)$$

onde  $E_{ce}$  é a entropia cruzada usual e  $\lambda$  uma constante. O segundo termo da equação representa uma penalidade caso algum parâmetro seja muito alto. Portanto, é bem simples implementar a regularização L2. Mas podemos fazer uma implementação ainda mais eficiente se notarmos que

$$\nabla L(\mathbf{w}) = \nabla E_{ce}(\mathbf{w}) + 2\lambda\mathbf{w} \quad (1.61)$$

Isto é, o gradiente da loss é dado pelo gradiente usual da entropia cruzada somado ao termo  $2\lambda\mathbf{w}$ . O método gradiente descendente consiste em realizar a otimização

$$\mathbf{w}' = \mathbf{w} - lr\nabla L(\mathbf{w}) \quad (1.62)$$

$$= \mathbf{w} - lr\nabla E_{ce}(\mathbf{w}) - 2lr\lambda\mathbf{w} \quad (1.63)$$

O termo  $\mathbf{w} - lr\nabla E_{ce}(\mathbf{w})$  é a expressão usual do gradiente descendente. O termo  $-2lr\lambda\mathbf{w}$  representa uma subtração de uma fração do valor de  $\mathbf{w}$  a cada passo do gradiente descendente. Isso tem o efeito de manter os parâmetros em valores pequenos. É mais eficiente implementar a regularização L2 como uma simples subtração de uma fração do parâmetro do que utilizar a Equação 1.60 e ter que derivar a loss em relação aos parâmetros. Por isso os otimizadores do Pytorch aceitam como parâmetro o valor de  $\lambda$ , que é chamado de *weight\_decay*. A Figura 1.4 mostra um exemplo do efeito do parâmetro  $\lambda$  no método de regressão linear.

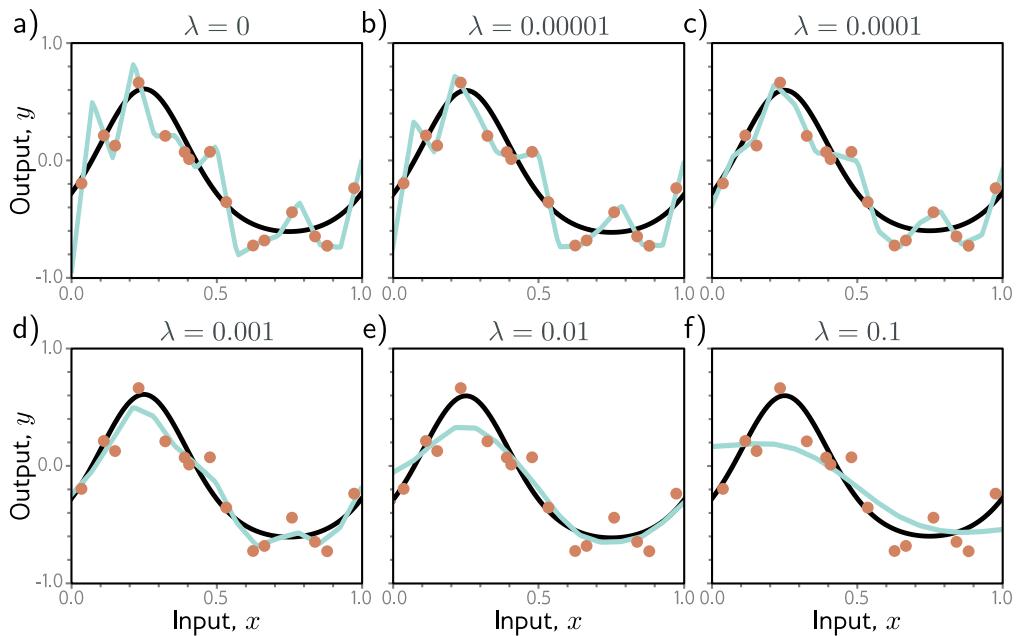


Figura 1.4: Influência da regularização L2 no ajuste de um modelo. A linha preta representa o modelo real que gerou os pontos em marrom. A linha azul representa um modelo que foi ajustado aos dados. Quanto maior a regularização (valor de  $\lambda$ ), mais suave é a função ajustada.

## 1.4 Figuras sobre Regressão Linear e Logística

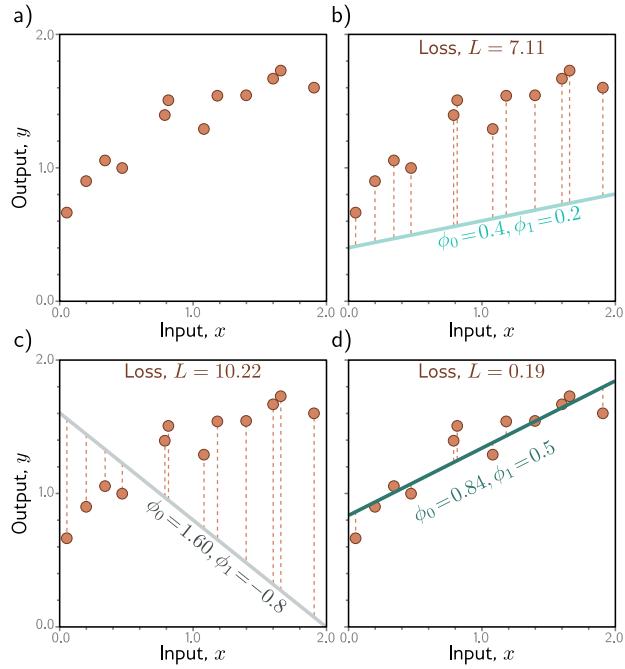


Figura 1.5: Cada gráfico mostra um exemplo de modelo linear com parâmetros  $\phi_0$  e  $\phi_1$ . As linhas tracejadas indicam a qualidade do ajuste para cada ponto.

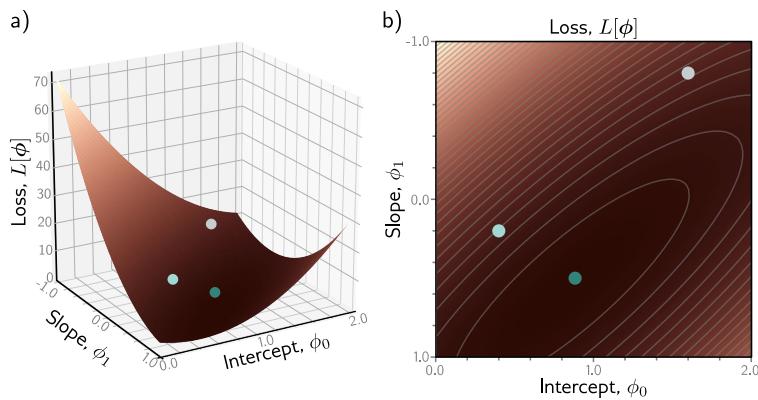


Figura 1.6: Erro quadrático em função do coeficiente angular e linear do modelo.

### 1.4.1 Capacidade de uma rede

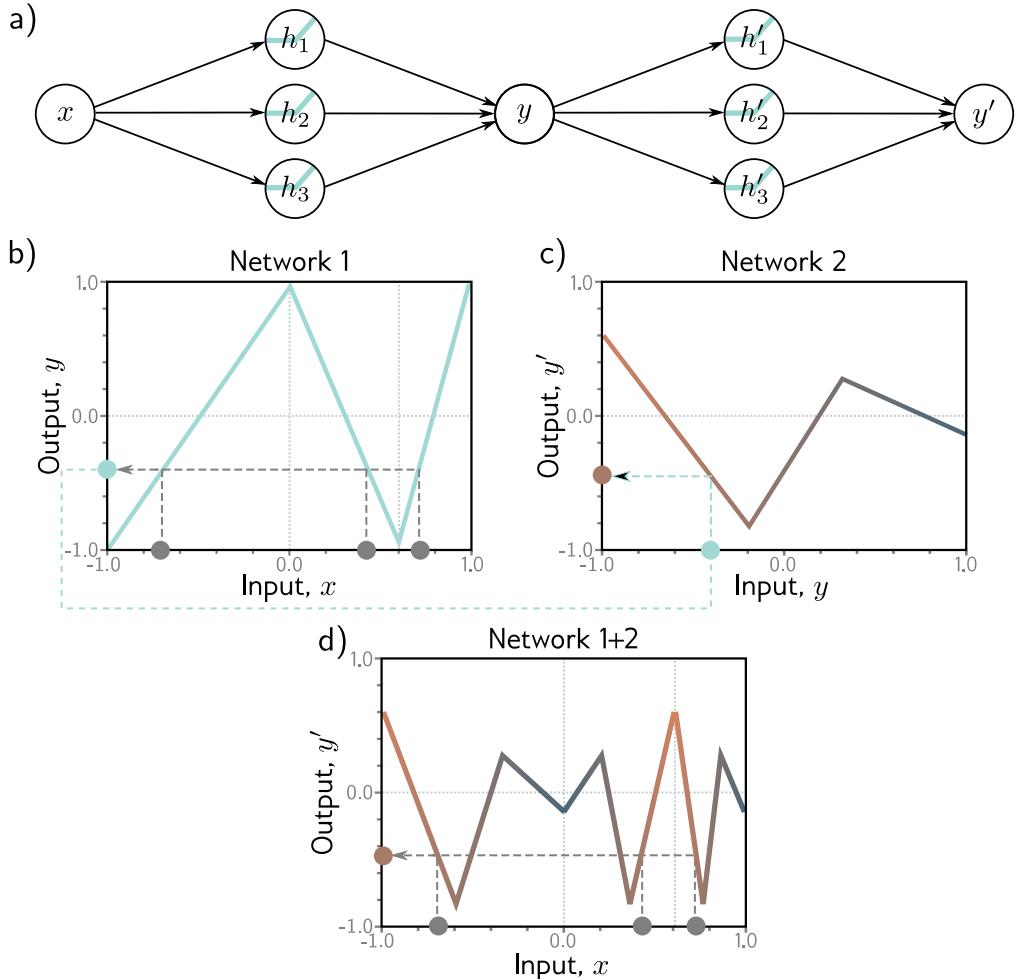


Figura 1.7: Exemplos de funções que podem ser criadas por camadas de uma rede neural. Cada neurônio dessa rede realiza a operação  $f(e) = a_i e + b_i$ , onde  $e$  é a entrada e  $a_i$  e  $b_i$  são parâmetros do neurônio  $i$ . A subrede 1 (network 1) recebe um valor  $x$  e retorna três ativações que são somadas para formar  $y$ . A subrede 2 realiza a mesma operação, mas com o valor  $y$  de entrada. Cada subrede pode criar no máximo 3 regiões lineares. A combinação das duas redes em uma única rede possibilita criar 9 regiões lineares.

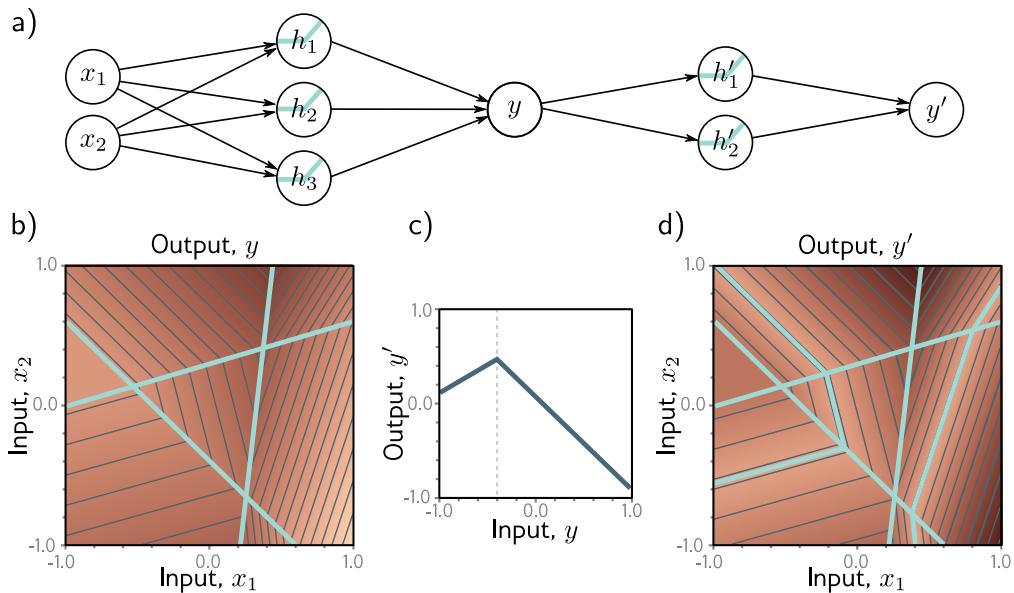


Figura 1.8: Exemplos de funções criadas quando há dois atributos de entrada. A primeira subrede consegue criar apenas 7 regiões (b). A rede inteira consegue criar muito mais (d).

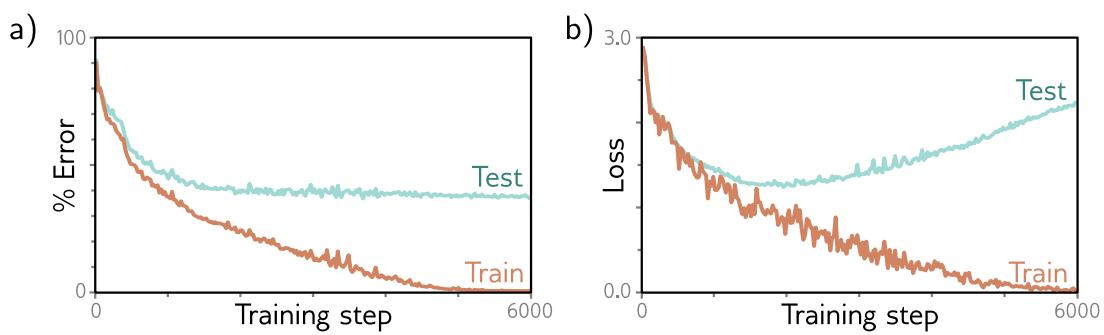


Figura 1.9: (a) Performance típica de um modelo que convergiu. (b) Performance típica de overfitting. Note que o overfit ocorre quando a loss de teste ou validação começa a aumentar. Quanto maior a capacidade de uma rede, maior a possibilidade de overfitting.

## 2 Figuras sobre redes neurais convolucionais (módulos 4 a 8)

### 2.1 Convolução

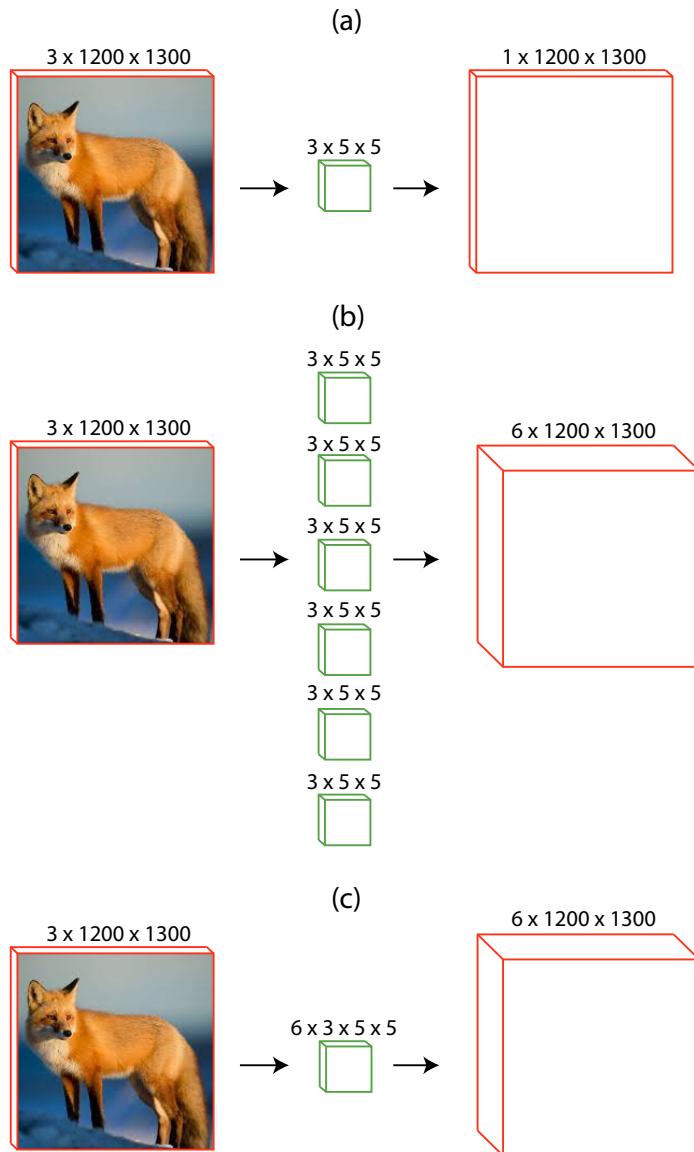


Figura 2.1: Representação dos filtros de uma CNN. (a) Uma imagem de três canais e tamanho  $1200 \times 1300$  é convoluída com um filtro de três canais e tamanho  $5 \times 5$ . O resultado é uma imagem de um único canal, chamada de *ativação* da camada. (b) A mesma imagem é convoluída com 6 filtros de três canais e tamanho  $5 \times 5$  cada. São obtidas 6 imagens possuindo um único canal, essas imagens são organizadas em um tensor possuindo 6 canais. Esse resultado já não pode mais ser interpretado como uma imagem colorida, é mais adequado chamarmos ele de ativação da rede. (c) O mesmo procedimento realizado em (b), mas representamos os 6 filtros por um único tensor de tamanho  $6 \times 3 \times 5 \times 5$ .

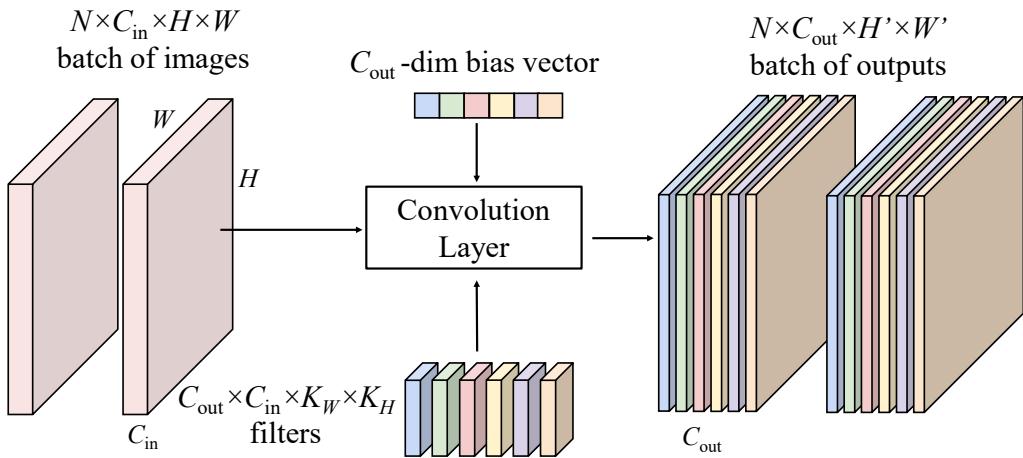


Figura 2.2: Uma camada de convolução completa. A entrada da camada é um batch de  $N$  imagens. Na ilustração está representado um batch de 2 imagens. Cada imagem possui  $C_{in}$  canais e tamanho  $H \times W$ . O batch é um único tensor com tamanho  $N \times C_{in} \times H \times W$ . A camada de convolução possui  $C_{out}$  filtros, cada um com tamanho  $C_{in} \times K_W \times K_H$ . A camada também possui um valor de bias para cada filtro. Todos os filtros são alocados em um único tensor de tamanho  $C_{out} \times C_{in} \times K_W \times K_H$  (note que a primeira dimensão é o número de filtros). Os valores de bias são um tensor com tamanho  $C_{out}$ . Cada filtro “desliza” pelas posições espaciais de cada imagem de entrada, realizando a convolução. O resultado de cada convolução é somado com o respectivo valor de bias. O resultado da camada é um batch de  $N$  imagens. Cada imagem possui  $C_{out}$  canais e tamanho  $H' \times W'$ . O tamanho da imagem de saída depende do padding usado e do stride do filtro. O número de canais  $C_{out}$  em geral é bem alto, por exemplo, 64, 128, 256, etc.

## 2.2 Arquiteturas de CNNs

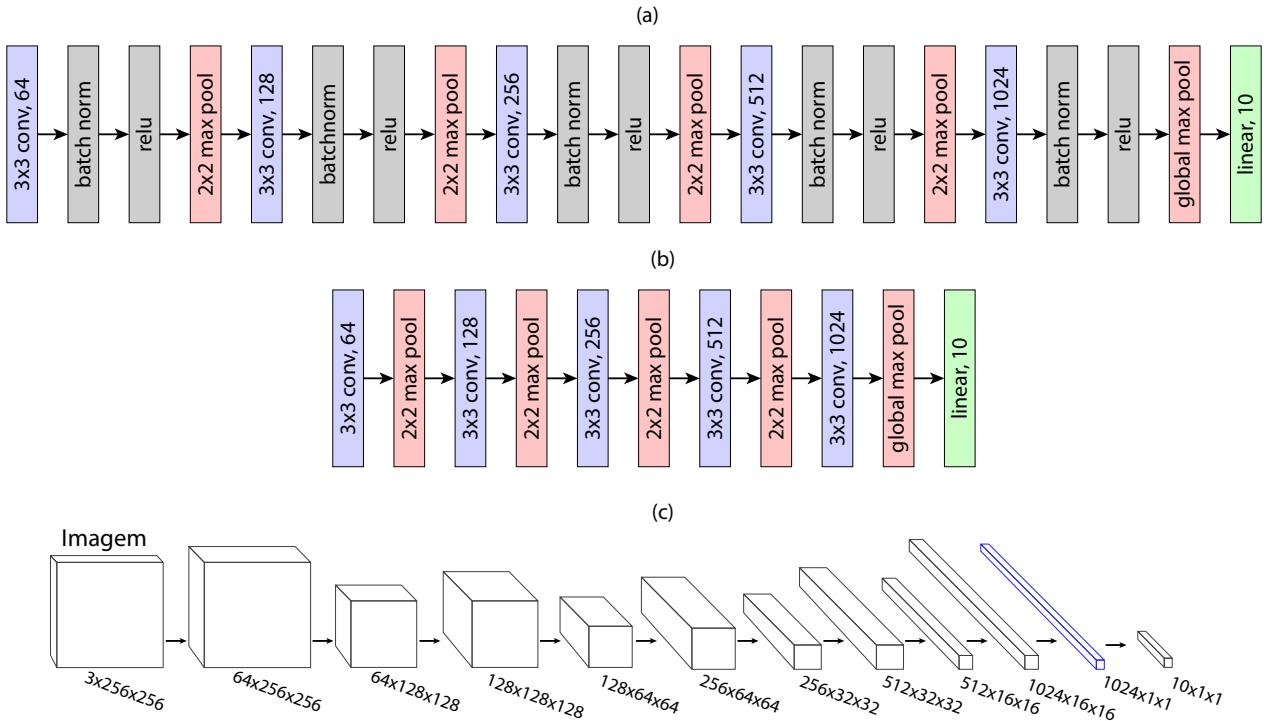


Figura 2.3: Arquitetura de uma CNN. (a) A arquitetura é descrita por uma sequência de operações de convolução, normalização, ativação não linear e pooling. Por exemplo, “3x3 conv, 64” significa que 64 filtros de tamanho 3x3 serão aplicados na entrada. A saída dessa camada será um tensor com 64 canais. A saída de cada convolução é normalizada utilizando o método “batch norm”. A função de ativação não linear “relu” é aplicada no resultado. “2x2 max pool” significa que o valor máximo de regiões de tamanho 2x2 será calculado. É subentendido que será usado stride 2 para a camada e, portanto, a saída terá metade do tamanho espacial da entrada. A penúltima camada realiza um pooling global, que gerará um vetor de 1024 valores. A última camada realiza a classificação em 10 classes. (b) Mesma arquitetura de (a) representada de forma mais sucinta. Praticamente em qualquer arquitetura, após uma convolução sempre é aplicada normalização e uma função não linear. Portanto, é comum que a rede seja representada indicando apenas as convoluções e técnica de pooling. (c) Tensores de saída de cada camada do modelo (b), chamados de ativação das camadas. A primeira camada terá como saída um tensor de tamanho 64x256x256, a segunda um tensor de tamanho 64x128x128, e assim por diante. A última camada, uma camada linear, recebe como entrada um tensor de tamanho 1024 que caracteriza a imagem (mostrado em azul), e realiza a classificação em 10 classes. O tensor de tamanho 1024 representa 1024 características que foram extraídas da imagem de entrada.

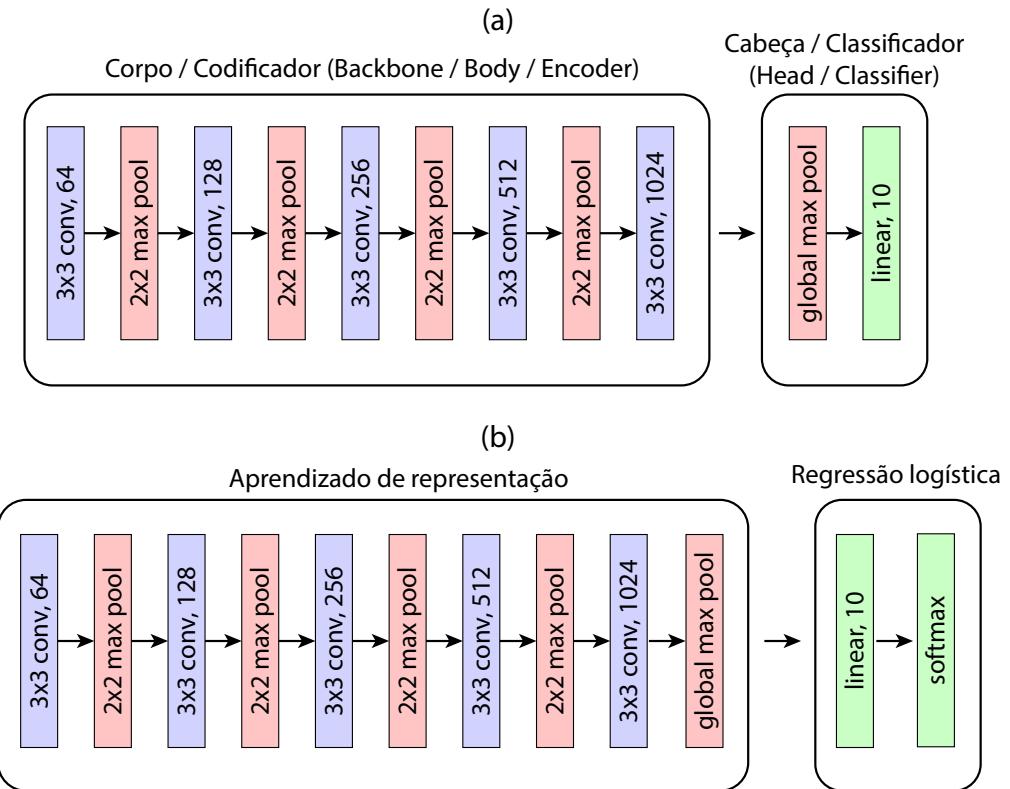


Figura 2.4: Partes de uma CNN. (a) As camadas logo antes do pooling global são chamadas de *corpo* da rede ou *codificador* (do inglês backbone e encoder). As camadas finais são chamadas de *cabeça* ou *classificador*. (b) Uma rede neural pode ser entendida como um sistema que aprende uma representação apropriada para um conjunto de dados. A camada final seguida de uma função softmax (ou logística) é um classificador simples aplicado nessa representação. Portanto, podemos dizer que redes neurais são usadas para realizarmos um *aprendizado de representação* dos dados.



Figura 2.5: Visualização das 9 regiões de imagens que mais ativaram filtros específicos de uma CNN treinada no dataset ImageNet. São mostradas regiões para filtros de diferentes camadas da rede, começando da mais próxima da entrada (camada 1) até a camada mais próxima da saída (camada 5). Para a camada 1 são mostradas as regiões de 9 filtros distintos. Para as camadas 2 e 3 são mostradas regiões de 16 filtros. Para as camadas 4 e 5 é mostrado o resultado para 10 filtros. Note que os filtros iniciais possuem maior resposta para detalhes locais das imagens como bordas e cores. Filtros mais profundos detectam atributos de maior escala como rostos, objetos circulares ou animais. Figura adaptada de *M.D. Zeiler, Visualizing and understanding convolutional networks, 2014*.

## 2.3 Redes Residuais (ResNets)

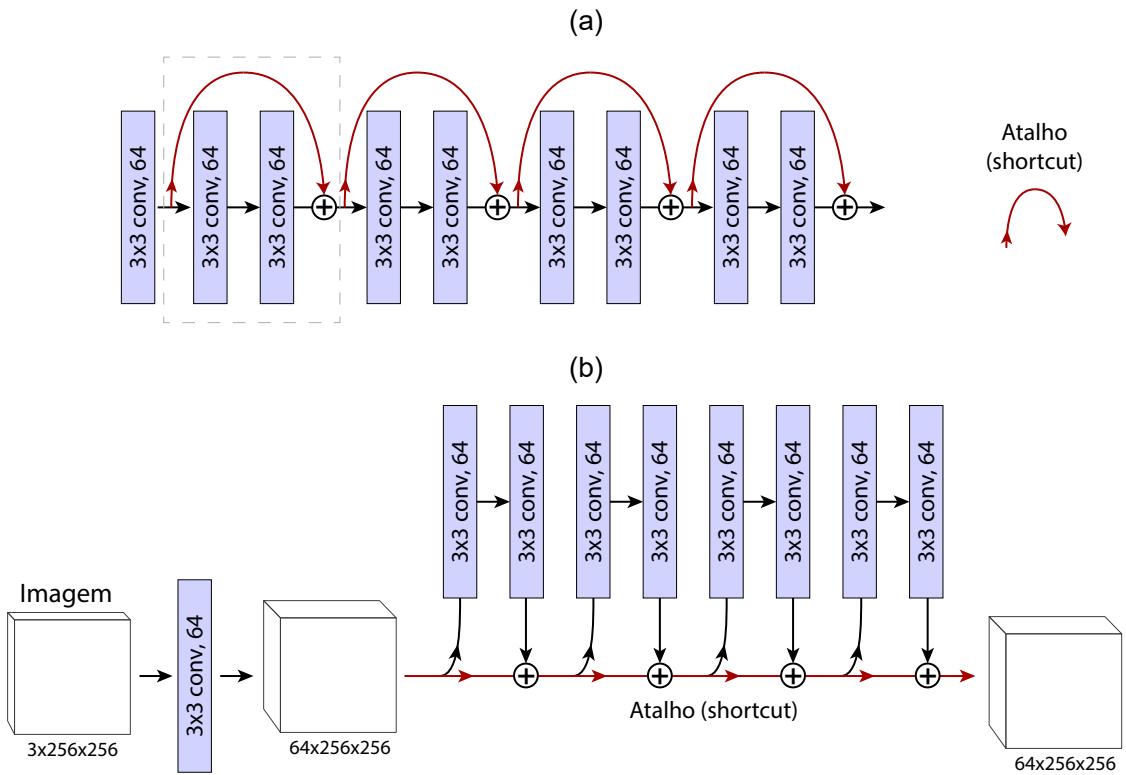


Figura 2.6: Ilustração de redes residuais. (a) Exemplo de uma rede residual composta por 4 blocos residuais. O primeiro bloco residual está indicado pelo retângulo tracejado. O bloco consiste em realizar duas convoluções (seguidas por normalização e ativação não linear). A saída das convoluções é *somada* com o tensor de entrada do bloco. O processo é repetido para cada bloco. (b) Uma outra forma de visualizar a mesma arquitetura de (a). Após a primeira convolução, a rede aprende o que deve ser modificado em relação à entrada para que seja obtida uma saída desejada. De forma similar, cada bloco residual aprende o que deve ser modificado em relação à entrada. Esse paradigma permite uma melhor propagação dos gradientes na rede pois existe um caminho direto entre a entrada e saída dos dados. Adicionalmente, sem qualquer treinamento a rede já possui capacidade de representar a função identidade, que apenas copia a entrada para a saída. Portanto, dizemos que cada bloco aprende o *resíduo* dos dados, isto é, o que deve ser mudado em relação à entrada.

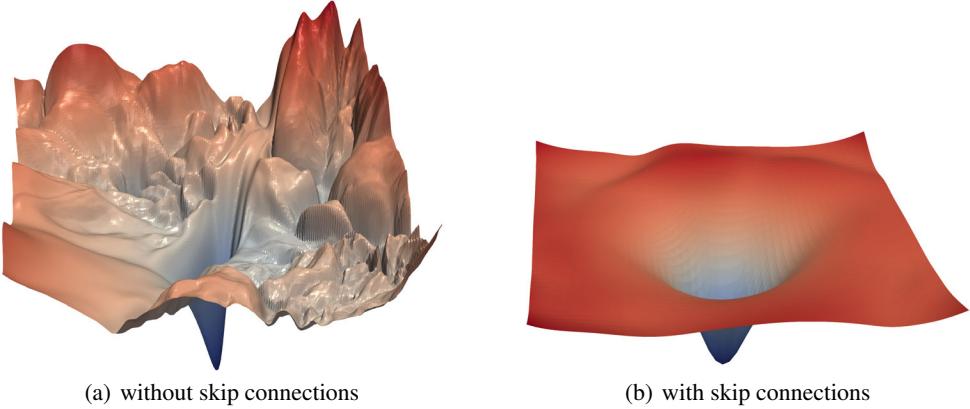


Figura 2.7: Efeito de caminhos residuais na função de perda (loss). (a) Visualização da função de perda sem caminhos residuais. (b) Visualização da mesma função com caminhos residuais. Li, Hao, et al. Visualizing the loss landscape of neural nets. Advances in neural information processing systems 31 (2018).

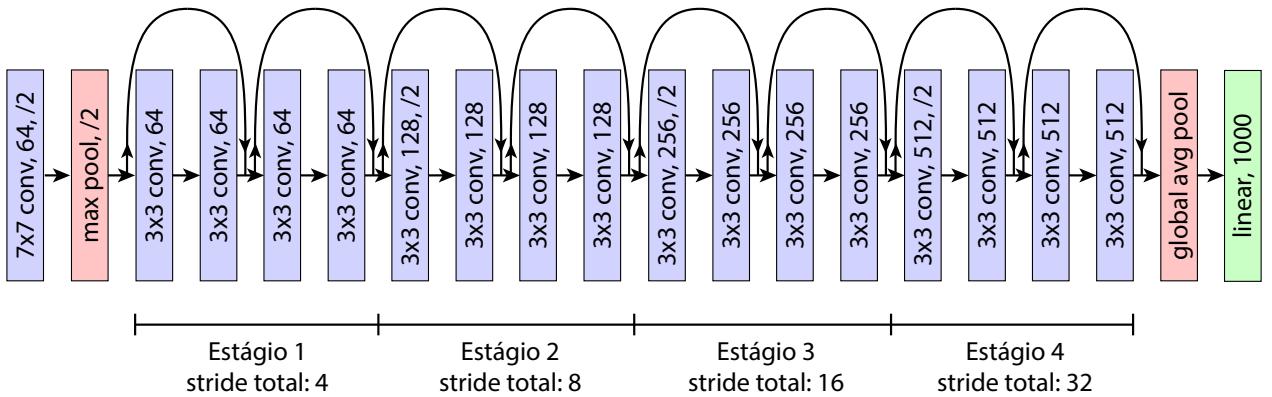


Figura 2.8: Arquitetura ResNet 18. Modelos da família ResNet são divididos em 4 estágios. Cada estágio reduz a resolução pela metade e dobra o número de filtros. Na figura, “/2” indica camadas que possuem stride 2 e portanto reduzem a resolução pela metade. Logo no início do modelo a resolução é reduzida em 4x por uma camada de convolução de filtro 7x7 e uma camada de max pooling, ambas com stride 2. O stride total de um estágio indica o nível de resolução no qual o estágio opera. Ele é dado por  $2^k$ , onde  $k$  é o número de camadas com stride 2 que foram aplicadas anteriormente. Por exemplo, no quarto estágio as ativações terão tamanho 32x menor do que a imagem de entrada. Se a imagem de entrada possuir tamanho 256x256, a saída de cada camada do estágio 4 terá tamanho 8x8. Uma operação de pooling global gera um vetor de atributos de tamanho 512, e uma camada linear usa esses atributos para classificar a imagem em 1000 classes distintas. As 1000 classes utilizadas por padrão são as classes do dataset ImageNet.

## 2.4 Decodificadores

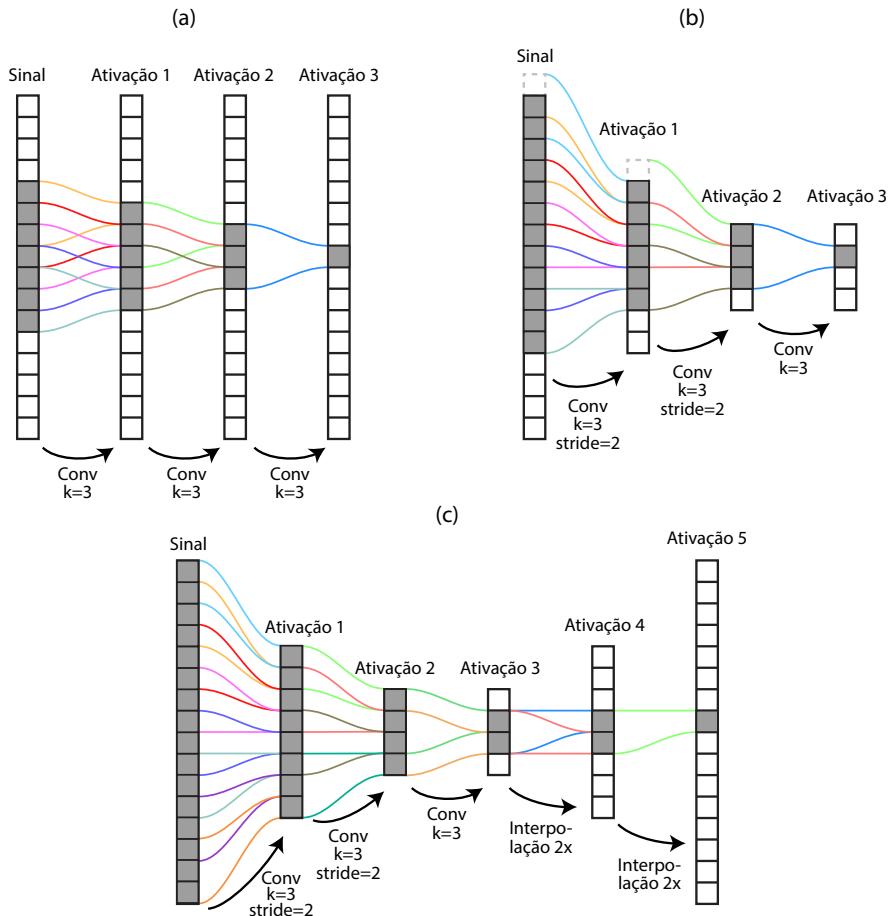


Figura 2.9: Campo receptivo de redes neurais. A ilustração mostra a ativação de diferentes camadas para um sinal 1D. (a) Ativações de três camadas de convolução possuindo filtros de tamanho  $k=3$ . Um pixel da ativação 3 é formado por 3 pixels da ativação 2, pois o filtro possui tamanho 3. Cada pixel da ativação 2 é formado por 3 pixels da ativação 1. Portanto, o pixel da ativação 3 é influenciado por 5 pixels diferentes da ativação 1. Como cada pixel da ativação 1 também é formado por 3 pixels do sinal original, o pixel da ativação 3 é influenciado pelos valores de 7 pixels distintos do sinal de entrada. Portanto, dizemos que o campo receptivo de cada pixel da ativação 3 possui tamanho 7. (b) Ao modificarmos as convoluções para possuírem stride 2, o campo receptivo de cada pixel da ativação 3 passa a ser maior. Nesse caso, ele possui tamanho 12. (c) Após inclusão de camadas de interpolação linear, cada pixel de uma ativação após uma interpolação é influenciado por 2 pixels da ativação anterior (uma propriedade da interpolação linear). Nesse caso, um pixel da ativação 5 possui um campo receptivo de tamanho 16, ou seja, todo o sinal de entrada influencia o valor do pixel.

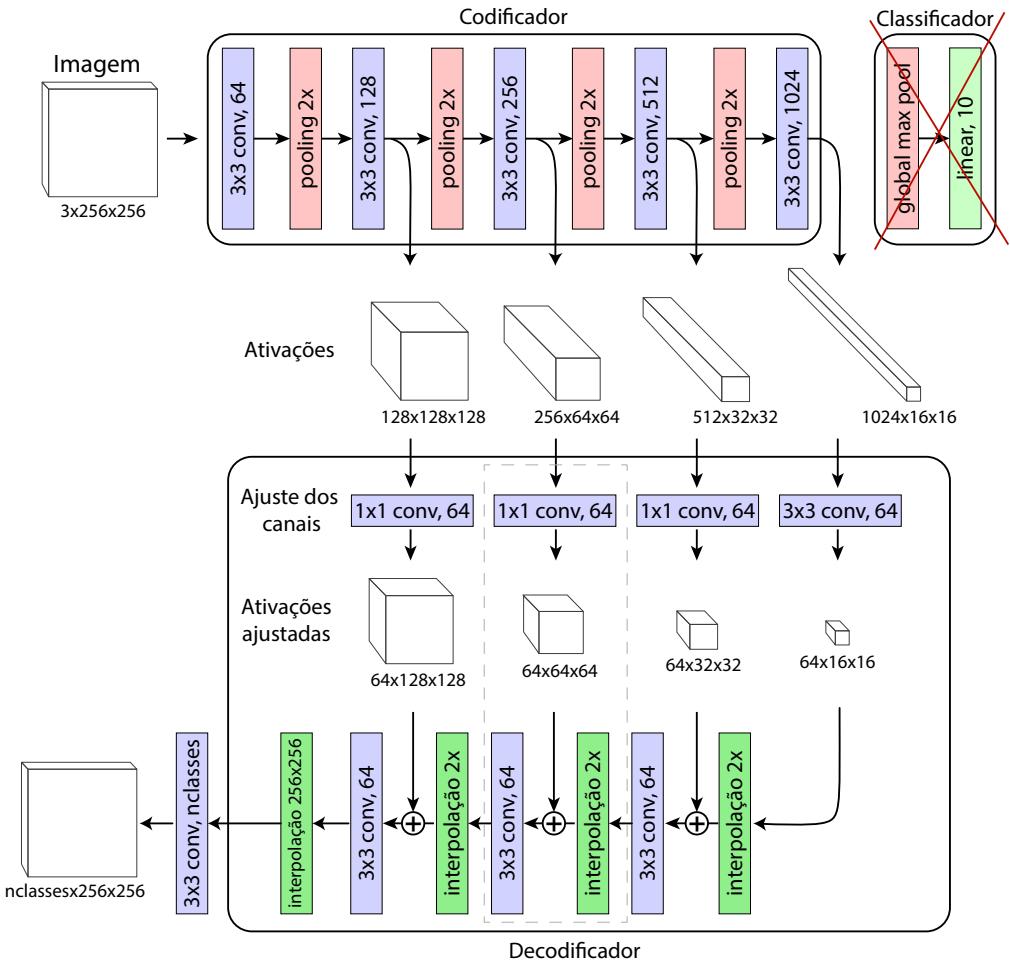


Figura 2.10: Ilustração de um decodificador. Um codificador é utilizado para extrair atributos de uma imagem em múltiplas escalas espaciais. Os atributos são dados por ativações intermediárias do codificador. Um decodificador recebe esses atributos e realiza convoluções para combiná-los e gerar atributos para cada pixel da imagem. A combinação é feita através de blocos de decodificação (um bloco é indicado pelo retângulo tracejado). Cada bloco recebe a ativação do codificador em uma escala e também a ativação do bloco de decodificação anterior. As ativações possuem tamanhos espaciais e número de canais distintos. Uma interpolação e uma convolução 1x1 são utilizadas para ajustar as dimensões das ativações. Após o ajuste, as duas ativações são somadas e passam por uma nova camada de convolução. A última camada do decodificador interpola o resultado para ter o mesmo tamanho espacial da imagem original. Um classificador é aplicado na saída do decodificador para classificar cada pixel. Na ilustração, o decodificador foi aplicado a partir da resolução 128x128 das ativações, e o número de canais utilizado pelo decodificador foi 64, mas é possível variar esses parâmetros.

### 3 Mecanismo de atenção e transformers (módulo 9)

Considere a tarefa de traduzir um texto do português para o inglês. Uma estratégia básica que pode ser utilizada é a criação de um grande dicionário contendo a tradução de cada palavra da língua portuguesa para a inglesa. Por exemplo

```
portugues_ingles = {  
    casa: house,  
    valor: value,  
    bateria: battery,  
    ...  
}
```

Nesse caso, as palavras em português representam *chaves* do dicionário e as palavras em inglês representam os respectivos *valores*. Um dicionário pode ser implementado como uma tabela hash. Portanto, a tradução de uma palavra é extremamente eficiente. Dada uma palavra qualquer em português, uma função de hash é aplicada na palavra, e o valor respectivo é recuperado.

Essa estratégia possui diversos problemas, um deles sendo o fato de que uma mesma palavra pode possuir diversas traduções possíveis. Por exemplo, considere a palavra [bateria](#). Ela pode ser traduzida para [battery](#), que é o nome de um dispositivo para armazenamento de energia, mas também pode ser traduzida para [drums](#), que é um instrumento musical. A tradução depende do contexto no qual a palavra foi utilizada. Para possibilitar a adição de contexto, primeiramente substituímos a função de hash por duas funções: uma função de *codificação*, que designa um código único para cada palavra, e uma função de *similaridade*, que compara a codificação entre as palavras. Na codificação, substituímos o dicionário de tradução por um dicionário de *embedding*, que designa um vetor de valores para cada palavra:

```
portugues = {  
    casa: (1.2, 0.3, -4.5),  
    valor: (-4.3, -2.3, 1.4),  
    bateria: (3.1, 6.2, 0.4),  
    ...  
}  
  
inglês = {  
    house: (1.6, 0.5, -3.5),  
    value: (-4.4, -2.7, 1.1),  
    battery: (3.0, 5.8, 0.7),  
    drums: (3.3, 6.5, -0.2),  
}
```

...

}

Para traduzir uma palavra de uma língua, buscamos as palavras mais similares a ela no dicionário da outra língua. Note que devido à função de similaridade, os códigos não precisam ser idênticos. Basta que eles sejam suficientemente similares, isso possibilita o uso de codificações compactas. Veremos mais adiante como definir a codificação. Para medir a similaridade, é comum o uso da similaridade de cosseno:

$$s_{ij} = \frac{\sum_k \mathbf{x}_i(k) \mathbf{x}_j(k)}{|\mathbf{x}_i| |\mathbf{x}_j|} \quad (3.1)$$

onde  $\mathbf{x}_i(k)$  é  $k$ -ésimo valor do vetor  $\mathbf{x}_i$  e  $|\mathbf{x}_i|$  é a norma (tamanho) do vetor. Por exemplo, na codificação apresentada acima a similaridade entre **bateria** e **battery** será

$$\begin{aligned} s_{ij} &= \frac{3.1 * 3.0 + 6.2 * 5.8 + 0.4 * 0.7}{\sqrt{3.1^2 + 6.2^2 + 0.4^2} \sqrt{3.0^2 + 5.8^2 + 0.7^2}} \\ &= 0.9987 \end{aligned}$$

A similaridade entre **bateria** e **house** será

$$\begin{aligned} s_{ij} &= \frac{3.1 * 1.6 + 6.2 * 0.5 - 0.4 * 3.5}{\sqrt{3.1^2 + 6.2^2 + 0.4^2} \sqrt{1.6^2 + 0.5^2 + -3.5^2}} \\ &= 0.2471 \end{aligned}$$

Adicionalmente, a similaridade entre **bateria** e **drums** é calculada como 0.9964. Com isso, é possível associarmos diferentes conceitos e definirmos um grau de similaridade entre eles. Para essa estratégia funcionar é preciso que palavras com significados *semânticos* similares sejam associadas a vetores de atributos similares.

Essa nova forma de representar os dados pode ser utilizada para traduzir uma palavra dependendo do contexto no qual ela é utilizada. Considere que queremos traduzir a seguinte sentença

“Ele colocou a bateria no depósito. Não fazia sentido mantê-la no quarto pois ele não a tocava há um ano.”

A palavra **bateria** deve ser traduzida para **drums** por causa da presença da palavra **tocava**. Portanto, um sistema de tradução precisa identificar a presença do termo **tocava** para traduzir a palavra **bateria**. Veremos como construir um sistema que realize tal tarefa.

Primeiramente, considere que todas as palavras da língua inglesa e portuguesa, e as respectivas codificações, estão armazenadas em um banco de dados, que também pode ser considerado como um banco de memória. Inicialmente, essa codificação pode não ser adequada, veremos que será possível aprimorar a codificação através de um processo de otimização.

Mas para simplificar vamos considerar que há uma codificação preliminar boa o bastante para esse sistema fazer algumas previsões corretas. A codificação utiliza apenas 2 valores para representar cada palavra. Suponha que o banco de memória atual seja o seguinte:

	...
bateria	(0.43, 0.82)
tocava:	(0.26, 0.89)
battery:	(0.51, 0.77)
drums:	(0.35, 0.86)
energia:	(0.58, 0.72)
	...

Note que `bateria`, `battery` e `drums` possuem códigos relativamente similares. A similaridade de cosseno entre `bateria` e `battery` é 0.9948 enquanto que a similaridade entre `bateria` e `drums` é 0.9954. Cada palavra da sentença a ser traduzida é transformada no respectivo código. Chamaremos de  $\mathbf{k}_i$  o código da  $i$ -ésima palavra. Note que cada vetor  $\mathbf{k}_i$  possui apenas 2 valores no nosso exemplo, mas em geral eles são bem maiores. É comum chamar a quantidade de valores dos vetores  $\mathbf{k}_i$  de *embed\_dim*.

Para traduzir a palavra `bateria`, ela é tratada como uma consulta em uma base de dados. Ela será a *query*, e representada por um vetor  $\mathbf{q}$ . A similaridade entre `bateria` e cada palavra da sentença é calculada como

$$s_i = \mathbf{q} \mathbf{k}_i \quad (3.2)$$

Os valores  $s_i$  são então normalizados por uma função softmax <sup>1</sup>. Os valores  $s_i$  são chamados de *valores de atenção*, eles indicam a relação entre a palavra sendo analisada e as demais palavras presentes no *contexto* da palavra de interesse. Após isso, a representação da palavra `bateria` é atualizada como uma combinação ponderada das representações de todas as palavras do contexto:

$$\mathbf{q}' = \mathbf{q} + \sum_i s_i \mathbf{k}_i \quad (3.3)$$

Quanto maior a similaridade com a palavra, maior será o peso dela na atualização da representação de `bateria`. Para entendermos porque essas equações são úteis, vamos ignorar a similaridade da palavra `bateria` com outras palavras além de `tocava`. Vamos supor que após aplicar a Equação 3.2 entre a palavra `bateria` e as demais palavras da sentença, e normalizar os valores de  $s_i$ , a similaridade entre `bateria` e `tocava` é igual a 0.9 <sup>2</sup>. Com isso, a Equação 3.3 pode ser escrita como

<sup>1</sup>Lembrando que a função softmax basicamente normaliza os valores para que eles tenham soma 1. De forma similar poderíamos normalizar  $s_i$  como  $s'_i = \frac{s_i}{\sum_j s_j}$ .

<sup>2</sup>Portanto, a soma das similaridades entre `bateria` e as demais palavras da sentença, excluindo `tocava`, é igual a 0.1.

$$q' = (0.43, 0.82) + 0.9 * (0.26, 0.89) + \dots$$

Ignorando os demais termos da soma, a nova representação de **bateria** é dada por  $(0.664, 1.621)$ . A similaridade cosseno entre essa nova codificação de **bateria** e as palavras **battery** e **drums** agora é dada por, respectivamente,  $0.9808$  e  $0.9999$ . Portanto, **battery** se tornou mais similar com **drums** do que com **battery**. Se a sentença possuísse a palavra **energia** ao invés de **tocava**, teríamos que

$$\begin{aligned} q' &= (0.43, 0.82) + 0.9 * (0.58, 0.72) + \dots \\ &= (0.952, 1.468) \end{aligned}$$

Nesse caso, a similaridade cosseno entre **bateria** e as palavras **battery** e **drums** passaria a ser  $0.9999$  e  $0.9822$ . Portanto, **bateria** se tornaria mais similar à **battery**.

Os cálculos que vimos acima correspondem ao *mecanismo de atenção*. Cada palavra atende às demais palavras do contexto, o que define valores de ponderação, e a representação de cada palavra é atualizada como uma soma ponderada das representações de todas as palavras do contexto. Podemos imaginar que o processo nada mais é do que uma *mistura* das codificações das palavras do contexto, da mesma forma que uma convolução é uma *mistura* dos píxeis de cada região de uma imagem. Note que o sistema não precisa necessariamente atualizar as codificações das palavras no banco de memória. As novas codificações podem ser utilizadas para realizar previsões de acordo com o contexto da sentença, e as codificações originais serem mantidas no banco de memória. Em outras palavras, é possível criar codificações adequadas dependendo do contexto no momento da *inferência*, sem qualquer atualização de parâmetros.

Em uma camada de atenção, as equações 3.2 e 3.3 são aplicadas em todas as palavras ao mesmo tempo, considerando cada palavra como uma query diferente. Isso é feito da seguinte forma. É definida uma matriz  $Q$ , possuindo em cada linha a codificação de uma palavra. No nosso exemplo essa matriz teria tamanho  $20 \times 2$ , pois a sentença possui 20 palavras representadas por vetores de 2 valores. Essa é a matriz de queries que serão feitas. No caso de atenção que estamos analisando, são criadas também as matrizes  $K = Q$  e  $V = Q$ , que são idênticas à matriz  $Q$ , mas em outras situações elas podem não ser iguais. A equação matricial equivalente à Equação 3.2 é

$$S = QK^T \tag{3.4}$$

onde  $K^T$  é a transposta da matriz  $K$ . Cada linha da matriz  $S$  é então normalizada pela

função softmax <sup>3</sup>. As linhas de  $S$  representam a similaridade entre uma dada query e as demais palavras do contexto. Ela é chamada de *matriz de atenção*. As codificações são então atualizadas como

$$Q' = SV \quad (3.5)$$

que é a forma matricial da Equação 3.3.

O mecanismo de atenção permite que contexto seja adicionado à codificação das palavras. Mas é interessante notar que ele não possui parâmetros treináveis! Os únicos valores que podem ser treinados pelo método gradiente descendente são dados pela codificação inicial atribuída à cada palavra antes da aplicação do mecanismo de atenção. Por isso, é comum a aplicação de uma camada linear antes ou depois do mecanismo de atenção. Essa camada permite que parâmetros treináveis sejam otimizados para aprimorar a codificação das palavras. A aplicação da camada nas codificações pode ser representada como

$$A = QW \quad (3.6)$$

onde  $W$  é a matriz de parâmetros da camada. Por causa da maneira que essa camada é aplicada, os mesmos parâmetros são utilizados para cada linha da matriz  $Q$  (cada palavra, ou query). Por isso, essa camada é usualmente chamada de *feed-forward* ao invés de linear <sup>4</sup>. A arquitetura transformers possui alguns elementos adicionais que são explicados nos notebooks da disciplina, mas os elementos essenciais são os que vimos nesta seção: camadas de atenção e camadas feed-forward.

No caso de imagens o procedimento é exatamente o mesmo. Uma imagem é transformada em uma matriz  $Q$  na qual cada linha representa um *patch*, ou região, da imagem e as colunas representam a codificação atribuída para cada região. Nos notebooks da disciplina veremos como isso é feito. O mecanismo de atenção consiste em atualizar a codificação das regiões da imagem de acordo com o contexto no qual as regiões ocorrem. No caso de imagens, o contexto usualmente é a imagem toda.

<sup>3</sup>Em Pytorch isso é feito como `torch.softmax(S, dim=1)`.

<sup>4</sup>Uma camada linear usual seria aplicada em um vetor como  $\mathbf{q}W$ . Uma camada feed-forward representa uma camada linear aplicada em um batch de vetores representados pela matriz  $Q$ .

### 3.1 Figuras sobre implementação de atenção

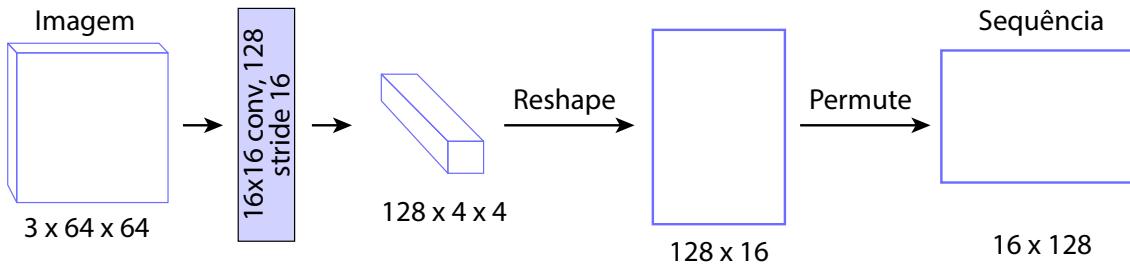


Figura 3.1: Ilustração do processo de transformar uma imagem em uma sequência de tokens. Uma convolução com filtros de tamanho  $16 \times 16$  e stride  $16 \times 16$  é aplicada na imagem. Essa convolução não possui padding. Efetivamente, o filtro será aplicado em regiões disjuntas de tamanho  $16 \times 16$  na imagem. Nessa configuração, o filtro possui 16 posições possíveis na imagem (4 para cada linha e 4 para cada coluna). A camada possui 128 filtros e, portanto, será gerado um tensor de tamanho  $128 \times 4 \times 4$ . É realizada uma operação de reshape no resultado para unir as duas últimas dimensões, gerando uma matriz de tamanho  $128 \times 16$ . Por fim, as dimensões são permutadas e a matriz final possuirá tamanho  $16 \times 128$ . Essa matriz pode ser interpretada da seguinte forma: cada linha representa uma região distinta da imagem (16 regiões), e as colunas representam características extraídas da respectiva região. As linhas são também chamadas de *tokens*, em referência à mesma representação usada em processamento de linguagem natural. Dessa forma, a matriz resultante representa uma *sequência* de tokens.

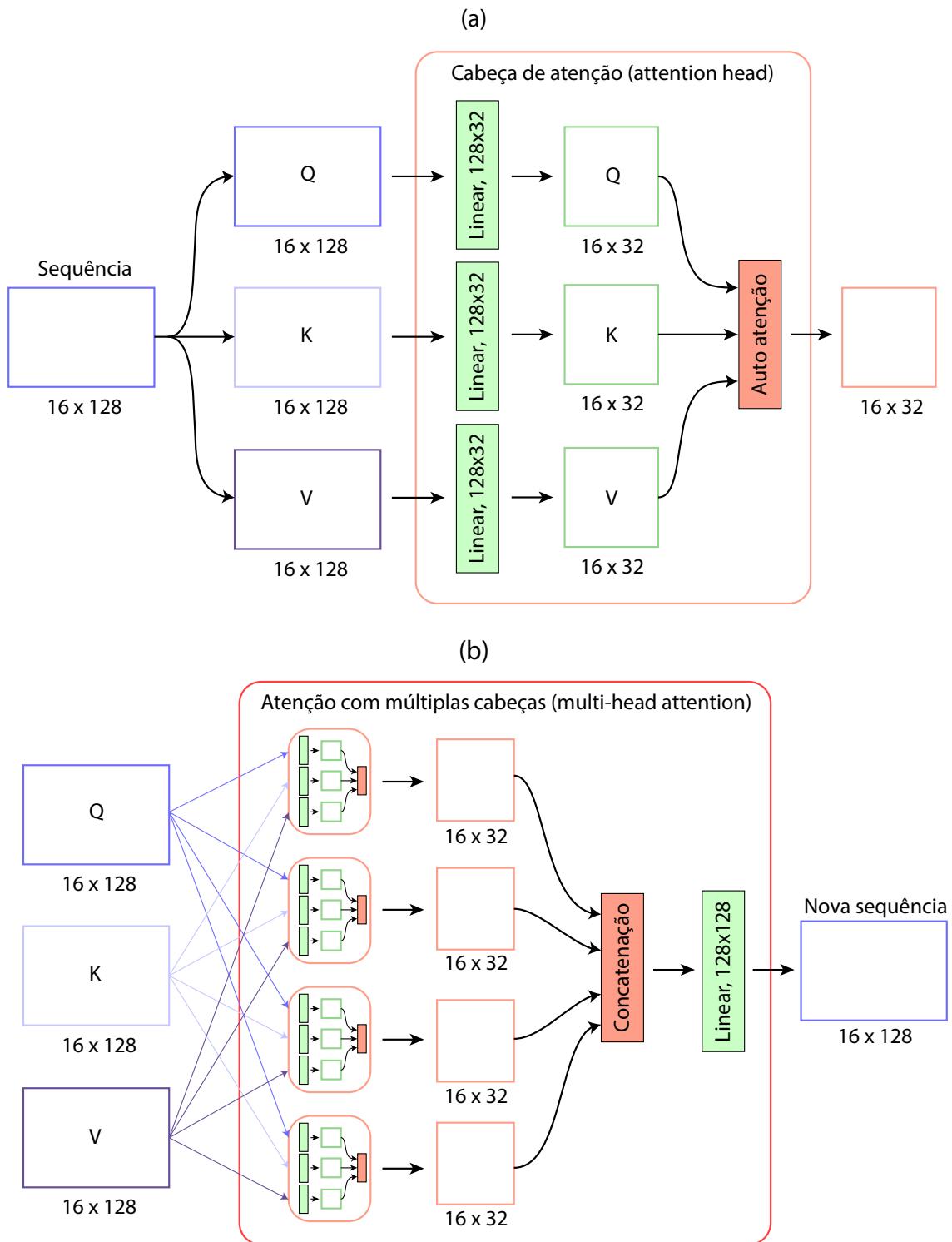


Figura 3.2: (a) Exemplo de uma cabeça de auto atenção. Uma sequência de tokens é copiada três vezes, gerando as matrizes Q (query), K (key) e V (value). Camadas lineares distintas são aplicadas em cada matriz. No exemplo, cada camada gera 32 values para cada token de entrada. Portanto, serão geradas 3 ativações de tamanho  $16 \times 32$ . O mecanismo de atenção é aplicado nas ativações, gerando uma matriz de saída de tamanho  $16 \times 32$ . (b) Exemplo de atenção com múltiplas cabeças. O procedimento mostrado em (a) é aplicado 4 vezes nas mesmas matrizes de entrada. Nesse caso dizemos que a camada possui 4 cabeças. Cada cabeça gera uma matriz de tamanho  $16 \times 32$ , que são concatenadas para gerar uma matriz de tamanho  $16 \times 128$ . Uma camada linear é então aplicada no resultado, gerando uma nova versão da sequência na qual foi dado contexto para cada token.

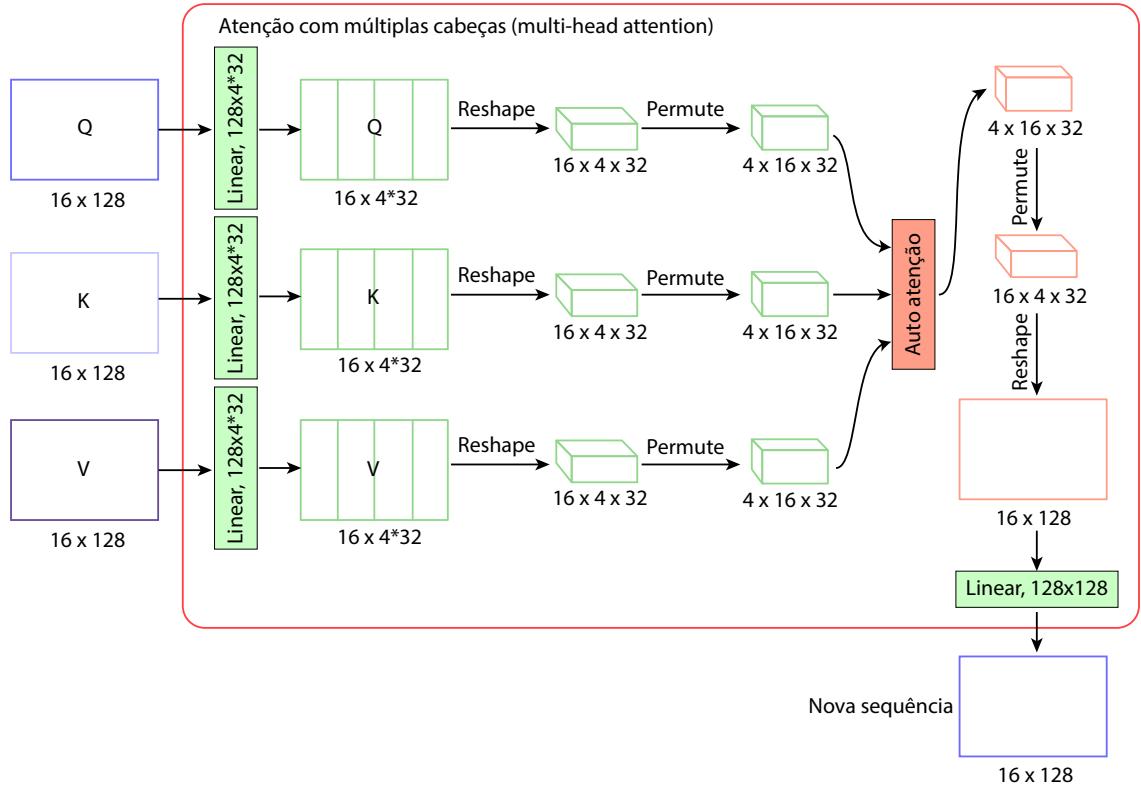


Figura 3.3: Implementação eficiente de atenção com múltiplas cabeças. Três camadas lineares são aplicadas nas entradas. Cada camada linear gera  $4 * 32 = 128$  novos valores para cada token. Essas camadas consistem na aplicação das 4 cabeças mostradas na Figura 3.2(b) em uma única multiplicação matricial. Uma operação de reshape é aplicada em cada resultado, gerando tensores de tamanho  $16 \times 4 \times 32$ , seguida de uma operação permute, gerando tensores de tamanho  $4 \times 16 \times 32$ . Essa configuração de dimensões permite aplicar a camada de auto atenção tratando a primeira dimensão (de tamanho 4) como se fosse uma dimensão de batch. O resultado é permutado e redimensionado para  $16 \times 128$ . Uma camada linear final é aplicada, gerando uma nova sequência de tamanho  $16 \times 128$  na qual os tokens foram alterados para a inclusão de contexto.

## 4 Figuras sobre processamento multimodal (módulo 10)

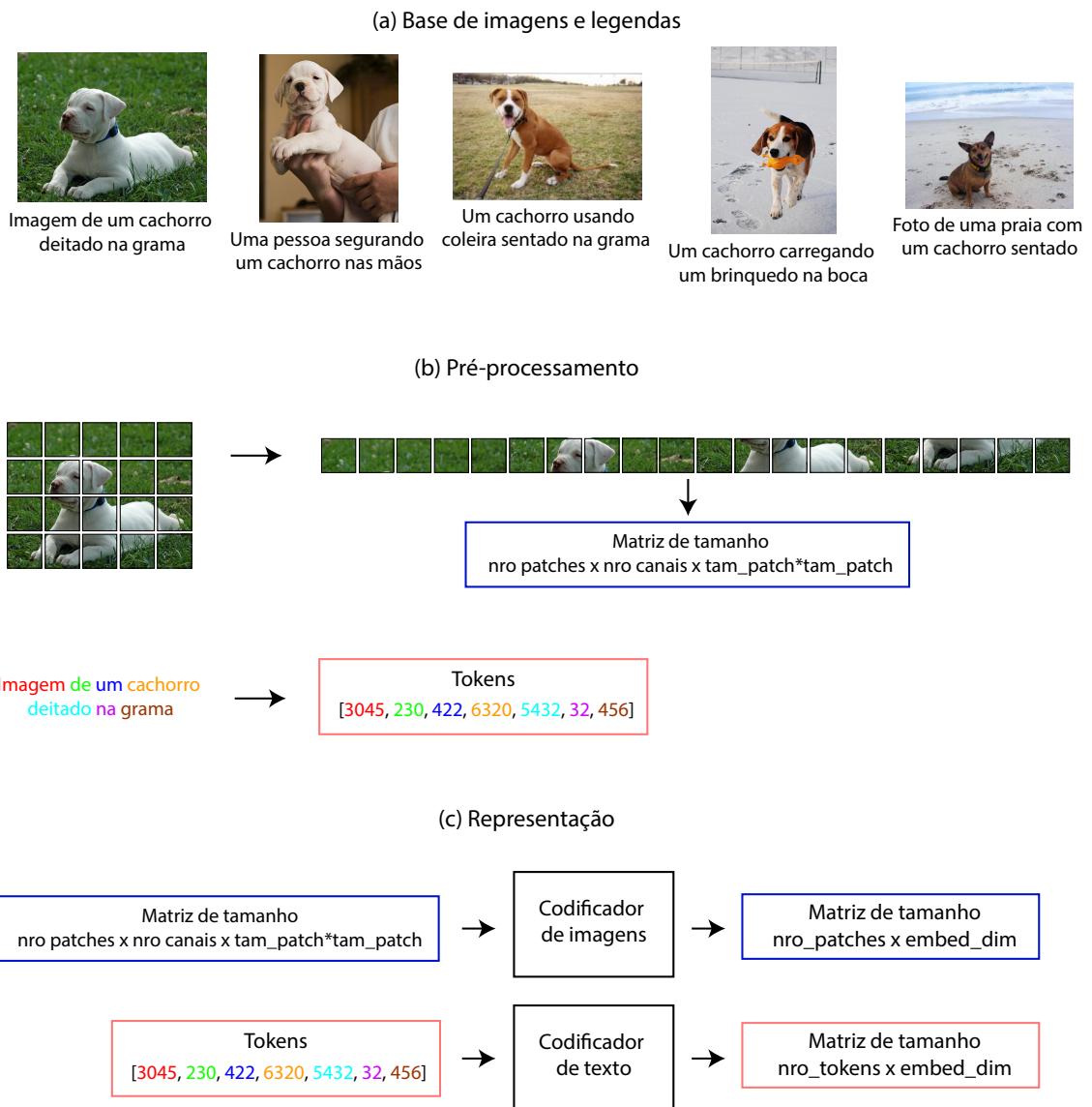


Figura 4.1: Visão geral sobre processamento multimodal, no caso, textos e imagens. (a) Os dados consistem em imagens e textos associados a cada imagem. (b) Na etapa de pré-processamento, a imagem é dividida em regiões (patches) de tamanho  $\text{tam\_patch}$  x  $\text{tam\_patch}$ , e transformada em uma sequência. A sequência terá uma tamanho igual ao número de patches criados. O respectivo texto é transformado em tokens, que consiste em substituir conjuntos de caracteres por índices. Neste exemplo cada palavra é representada por um índice distinto. (c) Uma rede neural é aplicada na imagem (ResNet, ViT, etc) e outra é aplicada no texto (Transformer padrão, BERT, etc) para gerar representações adequadas dos dados. Essas representações podem ser utilizadas em diversas tarefas.

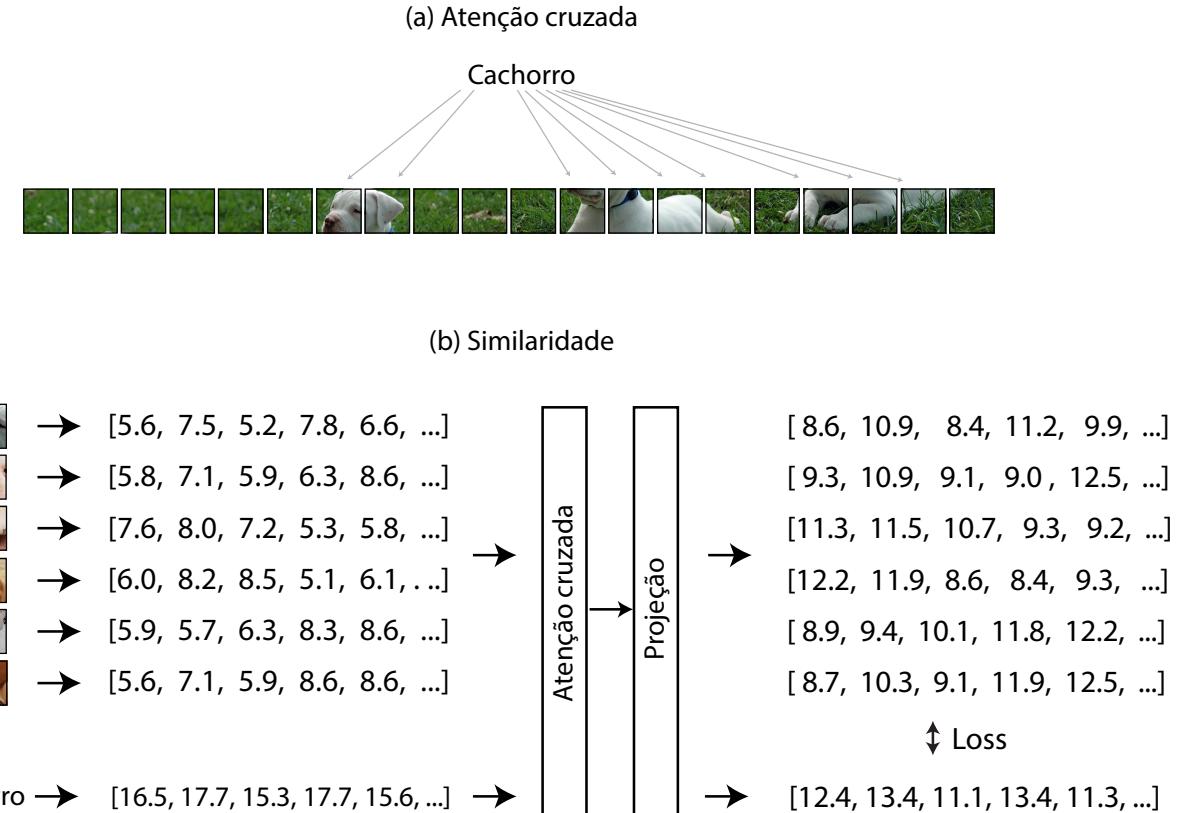


Figura 4.2: Exemplo de alinhamento entre representações de imagens e textos. (a) Uma camada de atenção é aplicada na representação da palavra “cachorro” e de uma imagem de entrada. Na ilustração as representações são mostradas como texto e imagem para visualização, mas essas representações são na verdade matrizes de valores. A palavra “cachorro” é tratada como query da camada e os patches são tratados como keys e values. A atenção permite misturar as representações de textos e imagens. (b) Uma estratégia de treinamento consiste em otimizar a similaridade entre as representações. É aplicada uma camada de atenção cruzada seguida de uma camada linear (projecção). Uma função de loss mede a similaridade entre as representações. Com isso, os pesos da camada de projecção podem ser otimizados para aumentar a similaridade entre as representações de texto e imagem.

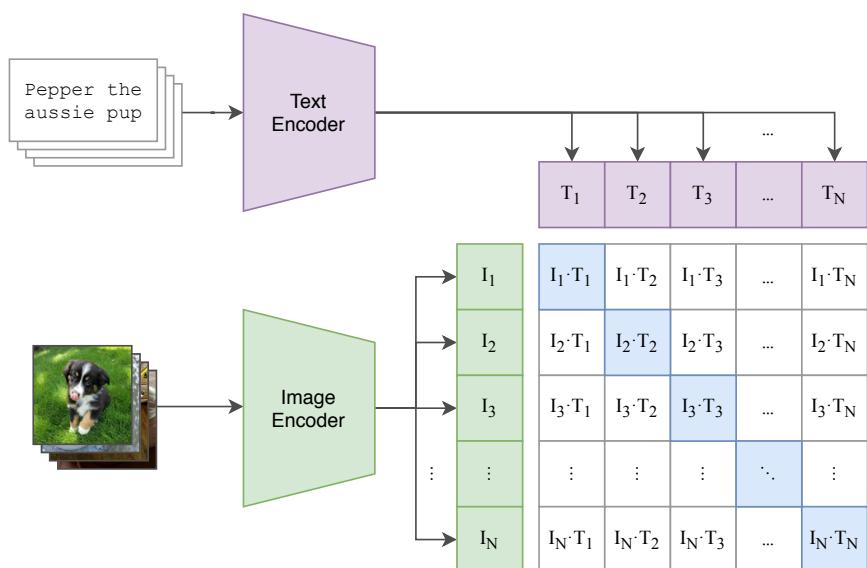


Figura 4.3: Ilustração do modelo CLIP. O modelo recebe como entrada um batch de imagens e respectivos textos descrevendo cada imagem. Um codificador de imagens (ResNet, ViT, etc) gera vetores de atributos visuais para as imagens e um codificador de texto (Transformer) gera vetores de atributos textuais. A similaridade de cosseno entre cada par de vetores é calculada. Uma função de loss apropriada recompensa valores altos na diagonal e valores baixos fora da diagonal, o que leva o modelo a gerar uma função de similaridade entre imagens e texto.

## A Símbolos

Escalar	Vetor
$x$	$\mathbf{x}$
$w$	$\mathbf{w}$
$f$	$\mathbf{f}$
$a$	$\mathbf{a}$
$r$	$\mathbf{r}$
$q$	$\mathbf{q}$
$p$	$\mathbf{p}$

Tabela A.1: Esta tabela mostra a tipografia dos caracteres usados para representar valores e vetores. Por exemplo, o caractere  $x$  é usado para representar um valor, enquanto que o caractere  $\mathbf{x}$  é usado para representar um vetor. Você pode consultar esta tabela caso tenha dúvidas no texto se alguma variável é escalar ou vetor.