

Deep Learning usando Tensorflow

Eduardo Montesuma
eduardomontesuma@alu.ufc.br



UNIVERSIDADE
FEDERAL DO CEARÁ

1 de dezembro de 2020

Summary

Introdução

Rede Neural Perceptron (uma camada)

Rede Neural Perceptron (várias camadas)

Prática I

Redes Neurais Convolucionais

Leitura Adicional

Prática II

Referências

Introdução

Organização do Minicurso

Objetivo: Aprendizado do uso do framework Tensorflow para aplicações de Machine Learning em processamento de imagens.

Divisão do conteúdo:

Quarta (02/12): Introdução, algoritmo Perceptron + Prática I,

Quinta (03/12): Redes Neurais Convolucionais + Prática II.

As aplicações serão feitas através da plataforma Google Colab.

O que é Machine Learning?

Definição: algoritmos de Machine Learning processam dados, aprendem à partir deles, e usam dados para tomar decisões.

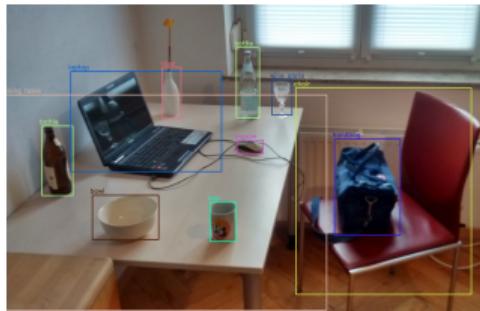


Figura 1: Exemplos de aplicações de aprendizagem de máquina. Da esquerda para a direita: reconhecimento de objetos, chatbots e carros autônomos.

O que é Machine Learning?

Este Minicurso irá focar em algoritmos de Machine Learning para **aprendizagem supervisionada**. Assumimos que possuímos os seguintes dados para cada indivíduo i num **conjunto de dados**,

- Um conjunto de características $\mathbf{x}_i = [x_{i1}, \dots, x_{id}]$. Essas características podem ser números reais (peso, idade, curvatura, etc.), ou valores categóricos.
- Um rótulo y_i , ou saída desejada. Quando nos referimos a um rótulo, assumimos que $y_i = \{1, \dots, K\}$ representa uma **classe**. Quando nos referimos à saída desejada, assumimos que y_i é um número real.

Ainda, fazemos a distinção: **classificação** (y_i representa uma classe) e **regressão** (y_i é uma saída desejada).

Machine Learning para Processamento de Imagens

Em particular, trataremos de dois problems:

1. **Classificação:** Dada uma imagem I , predizer seu rótulo y , ou seja, atribuir corretamente a que classe I pertence.



Figura 2: Classificação Muffin vs. Chihuahua.

Uma imagem I é uma matriz, com formato (h, w) , onde h é a altura, e w é a largura.

Representação Inteira

Cada elemento I_{ij} é um número inteiro de 8bits, portanto $I_{ij} = \{0, \dots, 255\}$.

Representação em Float

Cada elemento I_{ij} é um float (32 bits), portanto $I_{ij} \in [0, 1]$.

Conversão int \rightarrow float

$$I_{ij}^{float} = \frac{I^{int}}{255}.$$

Representação Computacional

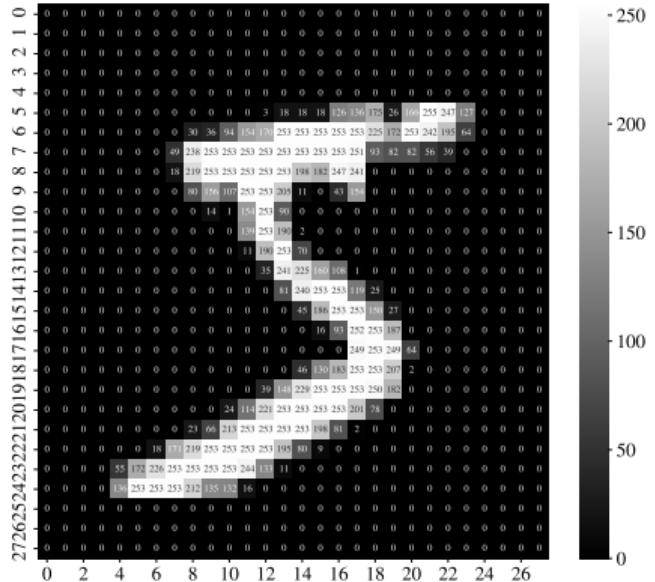


Figura 3: Amostra de um dígito 5 do conjunto de dados MNIST. Note que a faixa de valores $\{0, \dots, 255\}$ representa um degradê entre 0/preto e 255/branco.

Input Encoding

Uma matriz (h, w) pode ser entendida como uma coleção de w vetores-coluna. Esses vetores podem, por sua vez, serem concatenados ao longo da dimensão horizontal,

$$V = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow [\mathbf{a}_1 \quad \mathbf{a}_2 \quad \mathbf{a}_3] \rightarrow \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{32} \\ a_{33} \end{bmatrix} = v$$

onde $\mathbf{a}_j = [a_{1j}, a_{2j}, a_{3j}]$. O vetor final v tem dimensão $(h \times w, 1)$. Portanto, uma imagem pode ser representada como uma matriz $I \in \mathbb{R}^{h \times w}$, ou como um vetor $x \in \mathbb{R}^{hw}$.

Output Encoding

Um vetor de rótulos $y \in \{1, \dots, K\}^n$ pode ser entendido de várias formas. A mais intuitiva é a de que cada elemento $y_i \in \{1, \dots, K\}$ contém o rótulo do indivíduo i . No entanto, outra convenção será bastante útil.

One Hot Encoding: Suponha um problema que contenha $K = 5$ classes. A convenção "One-hot Encoding" corresponde à fazer a associação,

$$1 \rightarrow [1, 0, 0, 0, 0]$$

$$2 \rightarrow [0, 1, 0, 0, 0]$$

$$3 \rightarrow [0, 0, 1, 0, 0]$$

$$4 \rightarrow [0, 0, 0, 1, 0]$$

$$5 \rightarrow [0, 0, 0, 0, 1]$$

Note que, dessa forma, $\mathbf{y} \in \{0, 1\}^{n \times K}$,
onde cada \mathbf{y}_i é da forma,

$$\mathbf{y}_{ij} = \begin{cases} 1 & \text{se } y_i = j, \\ 0 & \text{caso contrário.} \end{cases}$$

O Google Colaboratory (Colab para os íntimos) é um serviço **cloud**, provido pelo Google. Ele oferece várias vantagens,

- Interface de uso amigável através de Jupyter Notebooks,
- Possibilidade de rodar códigos em computadores remotos (por um limite de até 6h–8h).
- **Possibilidade de rodar códigos em computadores remotos usando uma GPU.**



Datasets

Para as práticas, nós iremos explorar o conjunto de dados **MNIST**.



Figura 4: Exemplos de amostras do conjunto de dados MNIST. Cada amostra é tida como um indivíduo.

- Cada indivíduo possuí como características uma imagem do dataset,
- Cada imagem tem formato $(28, 28)$,
- Os indivíduos são representados como vetores de $28 \times 28 = 784$ dimensões

Álgebra Linear: trataremos sobretudo de transformações afins $T : \mathbb{R}^n \rightarrow \mathbb{R}^m$.
Uma transformação afim pode ser representada por uma matriz e um vetor,

$$T(x) = Ax + b$$

T transforma vetores $x \in \mathbb{R}^n$ em vetores $y = T(x) \in \mathbb{R}^m$. T é definida por $n \times (m + 1)$ parâmetros.

Tensores: em Machine Learning, a noção de tensor generaliza as noções de vetor e matriz. De maneira geral, um tensor é objeto d -dimensional,

Escalar:

v

Vetor:

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

Matriz:

$$\begin{bmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \vdots & \vdots \\ v_{m1} & \cdots & v_{mn} \end{bmatrix}$$

Tensor:



Na matemática, a noção de tensor é mais refinada. Para nós, é simplesmente um objeto com várias dimensões, ou seja, no espaço $\mathbf{R}^{n_1 \times \cdots \times n_d}$.

Derivadas: usualmente temos uma função $f : \mathbb{R} \rightarrow \mathbb{R}$, dependente de um parâmetro t , e calculamos sua derivada usando,

$$\frac{df}{dt}(t) = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

Nesse caso, f e t são escalares. Portanto, $\frac{df}{dt}$ é também um escalar.

Derivadas: nesse curso, iremos lidar com vetores, matrizes e funções destes objetos. Portanto, temos as seguintes noções,

Função/Variável	Escalar	Vetor	Matriz
Escalar	Escalar	Vetor	Matriz
Vetor	Vetor (gradiente)	Matriz (Jacobiana)	
Matriz	Matriz		

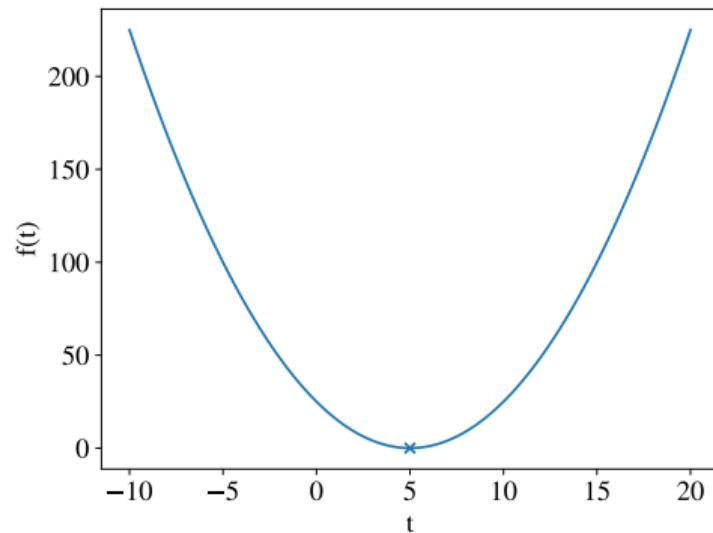
Tabela 1: Derivadas de escalares, vetores e matrizes.

Por que derivadas são importantes? Considere uma função real $f(t)$, a derivada $\frac{df}{dt}$ permite achar os pontos t em que f é máxima ou mínima,

Exemplo 1: Considere $f(t) = (t - 5)^2$.

Então $\frac{df}{dt} = 2(t - 5)$. A equação $\frac{df}{dt} = 0$ determina $t^* = 5$ como ponto crítico.

Note: $\frac{d^2f}{dt^2} = 2$, isso determina que o ponto crítico é um ponto mínimo.

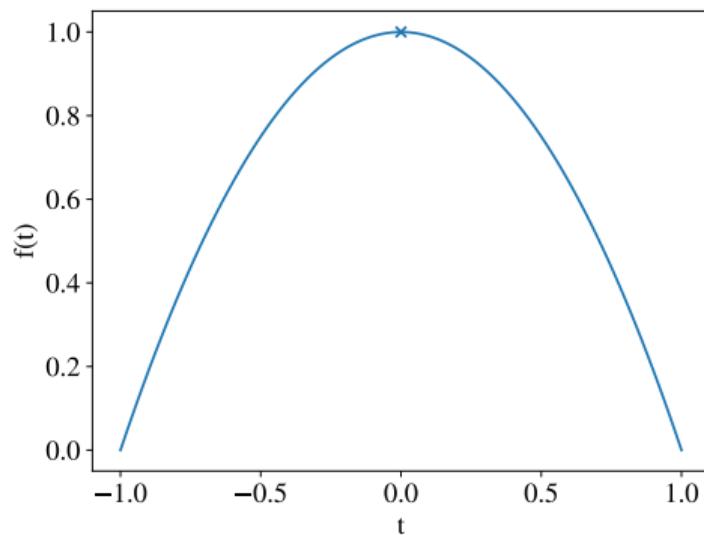


Por que derivadas são importantes? Considere uma função real $f(t)$, a derivada $\frac{df}{dt}$ permite achar os pontos t em que f é máxima ou mínima,

Exemplo 2: Considere $f(t) = 1 - t^2$.

Então $\frac{df}{dt} = -2t$. A equação $\frac{df}{dt} = 0$ determina $t^* = 0$ como ponto crítico.

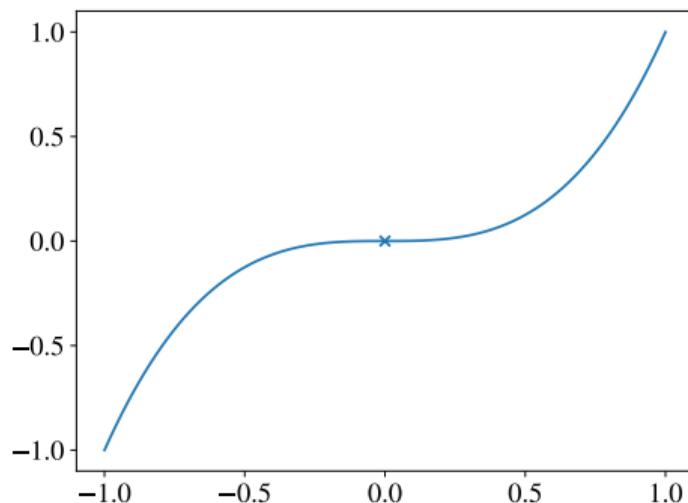
Note: $\frac{d^2f}{dt^2} = -2$, isso determina que o ponto crítico é um ponto mínimo.



Por que derivadas são importantes? Considere uma função real $f(t)$, a derivada $\frac{df}{dt}$ permite achar os pontos t em que f é máxima ou mínima,

Exemplo 3: Considere $f(t) = t^3$. Então $\frac{df}{dt} = 3t^2$. A equação $\frac{df}{dt} = 0$ determina $t^* = 0$ como ponto crítico.

Note: $\frac{d^2f}{dt^2}(t^*) = 6t^* = 0$, o que implica que o ponto crítico é um ponto de sela.



Noções de cálculo

Para funções de várias variáveis, $f(\mathbf{x}) = f(x_1, \dots, x_n) \in \mathbb{R}^n$, a noção de derivada se estende à cada variável:

$$\frac{\partial f}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n)}{h}$$

Podemos ainda agregar as derivadas num **vetor gradiente**,

$$\frac{\partial f}{\partial \mathbf{x}} = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right]$$

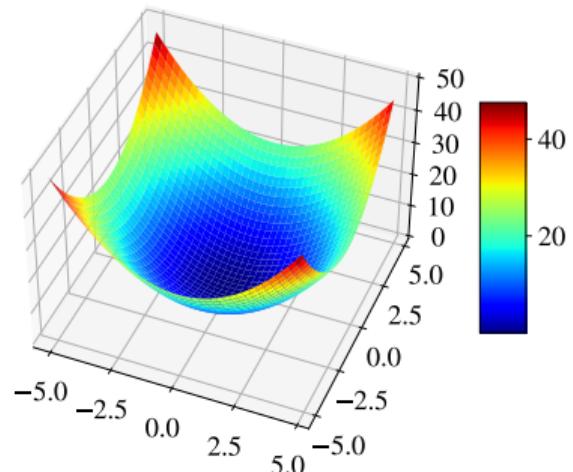


Figura 5: Gráfico da função $f(\mathbf{x}) = x_1^2 + x_2^2$, com mínimo em $x_1 = x_2 = 0$

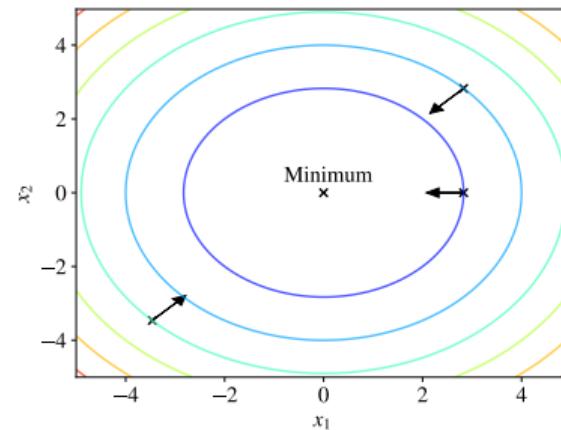
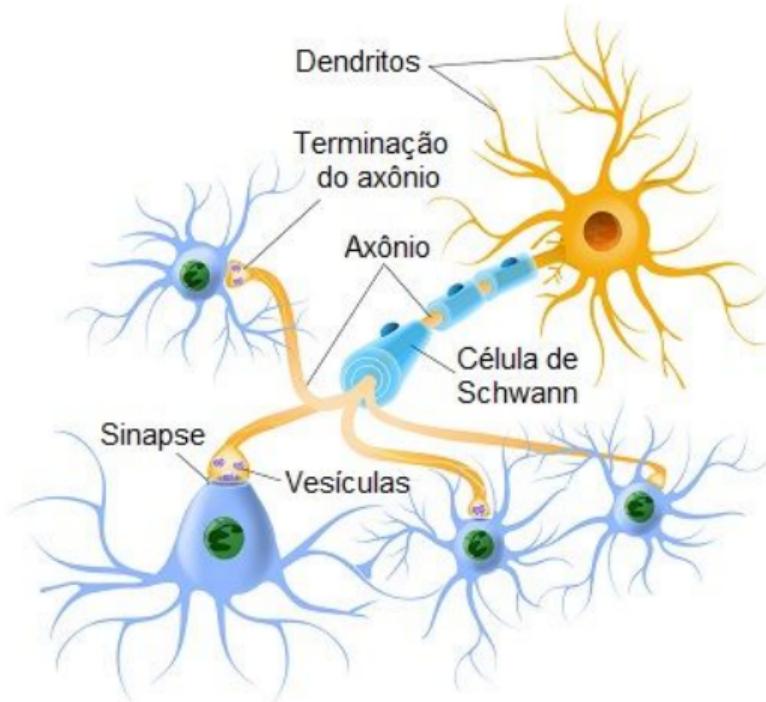


Figura 6: Contorno da função $f(\mathbf{x})$. Cada curva é definida por $f(\mathbf{x}) = k$. As flechas definem a direção dada pelo vetor gradiente.

Rede Neural Perceptron (uma camada)

Fisiologia de um neurônio



O neurônio é o bloco fundamental do sistema nervoso.

- Esta célula é formada por diferentes componentes,
 1. Uma estrutura receptora de sinais elétricos, chamada de **Dendrito**.
 2. Uma estrutura de processamento, chamada de **Corpo Celular**.
 3. Uma estrutura enviadora de sinais elétricos, chamada de **Axônio**.

Estrutura Sináptica

- O Fluxo de informação segue:
Neurônio Mensageiro → Dendritos → Corpo Celular → Axônio → Neurônio Receptor
- Este fluxo é denominado **sinapse**. Esta é a maneira pela qual neurônios comunicam-se entre si.
- Este fluxo funciona através do **disparo** das células neuronais. Um neurônio é dito ter disparado, quando **transmite sinal elétrico**, e, caso contrário, é dito estar em repouso.
- Em função dos equilíbrios iônicos presentes no neurônio, este tem um chamado **Potencial de Repouso**. O potencial de repouso é um valor de voltagem, o qual precisa ser vencido para que o neurônio dispare.

Potencial de Repouso

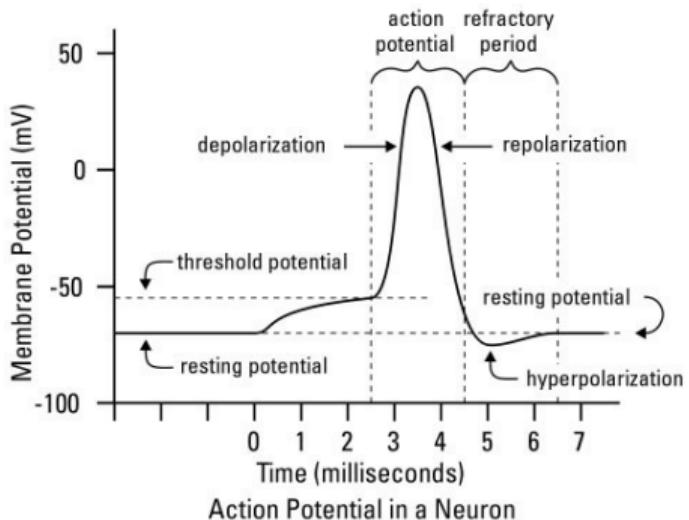


Figura 7: O potencial de repouso do neurônio é tido como 70mv. Abaixo disso, a célula não dispara, e acima disso (dentro de uma certa faixa), ele permite com que o sinal elétrico passe pela célula.

Um modelo matemático para o neurônio

Podemos modelar matematicamente o funcionamento de um neurônio com os seguintes conceitos,

1. Os sinais elétricos transmitidos pelos diversos neurônios serão denotados por x_1, \dots, x_n .
2. Cada sinal x_i é ponderado por um peso w_i .
3. A estrutura de processamento, chamada de **Corpo Celular**, responsável por **somar** os sinais de entrada ponderados pelos pesos, será representada matematicamente pela operação de somatório Σ .
4. O mecanismo de disparo do neurônio será dado por uma função φ , que determinará quando o neurônio é ativado.

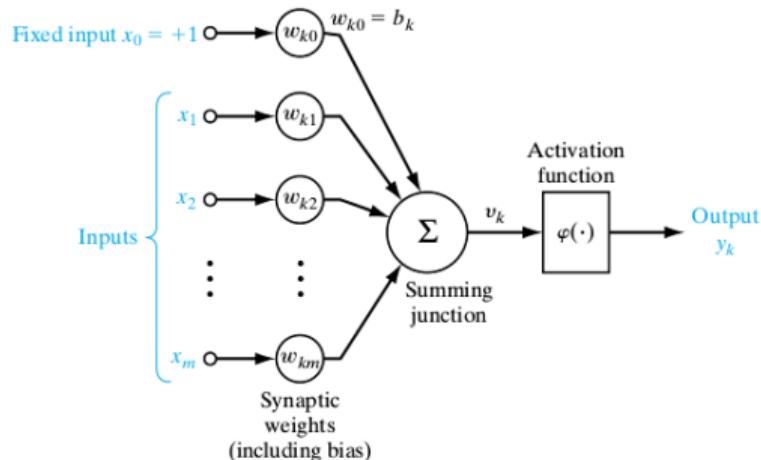


Figura 8: Modelo matemático de um neurônio. Figura extraída de [Haykin et al., 2009].

Rede Neural Perceptron

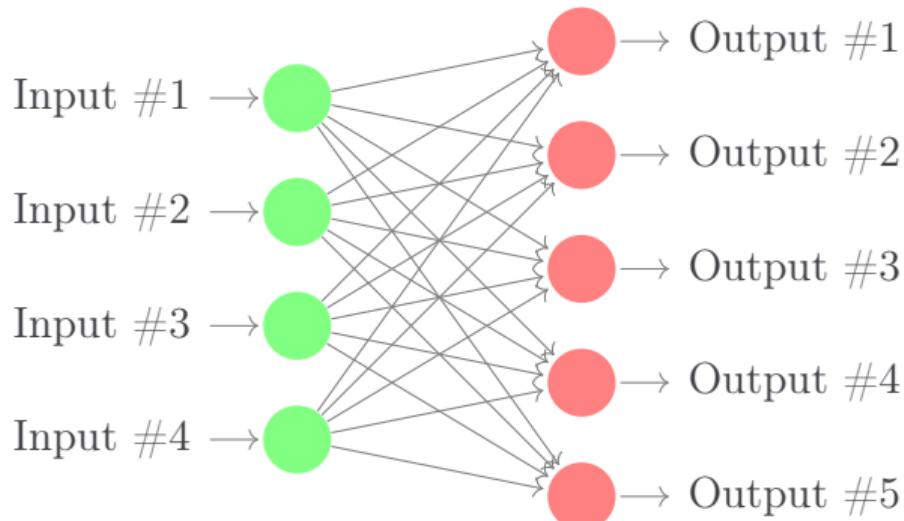


Figura 9: Perceptron com vários neurônios.

Funções de ativação

Existem várias funções de ativação:

- Sigmóide

$$\varphi(v) = \frac{1}{1 + \exp(-v)}$$

- Tangente Hiperbólica:

$$\varphi(\mathbf{v}) = \tanh(v)$$

Funções de ativação

Existem várias funções de ativação:

- Função ReLU:

$$\varphi(v) = \max(0, v)$$

- Softmax:

$$\varphi(\mathbf{v}) = \frac{\exp(\mathbf{v})}{\sum_{i=1}^n \exp(\mathbf{v}_j)}$$

Funções de ativação

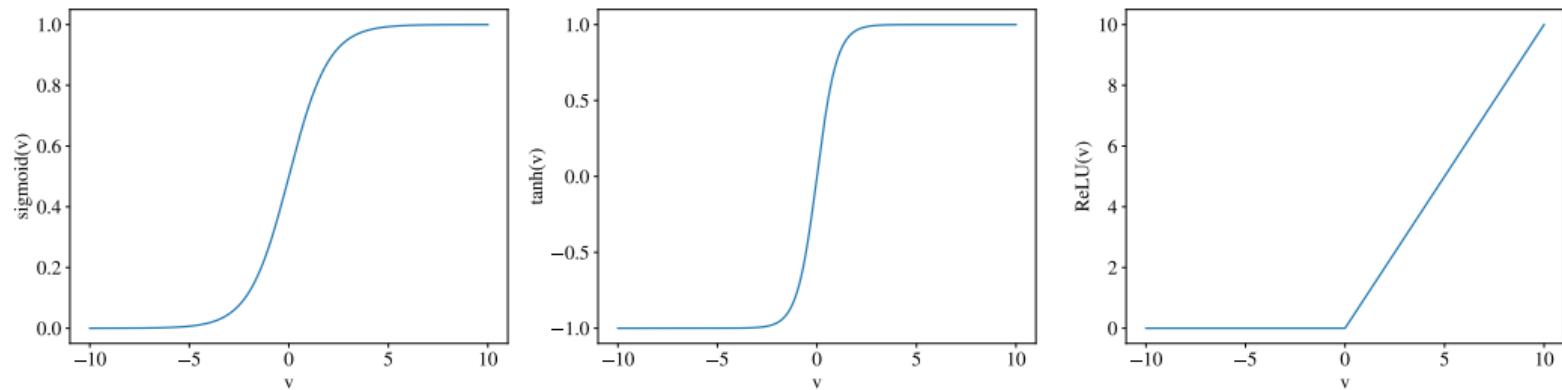


Figura 10: Comparação entre várias funções de ativação. Note que para valores muito grandes ou muito pequenos, *sigmoid* e *tanh* tem gradiente aproximadamente 0. Isso é conhecido como o fenômeno do desaparecimento de gradientes.

Funções de ativação

Comparação entre várias funções de ativação:

sigmoid:

- Saída no intervalo $[0, 1]$,
- Para redes profundas, há risco do fenômeno *vanishing gradients*,

tanh:

- Saída no intervalo $[-1, 1]$,
- Para redes profundas, há risco do fenômeno *vanishing gradients*,

ReLU:

- Saída no intervalo $[0, +\infty)$,
- Mais eficiente (e.g. implementação em hardware),
- Previne o fenômeno *vanishing gradient*.

Note: para essas três funções, quando v é um vetor, adotaremos a notação,

$$\varphi(\mathbf{v}) = [\varphi(v_1), \dots, \varphi(v_d)]^T$$

Uma atenção especial será dada à função de ativação softmax. Note que ela representa uma distribuição de probabilidade,

1. $0 \leq \text{softmax}(\mathbf{v})_j \leq 1,$
2. $\sum_{j=1}^d \text{softmax}(\mathbf{v})_j = \sum_{j=1}^d \frac{\exp(\mathbf{v}_j)}{\sum_{k=1}^d \exp(\mathbf{v}_k)} = 1.$

Portanto, essa função será usada sobretudo na saída, dando a probabilidade de \mathbf{x}_i ter classe y_i .

Perceptron

Forward-pass para um neurônio:

1. potencial $v_k = \sum_{i=1}^d \mathbf{W}_{ki}x_i + \mathbf{b}_k = \mathbf{W}_{k,:}^T \mathbf{x} + \mathbf{b}_k$
2. saída $\hat{y}_k = \varphi(\mathbf{W}_{k,:}^T \mathbf{x} + \mathbf{b}_k)$

Podemos agrupar 1. e 2. usando notação vetorial,

1. $\mathbf{v} = \mathbf{W}^T \mathbf{x} + \mathbf{b} = [v_1, \dots, v_K],$
2. $\hat{\mathbf{y}} = \varphi(\mathbf{v}) + \mathbf{b} = [\hat{y}_1, \dots, \hat{y}_K].$

No entanto, dado $\hat{\mathbf{y}}$ e o vetor de valores esperados \mathbf{y} , como calcular quão boa foi nossa predição?

Função de Custo

Definimos a noção de um custo para os nossos erros,

- Erro Quadrático:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 = \sum_{k=1}^K (\mathbf{y}_k - \hat{\mathbf{y}}_k)^2$$

- Entropia cruzada:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{k=1}^K -\mathbf{y}_k \log(\hat{\mathbf{y}}_k)$$

Otimização

Do exposto anteriormente, temos os seguintes conceitos:

- Parâmetros \mathbf{W} e \mathbf{b}
- Um mecanismo de predição, dado pelas equações, (1) $\mathbf{v} = \mathbf{Wx} + \mathbf{b}$ e (2) $\hat{\mathbf{y}} = \varphi(\mathbf{v})$.
- Uma função quantificando erros,

$$\begin{aligned}\mathcal{L}(\mathbf{W}, \mathbf{b}) &= \frac{1}{2} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2, \\ &= \frac{1}{2} \|\mathbf{y} - \varphi(\mathbf{Wx} + \mathbf{b})\|_2^2.\end{aligned}$$

Intuição: Minimizando \mathcal{L} com respeito aos parâmetros \mathbf{W} e \mathbf{b} , minimizamos a quantidade de erros que a rede neural comete.

Otimização

Porém, **como minimizar \mathcal{L} ?** Usando cálculo, uma condição necessária é:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = 0, \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = 0,$$

No entanto, as equações podem não ter solução analítica. **Alternativa:** minimização iterativa, usando **Gradient Descent**

$$\begin{aligned}\mathbf{W}^{\ell+1} &\leftarrow \mathbf{W}^\ell - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \\ \mathbf{b}^{\ell+1} &\leftarrow \mathbf{b}^\ell - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}}.\end{aligned}$$

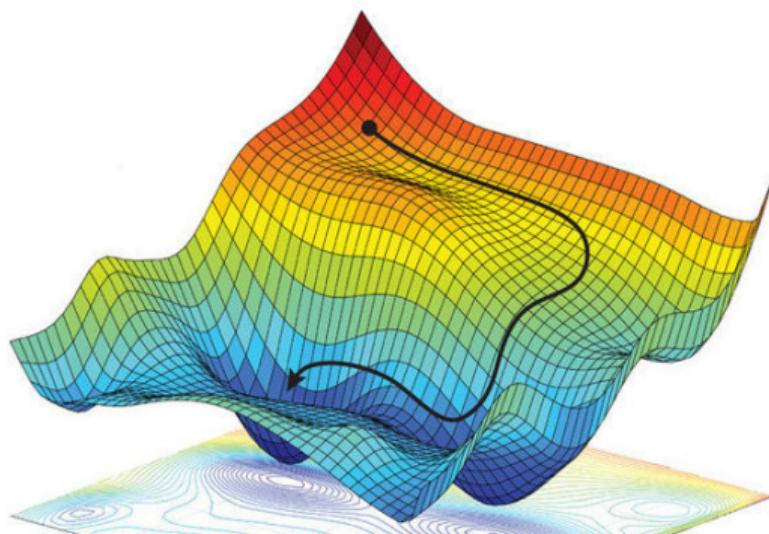


Figura 11: Ilustração do algoritmo Gradient Descent

Aprendizado

O aprendizado de uma rede neural se baseia na otimização da função de custo \mathcal{L} em relação aos seus parâmetros $\{\mathbf{W}_{ki}\}$ e $\{\mathbf{b}_k\}$. Note que **\mathcal{L} não depende explicitamente dos parâmetros mencionados**. Utilizamos portanto a regra da cadeia,

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ki}} &= \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_k} \times \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{v}_k} \times \frac{\partial \mathbf{v}_k}{\partial \mathbf{W}_{ki}}, \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}_k} &= \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_k} \times \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{v}_k} \times \frac{\partial \mathbf{v}_k}{\partial \mathbf{b}_k}.\end{aligned}$$

Os detalhes da derivação dessas derivadas são omitidos. Para os propósitos práticos do curso, não é necessário se aprofundar muito.

Aprendizado

Para $\mathcal{L} = \frac{1}{2} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$ e $\varphi(\mathbf{v}) = (1 + \exp(-\mathbf{v}))^{-1}$, temos os seguintes componentes,

- $\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = \hat{y}_k - y_k,$
- $\frac{\partial \hat{y}_k}{\partial v_k} = \varphi(v_k)(1 - \varphi(v_k)),$
- $\frac{\partial v_k}{\partial \mathbf{W}_{ki}} = x_i$

Portanto, $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ki}} = (\hat{y}_k - y_k)\varphi(v_k)(1 - \varphi(v_k))x_i,$

ou, usando notação vetorial, $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \left((\hat{\mathbf{y}} - \mathbf{y}) \otimes \varphi(\mathbf{v}) \otimes (1 - \varphi(\mathbf{v})) \right) \mathbf{x}^T$

Aprendizado

Para $\mathcal{L} = \frac{1}{2} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$ e $\varphi(\mathbf{v}) = (1 + \exp(-\mathbf{v}))^{-1}$, temos os seguintes componentes,

- $\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = \hat{y}_k - y_k,$
- $\frac{\partial \hat{y}_k}{\partial v_k} = \varphi(v_k)(1 - \varphi(v_k)),$
- $\frac{\partial v_k}{\partial \mathbf{W}_{ki}} = x_i$

Portanto, $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_k} = (\hat{y}_k - y_k)\varphi(v_k)(1 - \varphi(v_k)),$

ou, usando notação vetorial, $\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = (\hat{\mathbf{y}} - \mathbf{y}) \otimes \varphi(\mathbf{v}) \otimes (1 - \varphi(\mathbf{v}))$

Aprendizado: regularização

Algumas vezes, a modelagem pode sofrer com **overfitting**,

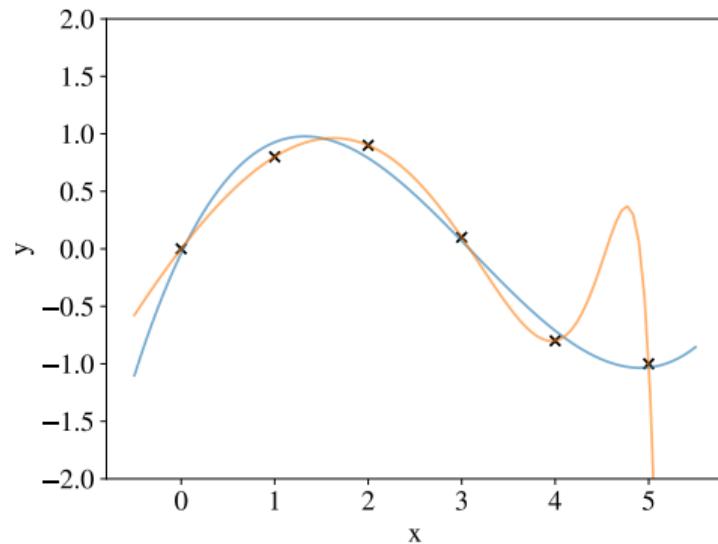


Figura 12: Exemplo de overfitting num modelo de regressão polinomial

Aprendizado: regularização

Uma maneira de evitar overfitting, é limitando o valor dos pesos da rede neural.
Isso corresponde à,

$$\begin{aligned}\mathcal{L}_{reg}(\mathbf{W}, \mathbf{b}) &= \frac{1}{2} \|\mathbf{y} - \varphi(\mathbf{Wx} + \mathbf{b})\|_2^2 + \lambda \Omega(\mathbf{W}), \\ &= \mathcal{L}(\mathbf{W}, \mathbf{b}) + \lambda \Omega(\mathbf{W})\end{aligned}$$

onde $\Omega(\mathbf{W})$ é uma função arbitrária. Uma escolha comum é a chamada **penalização ℓ^2** :

$$\Omega(\mathbf{W}) = \frac{1}{2} \|\mathbf{W}\|_2^2$$

Aprendizado: regularização

Note que Ω afeta o gradiente de \mathcal{L} com respeito à \mathbf{W} apenas,

$$\frac{\partial \mathcal{L}_{reg}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}} + \lambda \frac{\partial \Omega}{\partial \mathbf{W}},$$

Em particular, para a penalização ℓ^2 ,

$$\frac{\partial \Omega}{\partial \mathbf{W}} = \mathbf{W}$$

Aprendizado

Note o seguinte,

- Os gradientes dependem da escolha da função de custo \mathcal{L} e da função de ativação φ .
- Os parâmetros da rede neural são atualizados seguindo o algoritmo gradiente descendente,

$$\mathbf{W}_{ki}^{\ell+1} \leftarrow \mathbf{W}_{ki}^{\ell} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ki}}$$
$$\mathbf{b}_k^{\ell+1} \leftarrow \mathbf{b}_k^{\ell} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}_k}$$

Essa etapa de aprendizado é também chamada de **Backward Pass**.

Note que no exemplo anterior, usamos apenas uma amostra por iteração do algoritmo. Idealmente gostaríamos de usar **o dataset inteiro**, com N elementos:

$$\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{N \times d},$$

$$\mathbf{V} = [\mathbf{v}_1, \dots, \mathbf{v}_N] \in \mathbb{R}^{N \times K},$$

$$\hat{\mathbf{Y}} = [\varphi(\mathbf{v}_1), \dots, \varphi(\mathbf{v}_N)] \in \mathbb{R}^{N \times K}.$$

Aprendizado

Note que no exemplo anterior, usamos apenas uma amostra por iteração do algoritmo. Idealmente gostaríamos de usar **o dataset inteiro**, com N elementos:

$$L(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(\mathbf{Y}_i, \hat{\mathbf{Y}}_i),$$

Portanto, as derivadas são,

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}}{\partial \mathbf{W}}(\mathbf{Y}_i, \hat{\mathbf{Y}}_i),$$

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}}{\partial \mathbf{b}}(\mathbf{Y}_i, \hat{\mathbf{Y}}_i).$$

Batch vs. Stochastic Gradient Descent

Batch Gradient Descent

- Para cada amostra $\mathbf{X}_i, \mathbf{Y}_i$,
 - Calcule os gradientes $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}}$
 - Acumule os gradientes em variáveis $\Delta \mathbf{W}$,
 $\Delta \mathbf{b}$:

$$\Delta \mathbf{W} \leftarrow \Delta \mathbf{W} + \frac{1}{N} \frac{\partial \mathcal{L}}{\partial \mathbf{W}},$$
$$\Delta \mathbf{b} \leftarrow \Delta \mathbf{b} + \frac{1}{N} \frac{\partial \mathcal{L}}{\partial \mathbf{b}}$$

- Atualize os parâmetros:

$$\mathbf{W}^{\ell+1} \leftarrow \mathbf{W}^\ell - \eta \Delta \mathbf{W},$$
$$\mathbf{b}^{\ell+1} \leftarrow \mathbf{b}^\ell - \eta \Delta \mathbf{b}.$$

Stochastic Gradient Descent

- Para cada amostra $\mathbf{X}_i, \mathbf{Y}_i$,
 - Calcule os gradientes $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}}$
 - Atualize os parâmetros:
$$\mathbf{W}^{\ell+1} \leftarrow \mathbf{W}^\ell - \eta \Delta \mathbf{W},$$
$$\mathbf{b}^{\ell+1} \leftarrow \mathbf{b}^\ell - \eta \Delta \mathbf{b}.$$
- Embaralhe as amostras.

Processo repetido por um número definido de **épocas**.

Batch vs. Stochastic Gradient Descent

Batch Gradient Descent

- Maior acurácia no valor dos gradientes,
- Maior custo computacional.

Stochastic Gradient Descent

- Usar apenas uma amostra para calcular o gradiente é um caso extremo (Gradient Noise, significância estatística),
- Maior número de iterações, porém cada iteração é mais rápida.

Combinação das duas técnicas: Minibatch Gradient Descent.

Minibatch Stochastic Gradient Descent

Ao invés de utilizar apenas 1 exemplo por iteração, utilizaremos B amostras. B é chamado de batch size.

Minibatch Stochastic Gradient Descent

- Para cada batch de B amostras
 - Para cada amostra $\mathbf{X}_i, \mathbf{Y}_i$ no batch
 - Calcule os gradientes $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}}$
 - Acumule os gradientes em variáveis $\Delta \mathbf{W}, \Delta \mathbf{b}$:

$$\begin{aligned}\Delta \mathbf{W} &\leftarrow \Delta \mathbf{W} + \frac{1}{N} \frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \\ \Delta \mathbf{b} &\leftarrow \Delta \mathbf{b} + \frac{1}{N} \frac{\partial \mathcal{L}}{\partial \mathbf{b}}\end{aligned}$$

- Atualize os parâmetros:

$$\begin{aligned}\mathbf{W}^{\ell+1} &\leftarrow \mathbf{W}^\ell - \eta \Delta \mathbf{W}, \\ \mathbf{b}^{\ell+1} &\leftarrow \mathbf{b}^\ell - \eta \Delta \mathbf{b}.\end{aligned}$$

Note que $\mathbf{W}^{\ell+1}$ depende de \mathbf{W}^ℓ . Precisamos de um **ponto de partida**. Como inicializar os pesos?

- Inicialização com zeros: $W_{ij}^0 = 0$,
- Inicialização aleatória:

$$W_{ij}^0 \sim \mathcal{N}(0, \sigma^2),$$

$$W_{ij}^0 \sim \mathcal{U}(-\epsilon, +\epsilon).$$

- Inicialização de Glorot [Glorot and Bengio, 2010]: $\epsilon = \frac{1}{\sqrt{n}}$.

Rede Neural Perceptron (várias camadas)

Adicionando Camadas Ocultas

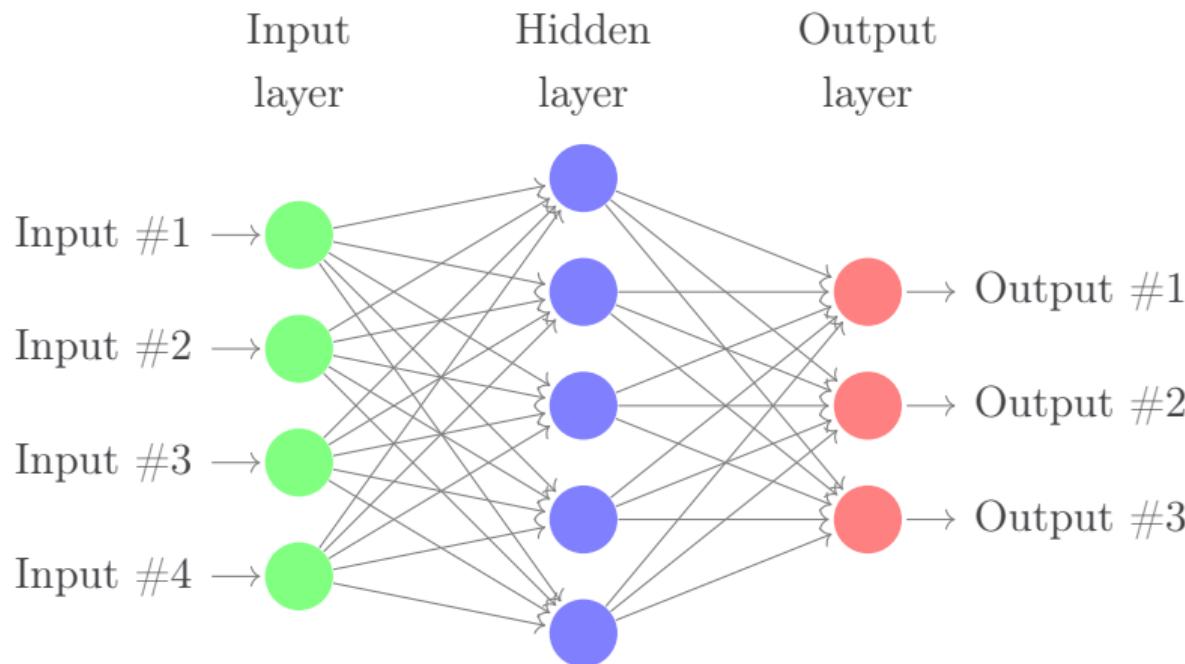


Figura 13: Rede Perceptron com mais de uma camada. Introduzimos uma camada oculta entre a camada de input, e a camada de output.

O forward pass é feito de maneira similar à quando temos uma camada apenas. Para cada camada ℓ , usaremos a seguinte notação:

1. **Input:** \mathbf{x}^ℓ ,
2. **Parâmetros:** $\mathbf{W}^\ell, \mathbf{b}^\ell$,
3. **Potencial:** $\mathbf{v}^\ell = \mathbf{W}^\ell \mathbf{x}^\ell + \mathbf{b}^\ell$
4. **Output:** $\hat{\mathbf{y}}^\ell = \varphi(\mathbf{v}^\ell)$

Além disso, $\mathbf{x}^{\ell+1} = \hat{\mathbf{y}}^\ell$, $\mathbf{x}^1 = \mathbf{x}$ e $\hat{\mathbf{y}}^2 = \hat{\mathbf{y}}$.

Exemplo: uma camada oculta

Forward pass:

- Camada oculta,

$$\mathbf{v}^1 = \mathbf{W}^1 \mathbf{x}^1 + \mathbf{b}^1,$$

$$\hat{\mathbf{y}}^1 = \varphi(\mathbf{v}^1),$$

- Camada de Saída,

$$\mathbf{v}^2 = \mathbf{W}^2 \mathbf{x}^2 + \mathbf{b}^2,$$

$$\hat{\mathbf{y}}^2 = \varphi(\mathbf{v}^2).$$

Exemplo: uma camada oculta

- Camada de Saída:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^2} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}^2} \times \frac{\partial \hat{\mathbf{y}}^2}{\partial \mathbf{v}^2} \times \frac{\partial \mathbf{v}^2}{\partial \mathbf{W}^2},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^2} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}^2} \times \frac{\partial \hat{\mathbf{y}}^2}{\partial \mathbf{v}^2} \times \frac{\partial \mathbf{v}^2}{\partial \mathbf{b}^2}.$$

- Camada Oculta:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}^2} \times \frac{\partial \hat{\mathbf{y}}^2}{\partial \mathbf{v}^2} \times \frac{\partial \mathbf{v}^2}{\partial \mathbf{x}^2} \times \frac{\partial \hat{\mathbf{y}}^1}{\partial \mathbf{v}^1} \times \frac{\partial \hat{\mathbf{v}}^1}{\partial \mathbf{W}^1},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^1} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}^2} \times \frac{\partial \hat{\mathbf{y}}^2}{\partial \mathbf{v}^2} \times \frac{\partial \mathbf{v}^2}{\partial \mathbf{x}^2} \times \frac{\partial \hat{\mathbf{y}}^1}{\partial \mathbf{v}^1} \times \frac{\partial \hat{\mathbf{v}}^1}{\partial \mathbf{b}^1}.$$

Exemplo uma camada oculta

Do exemplo anterior, vemos que os gradientes das camadas internas dependem dos gradientes das saídas,

- O cálculo dos gradientes é feito **recursivamente** no sentido inverso ao Forward pass. Essa fase é portanto chamada de **Backward pass**
- Existe um algoritmo recursivo para o cálculo automático dos gradientes numa rede neural arbitrária. Ele é chamado de Backpropagation Algorithm [Rumelhart et al., 1986]
- Este algoritmo é uma manifestação de uma teoria mais geral chamada de **diferenciação automática** [Baydin et al., 2017].

Prática I



TensorFlow

Tensorflow é uma biblioteca da linguagem Python, com foco em Machine Learning. Objetivos:

- Facilitar a construção de modelos de redes neurais,
- Robustez de modelos de Machine Learning,
- Experimentação para P&D.

Tensorflow – Introdução

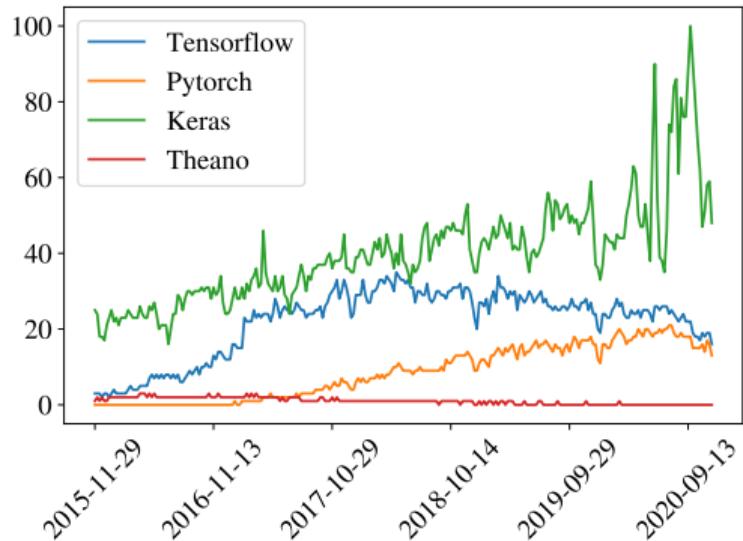


Figura 14: Linha do tempo de buscas no Google. Tensorflow é a linguagem mais usada. Desde 2017 Tensorflow tem usado Keras como uma API de alto-nível.

Aspectos Técnicos:

- A biblioteca Tensorflow implementa algoritmos de diferenciação automática em C++,
- As chamadas em Python são feitas através de uma interface Python - C++,
- Portanto, não é necessário que o programador saiba C++ (apenas Python).

Paradigma de programação: podemos classificar a biblioteca Tensorflow como declarativa. O foco do programa é em especificar **o que** as operações devem fazer ao invés de **como** elas farão o resultado.

Tensorflow – Elementos

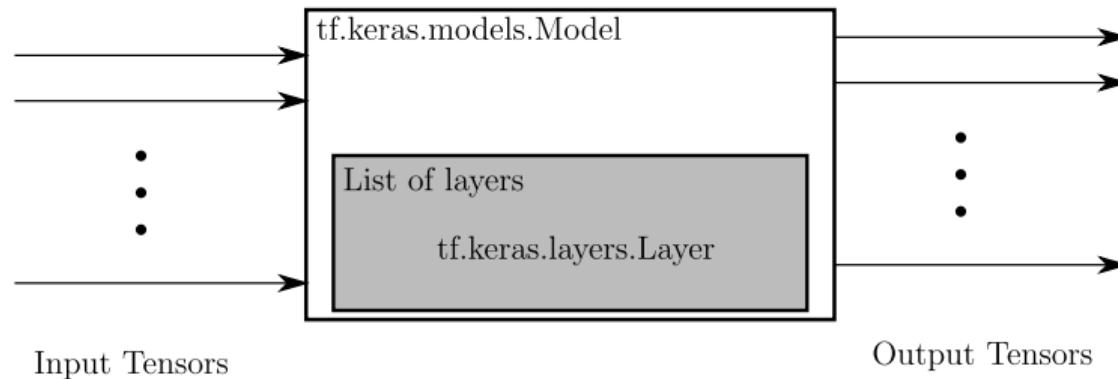


Figura 15: Esquema de elementos que constituem um modelo em Tensorflow.

Tensorflow – Sintaxe

Para o Perceptron com uma camada:

- Camada de Entrada:

```
1 x = tf.keras.layers.Input(shape=(784,))
```

- Camada de Saída:

```
1 y = tf.keras.layers.Dense(n_units=10, activation='sigmoid')(x)
```

- Definição do modelo:

```
1 y = tf.keras.models.Model(inputs=x, outputs=y)
```

Essas três linhas de código sintetizam a rede perceptron. Elas criam um **grafo computacional**, escondido atrás da abstração de modelo.

Tensorflow – Compilação

Até o presente momento, apenas o **grafo computacional** foi gerado. Suas variáveis, no entanto, não foram inicializadas. Para utilizar o modelo criado é necessário **compilá-lo**. Para compilar um modelo, precisamos:

1. Definir o algoritmo de otimização. **OBS:** vimos anteriormente o algoritmo **Gradient Descent**, portanto utilizaremos a implementação chamada de Stochastic Gradient Descent (SGD). Note que lr representa η em nossas equações,

```
1 optimizer = tf.keras.optimizers.SGD(lr=1e-1)
```

2. Definir a função de custo. **OBS:** nossas derivações utilizaram o **erro quadrático**, portanto utilizaremos o **Mean Squared Error**,

```
1 loss_fn = tf.keras.losses.MeanSquaredError()
```

Tensorflow – Compilação

Podemos finalmente compilar o modelo,

```
1 model.compile(  
2     loss=loss_fn ,  
3     optimizer=optimizer ,  
4     metrics=[ 'accuracy' ]  
5 )
```

A compilação de um modelo faz o seguinte:

1. inicializa todas as variáveis do modelo,
2. insere a função de custo no grafo computacional,
3. define as regras para o cálculo dos gradientes,
4. define as regras para a atualização dos parâmetros.

Treinamento

Treinar um modelo usando tensorflow resume-se à chamada do método `.fit()`,

```
1 Model.fit(  
2     x=Xtr,  
3     y=Ytr,  
4     batch_size=batch_size,  
5     epochs=epochs,  
6     validation_data=(Xts, Yts),  
7     shuffle=True  
8 )
```

Onde (Xtr, Ytr) são os dados (imagens + rótulos) de treinamento, (Xts, Yts) são os dados de teste, $batch_size$ é o número de amostras visto à cada iteração do algoritmo de otimização e $epochs$ é o número de iterações completas pelo conjunto de dados de treino.

Redes Neurais Convolucionais

Recapitulação – Encontro anterior

No encontro anterior, estudamos a rede neural Perceptron,

- Vimos que ela pode ser usada de maneira satisfatória para classificar dígitos,
- Através do uso de várias camadas, conseguimos um resultado interpretável com acurácia máxima de 89%

Uma característica interessante foi notada:

- O aprendizado do Perceptron se dá através da criação de protótipos dos dígitos em seus pesos. O reconhecimento é feito através da similaridade entre amostra e protótipo.
- Apesar de não haver overfitting, o uso de regularização **melhora a qualidade dos protótipos aprendidos**.

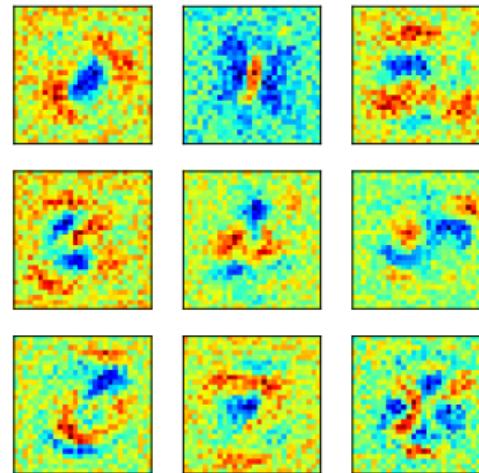


Figura 16: Matrizes de pesos aprendidas para a rede Perceptron simples.

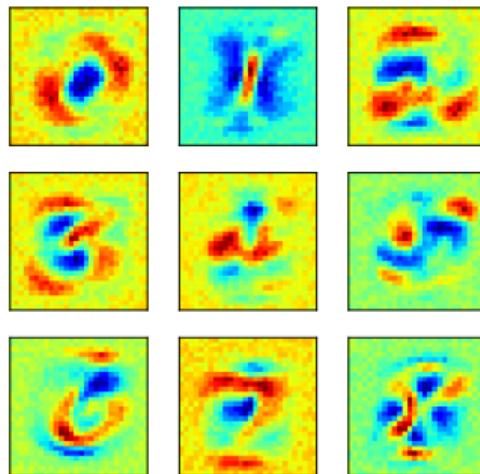


Figura 17: Matrizes de pesos aprendidas para a rede Perceptron simples com uso de regularização ℓ^2 . Note que os pesos são menos ruidosos.

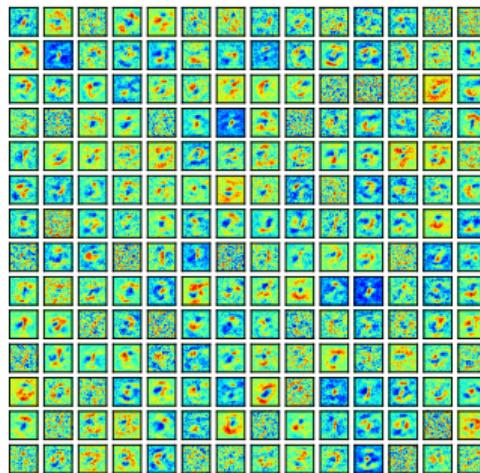


Figura 18: Matrizes de pesos aprendidas para a rede Perceptron multicamada com uso de regularização ℓ^2 . Note que os protótipos estão mais divididos entre os neurônios.

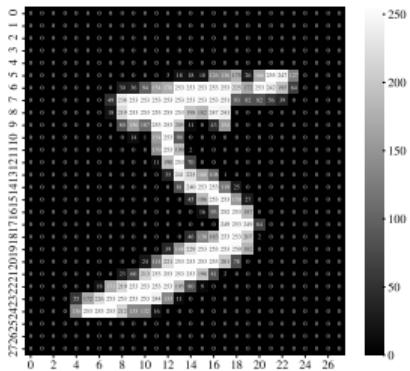
Tipos de camadas

Ao utilizarmos imagens como entradas para redes neurais, temos vários tipos novos de camadas,

- **Camada Densa:** chamada assim pois todos os elementos de entrada estão conectados à todos os elementos de saída. São equivalentes às camadas de um perceptron. **Parâmetros:** $\mathbf{W} \in \mathbb{R}^{n_{in} \times n_{out}}$ e $\mathbf{b} \in \mathbb{R}^{n_{out}}$.
- **Camada Convolucional:** baseada na operação de convolução. **Parâmetros:** n filtros de formato $\mathbf{W} \in \mathbb{R}^{k \times k}$ e $\mathbf{b} \in \mathbb{R}^n$.
- **Camada Agregativa:** similar à convolução, mas aplica uma função de agregação (média, máximo, mínimo, etc).

Motivação: Imagens

Ao invés de representar uma imagem como um vetor, iremos representá-la como uma matriz,



Vantagens:

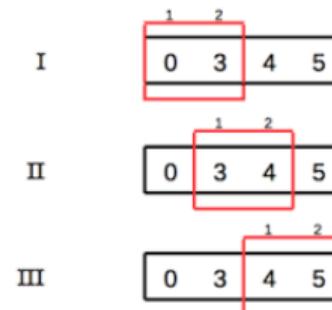
- Representação espacial de imagens (duas coordenadas, x e y),
- Melhor aproveitamento das características das imagens (e.g. redundância, simetria),
- Nova operação: convolução 2D.

Motivação: Convolução

Convolução 1D Operação entre dois vetores:

$$s_i = (x \star w)_i = \sum_{k=-n}^n x_{i-k} w_k$$

- x é um sinal (representado por um vetor),
- w é chamado de **kernel** ou **filtro**.



Convolução 1D Operação entre duas matrizes:

$$S_{i,j} = (I \star K)_{i,j} = \sum_m \sum_n I_{i-m,j-n} K_{m,n}$$

- I é uma imagem (representada por uma matriz),
- K é chamado de **kernel** ou **filtro**.

Motivação: Convolução

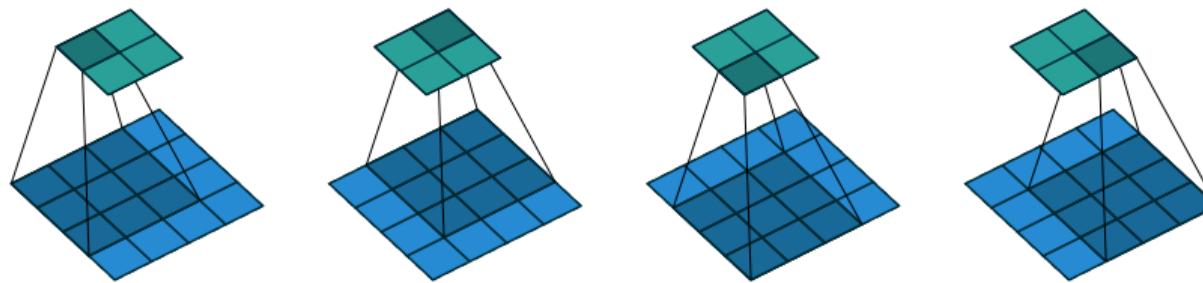


Figura 19: Convolução padrão. Visualização retirada de [Dumoulin and Visin, 2016].

Problema: formato da matriz de entrada (4×4) é diferente do formato da matriz de saída (2×2). Na maioria das vezes, queremos que a operação **conserve o formato da entrada**.

Motivação: Convolução

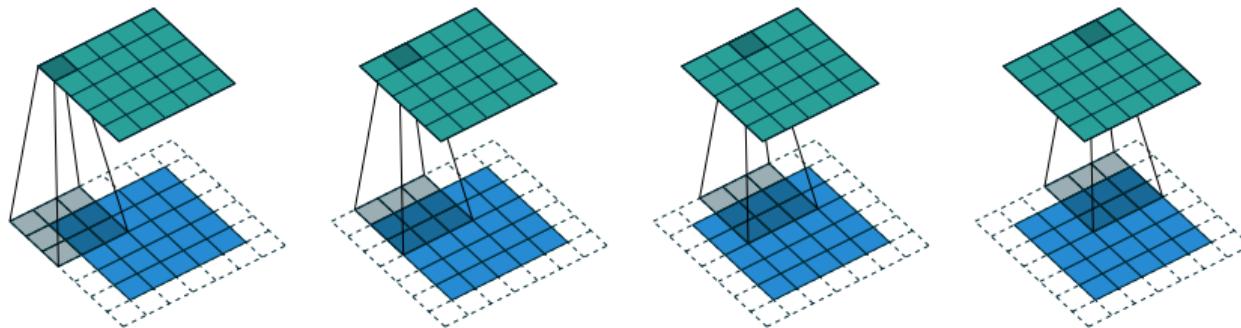


Figura 20: Convolução com zero-padding. Visualização retirada de [Dumoulin and Visin, 2016].

Solução: adicionar uma borda de zeros ao redor da imagem de entrada.

Motivação: conectividade esparsa

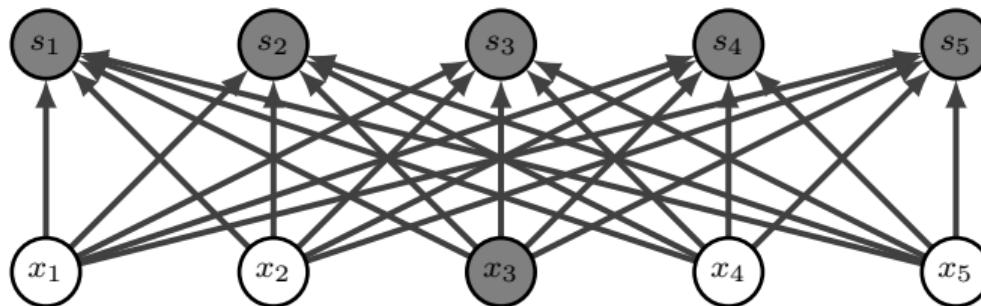


Figura 21: Dependência entre variáveis, camada densa. Figura retirada de [Goodfellow et al., 2016]

Motivação: conectividade esparsa

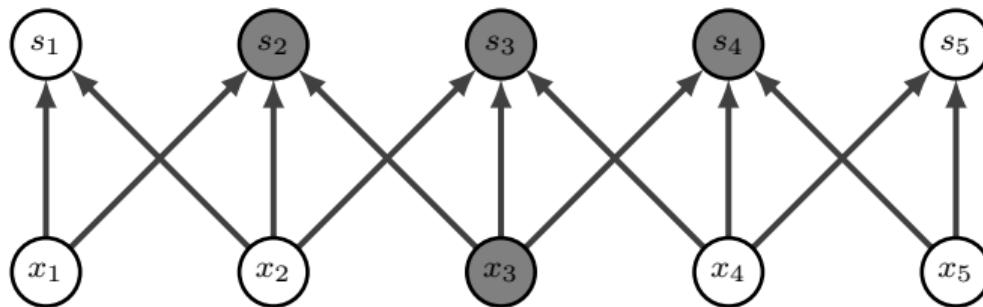


Figura 22: Dependência entre variáveis, camada convolucional. Figura retirada de [Goodfellow et al., 2016]

Motivação: partilha de parâmetros

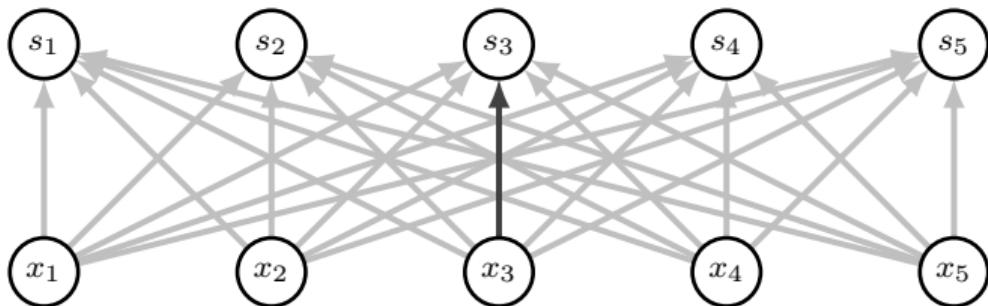


Figura 23: Camada densa. As flechas pretas representam um parâmetro usado por várias variáveis. Figura retirada de [Goodfellow et al., 2016]

Motivação: partilha de parâmetros

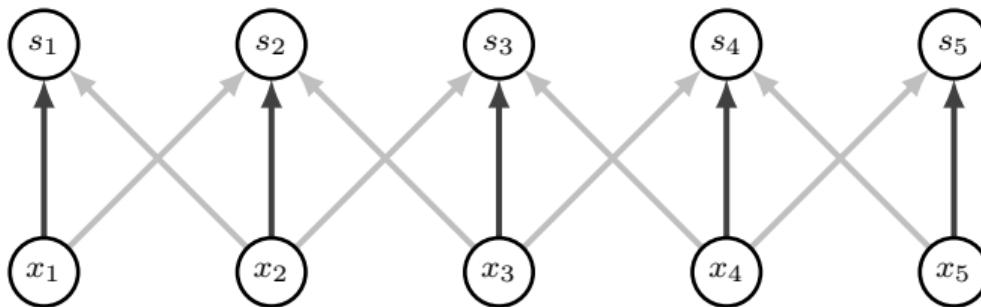


Figura 24: Camada convolucional. As flechas pretas representam um parâmetro usado por várias variáveis. Figura retirada de [Goodfellow et al., 2016]

Motivação: eficiência nas operações

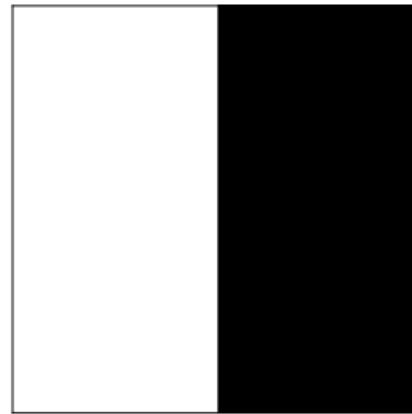


Figura 25: Da esquerda para a direita: imagem original, imagem filtrada, filtro utilizado. Número de parâmetros no filtro: 4. Número de parâmetros com uma transformação linear: 489 bilhões (!!!).

Camadas de agregação: intuição

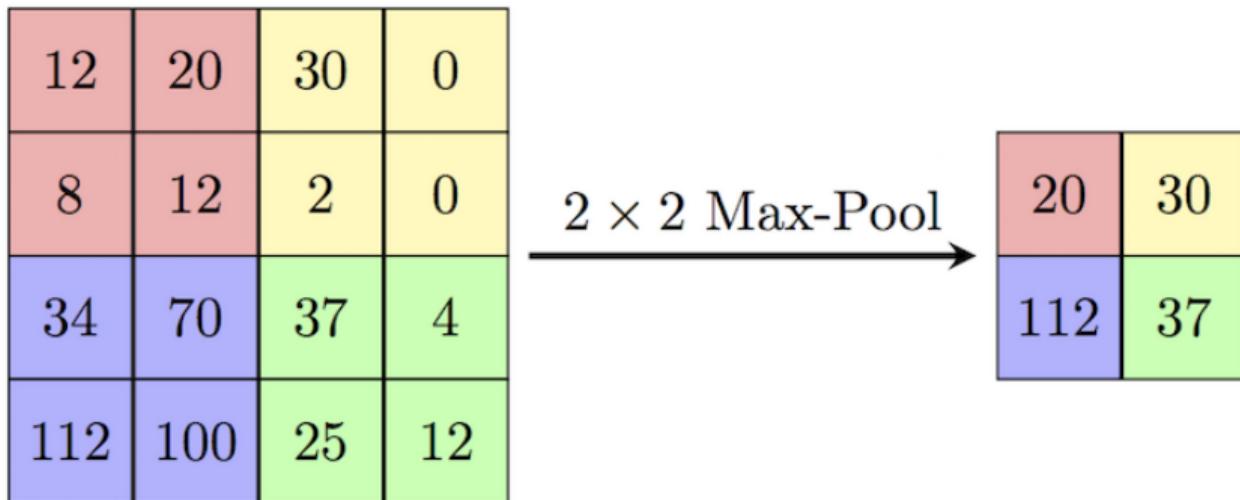


Figura 26: Exemplo de uma camada de agregação usando a operação max.

Camadas de agregação: intuição

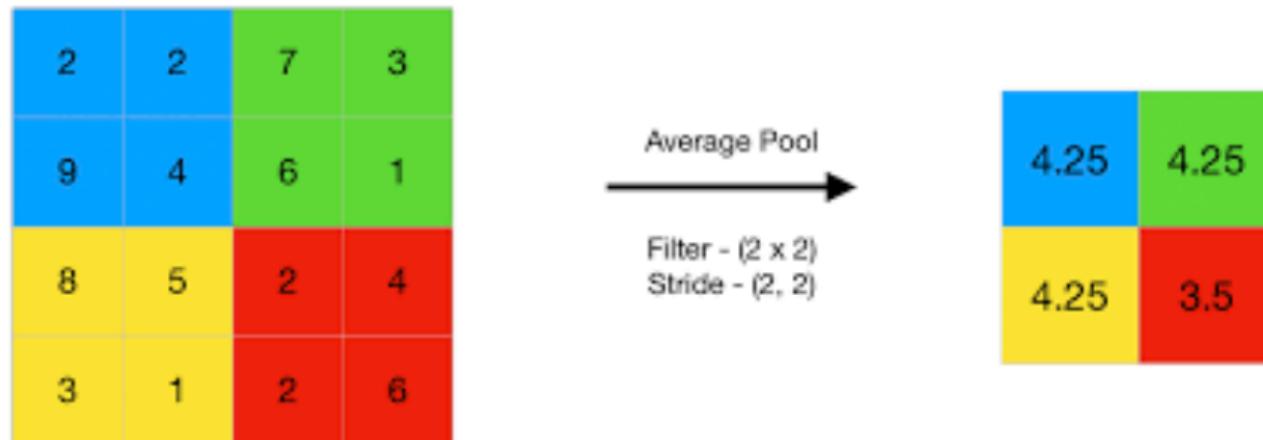


Figura 27: Exemplo de uma camada de agregação usando médias.

Camada de agregação: vantagem

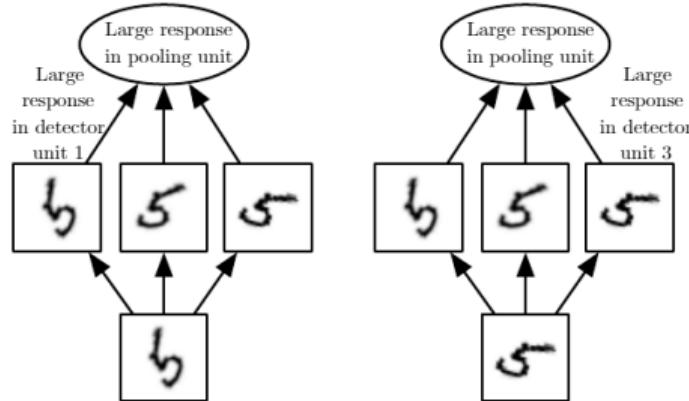


Figura 28: Vantagem do uso de camadas de agregação (pooling): aprendizado de invariâncias.

Aqui vamos discutir algumas técnicas do estado da arte que exigem um pouco mais de discussão:

- Função de ativação ReLU,
- Função de ativação Softmax na saída

Por que usar ReLU?

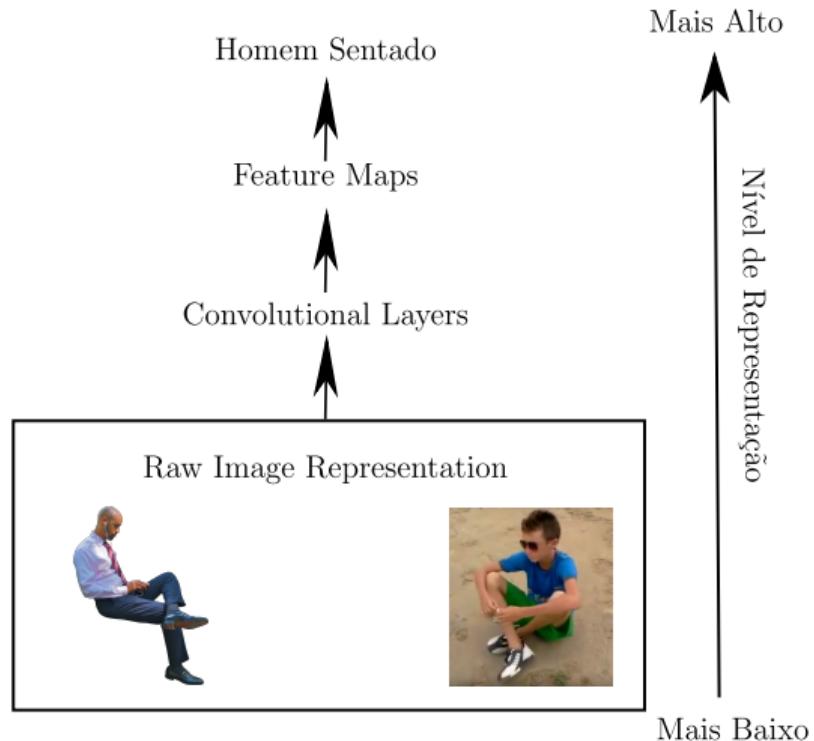
- Neurônios armazenam informação de forma esparsa e distribuída [Attwell and Laughlin, 2001],
- Compatibilidade funcional [Glorot et al., 2011],
- Maior rapidez de convergência [Krizhevsky et al., 2017],
- **bis:** maior velocidade na implementação em Hardware (sigmóide e tanh necessitam de uma expansão em série de taylor).

Esparsidade: Note que para pesos inicializados aleatoriamente ao redor de 0, cerca de 50% das ativações na camada oculta serão 0.

Por que se importar com esparsidade?

- Fundamentação biológica,
- Economia de memória,
- **Quebra do emaranhamento de representações/fatores de variação [Bengio, 2009]**

Técnicas avançadas



Por que usar Softmax na saída?

- Ao usarmos a função Softmax, a saída se torna uma distribuição de probabilidade
- Portanto, podemos ver não só a predição final (classe mais provável), mas o grau de certeza de sua predição.

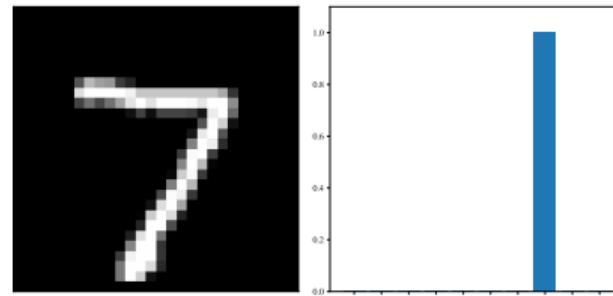


Figura 29: Comparaçõa entre o grau de certeza da predição de uma rede neural convolucional à respeito de um dígiro 7.

Leitura Adicional

Visualização das ativações

Existem várias técnicas de visualização de dados com muitas dimensões. Uma das mais populares é chamada de **Stochastic Neighbor Embeddings** (SNE) [Hinton and Roweis, 2002], ele é baseado na seguinte mecânica,

- Constrói-se uma distribuição de probabilidade nos dados brutos,

$$p_{ij} = \frac{\exp(-d_{ij}^2)}{\sum_{k \neq i} \exp(-d_{ik}^2)},$$

onde $d_{ij}^2 = \frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma_i^2}$ é uma função de distâncias.

- Constrói-se um vetor correspondente em baixa dimensão,

$$q_{ij} = \frac{\exp(-||\mathbf{y}_i - \mathbf{y}_j||^2)}{\sum_{k \neq i} \exp(-||\mathbf{y}_i - \mathbf{y}_j||^2)},$$

Visualização das ativações

Como $P = [p_{ij}]_{i,j}$ e $Q = [q_{ij}]_{i,j}$ são distribuições de probabilidade, definimos as posições \mathbf{y}_i em baixa dimensão através da minimização da função de custo,

$$\mathcal{L}(P, Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

A otimização de \mathcal{L} pode ser feita à partir do algoritmo Stochastic Gradient Descent, por exemplo.



Figura 30: Dada a relativa generalidade dos filtros de camadas convolucionais, levanta-se a pergunta: **podemos reutilizar redes neurais em outros datasets?**. Na imagem, filtros da rede neural Alexnet [Krizhevsky et al., 2017].

Aprendizagem por Transferência

A resposta é **sim!** [Yosinski et al., 2014],

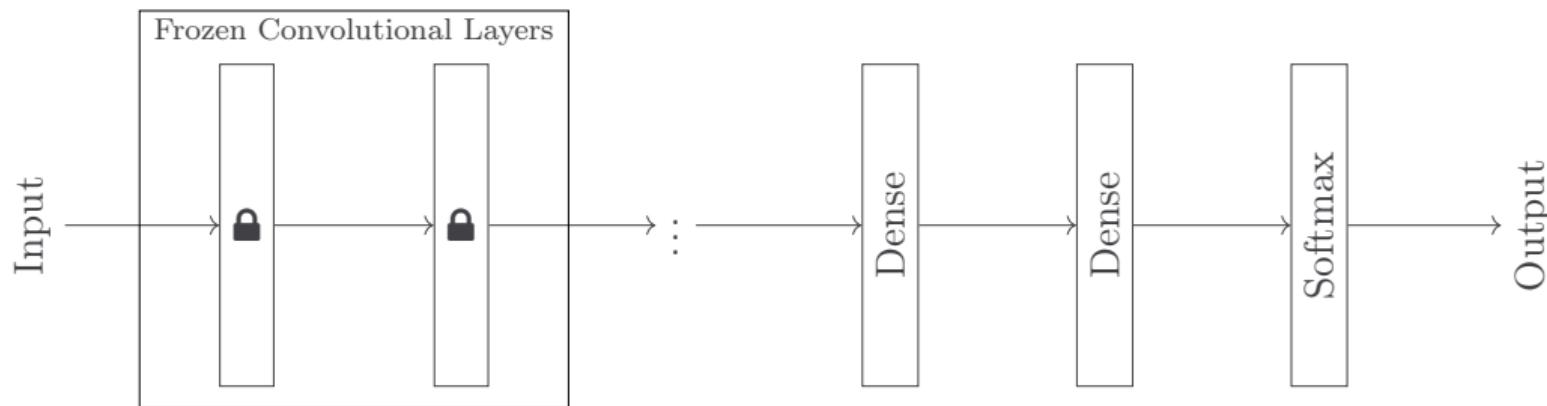


Figura 31: Exemplo da reutilização de camadas de uma rede neural. Camadas com o símbolo indicam que os parâmetros são fixados, de maneira que somente as demais são treinadas. Figura adaptada de [Yosinski et al., 2014].

A nova corrida espacial: quem tem a rede mais profunda?

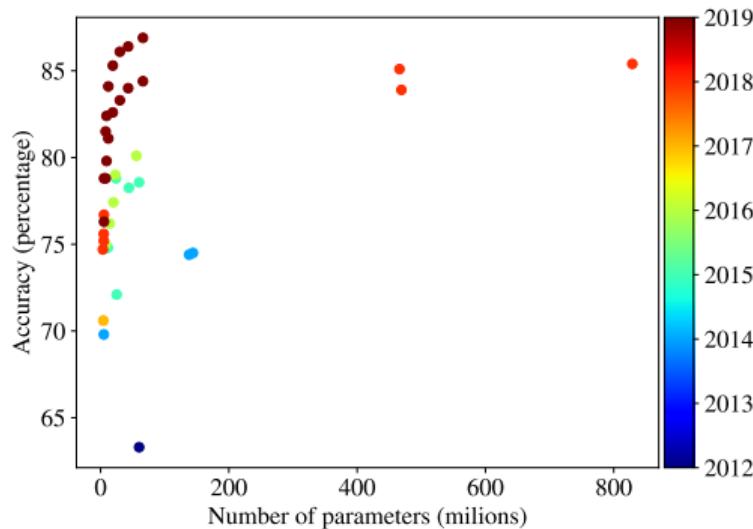


Figura 32: Análise número de parâmetros vs. taxa de acerto para o dataset imangenet.
Fonte: Paperswithcode

Redes Neurais Profundas implementadas em Hardware

Note que a parte complicada das redes neurais é a parte de treinamento. Portanto, hoje em dia as principais aplicações utilizam a seguinte filosofia:

- Treinamento da rede neural em Python (usando Tensorflow, Pytorch, Theano, etc.),
- Uma vez que o modelo é treinado, seus pesos são salvos num arquivo *.h5*,
- com os pesos salvos, é possível utilizá-lo para inferência.

Particularmente, é necessário apenas construir o pipeline de inferência (camada-à-camada) e implementar as operações das camadas densa e convolucional.

Prática II

Referências

Referências i

Attwell, D. and Laughlin, S. B. (2001).

An energy budget for signaling in the grey matter of the brain.

Journal of Cerebral Blood Flow & Metabolism, 21(10):1133–1145.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2017).

Automatic differentiation in machine learning: a survey.

The Journal of Machine Learning Research, 18(1):5595–5637.

Bengio, Y. (2009).

Learning deep architectures for AI.

Now Publishers Inc.

Dumoulin, V. and Visin, F. (2016).

A guide to convolution arithmetic for deep learning.

arXiv preprint arXiv:1603.07285.

Glorot, X. and Bengio, Y. (2010).

Understanding the difficulty of training deep feedforward neural networks.

In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011).

Deep sparse rectifier neural networks.

In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

Referências ii

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016).

Deep learning, volume 1.

MIT press Cambridge.

Haykin, S. S. et al. (2009).

Neural networks and learning machines.

Hinton, G. E. and Roweis, S. (2002).

Stochastic neighbor embedding.

Advances in neural information processing systems, 15:857–864.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017).

Imagenet classification with deep convolutional neural networks.

Communications of the ACM, 60(6):84–90.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986).

Learning representations by back-propagating errors.

nature, 323(6088):533–536.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014).

How transferable are features in deep neural networks?

In *Advances in neural information processing systems*, pages 3320–3328.