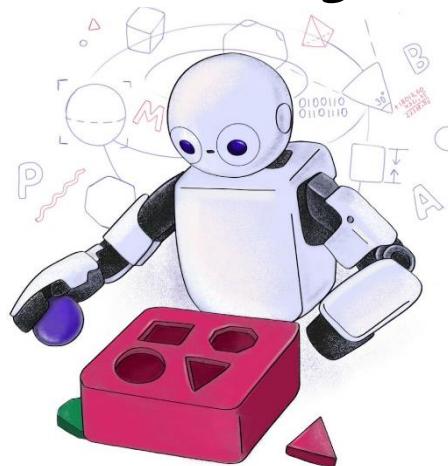


T320 - Introdução ao Aprendizado de Máquina II: *Redes Neurais Artificiais (Parte IV)*



Inatel

Felipe Augusto Pereira de Figueiredo
felipe.figueiredo@inatel.br

Recapitulando

- No último tópico, discutimos como as redes neurais aprendem.
- Vimos que isso é feito através da minimização de uma função de erro (também chamada de função de custo).
 - Usamos o ***erro quadrático médio*** por questões didáticas, mas existem várias outras funções como por exemplo a ***entropia cruzada***, usada para o treinamento de classificadores e a ***focal loss*** para o treinamento de detectores de objetos.
- Aprendemos que a minimização da função de erro é realizada de forma iterativa usando o algoritmo da retropropagação do erro para calcular os vetores gradiente.
- Analisamos como a retropropagação funciona através de um exemplo.
- Neste tópico, iremos discutir algumas questões práticas para o treinamento de redes neurais.

Algumas questões práticas sobre algoritmos de aprendizado

- Podemos dizer que os *elementos básicos do aprendizado de máquina* através de *redes neurais* foram apresentados até aqui.
- Porém, existem alguns aspectos práticos que nós precisamos discutir.
- Portanto, começamos relembrando sobre a questão do *cálculo do vetor gradiente*.

Cálculo do vetor gradiente

- Conforme vimos anteriormente, a base para o aprendizado de redes MLP é a obtenção do ***vetor gradiente*** e o estabelecimento de um ***processo iterativo de busca*** dos ***pesos*** que minimizem a ***função de erro***.
- Vimos que a obtenção do ***vetor gradiente*** se dá através do processo de ***retropropagação do erro***, o qual é dividido em duas etapas:
 - Etapa direta (***forward***) onde se apresenta um exemplo de entrada, x , e obtém-se a resposta da rede e, consequentemente, o ***erro de saída***.
 - Etapa reversa (***retropropagação***) em que se calculam as derivadas parciais necessárias ao longo das camadas da rede.

Cálculo do vetor gradiente

- Vimos que a derivada parcial do erro em relação a um peso qualquer é a média de **gradientes particulares (ou locais)**

$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^m} = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \boxed{\frac{\partial e_j^2(n)}{\partial w_{i,j}^m}} = \frac{1}{N_{\text{dados}}} \sum_{n=1}^{N_{\text{dados}}} \nabla J_n(\mathbf{W}).$$

Gradiente local

- O **gradiente local** é a derivada parcial do erro da j -ésima saída da rede para o n -ésimo exemplo de entrada em relação ao peso $w_{i,j}^m$.
- $\nabla J_n(\mathbf{W})$ é a média dos N_M **gradientes locais** para o n -ésimo exemplo de entrada.
- No entanto, aqui surge um questionamento importante:
 - O que é melhor, usar a **média dos N_M gradientes locais**, $\nabla J_n(\mathbf{W})$, e já dar um passo de otimização, ou seja, atualizar os pesos, **reunir o gradiente completo e então dar um passo único e mais preciso** ou **um meio termo**?

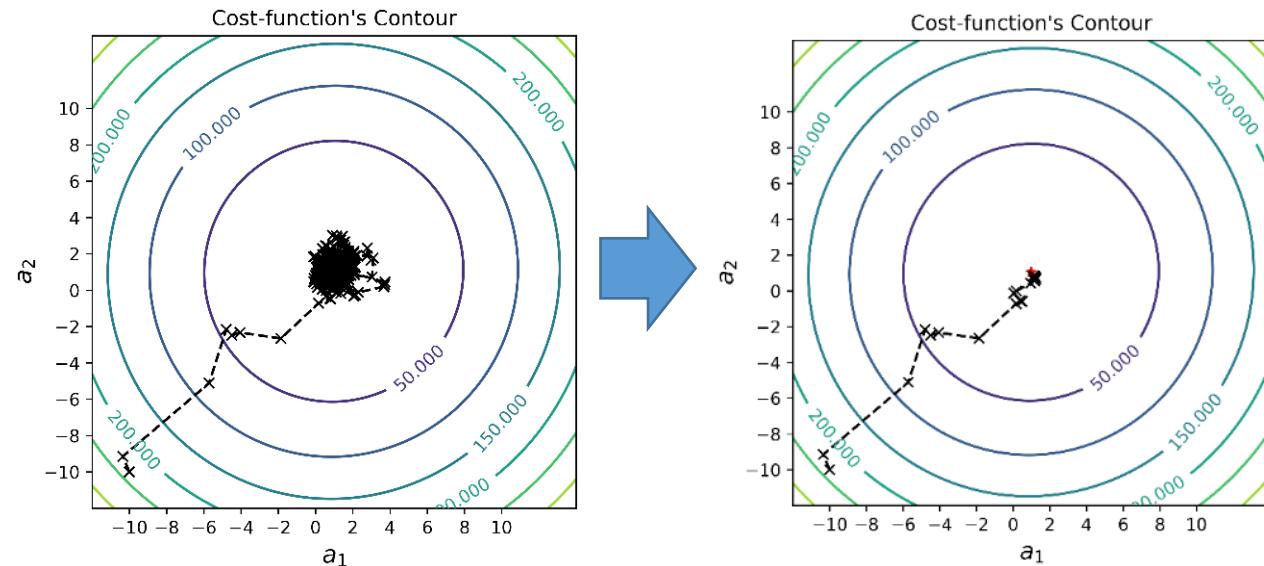
Cálculo do vetor gradiente

- Esse questionamento gera três abordagens possíveis para o cálculo do vetor gradiente.
 - O cálculo usando todos os exemplos (batelada).
 - O cálculo (i.e., estimativa) usando um único exemplo (estocástica).
 - O cálculo usando um subconjunto de exemplos (*mini-batches*).
- Nas *redes neurais profundas* (ou *deep learning*), usadas com muita frequência em problemas possuem enormes conjuntos de dados, usa-se a abordagem com *mini-batches*, pois com ela, podemos controlar a complexidade computacional necessária para o treinamento.
- **OBS.:** Os exemplos para estimativa do vetor gradiente com as versões *estocástica* e *mini-batch* devem ser *aleatoriamente* escolhidos a partir do conjunto de treinamento.

Variações dos algoritmos de otimização dos pesos

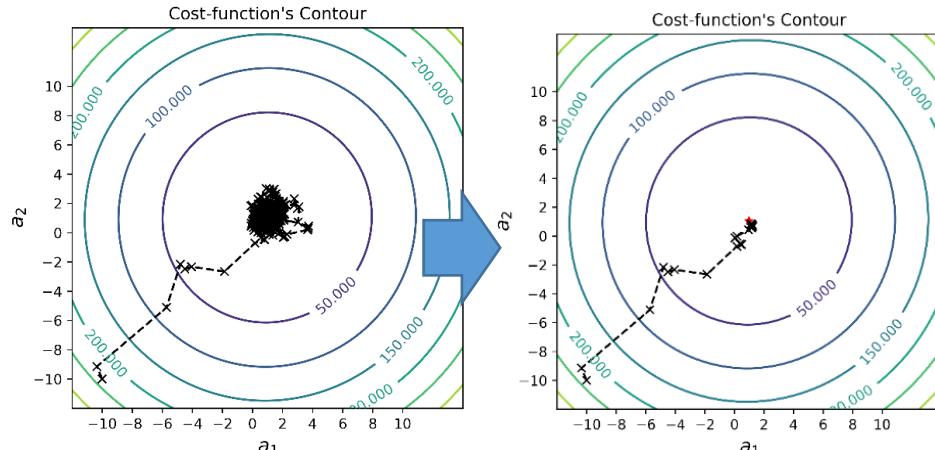
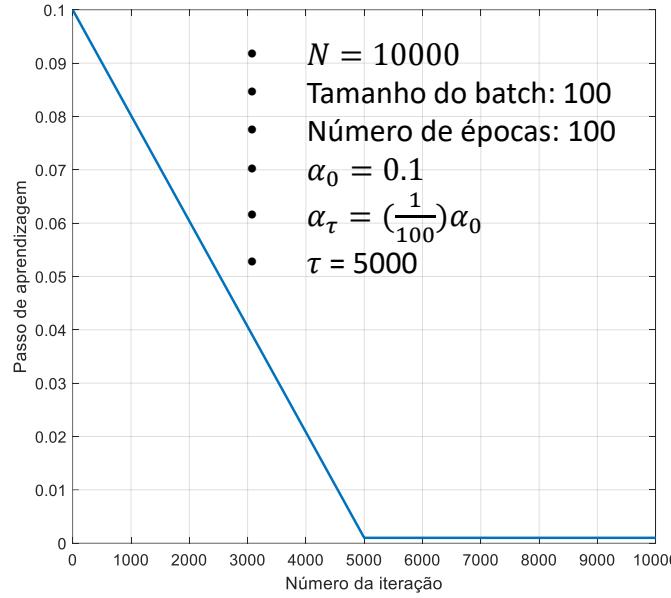
- Existem algumas *modificações* que podem ser aplicadas às versões estocásticas (mini-batch e estocástica) para *melhorar seu desempenho sem aumentar muito sua complexidade computacional.*
- As modificações mais usadas são:
 - Redução gradual do passo de aprendizagem,
 - Adição do termo momentum,
 - Adição do termo momentum de Nesterov,
 - Adição de passos de aprendizagem adaptativos.

Redução gradual do passo de aprendizagem



- Assim como fizemos com as versões estocásticas do gradiente descendente quando trabalhamos com regressores lineares, podemos **reduzir o passo de aprendizagem para tornar essas versões mais comportadas e, esperançosamente, obter a convergência.**
- Podemos utilizar todas as técnicas que aprendemos antes: **redução por degraus, decaimento exponencial ou temporal.**

Redução gradual do passo de aprendizagem



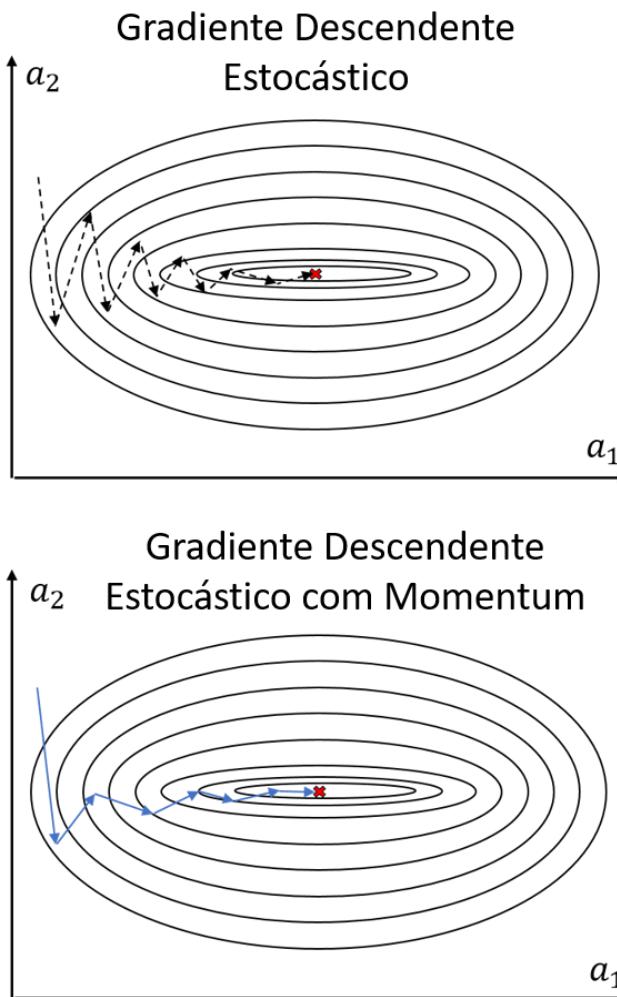
- As figuras mostram o resultado do uso da técnica de redução temporal com a equação

$$\alpha_j = \left(1 - \frac{j}{\tau}\right) \alpha_0 + \frac{j}{\tau} \alpha_\tau,$$

onde j é o contador de iterações, α_0 é o valor inicial do passo, τ é o número da iteração a partir da qual o passo fica constante e α_τ é o valor constante do passo após a τ -ésima iteração.

- Entretanto, percebam que *ainda temos que encontrar os valores ideais para os hiperparâmetros*, nesse caso, α_0 , α_τ e τ .

Termo momentum

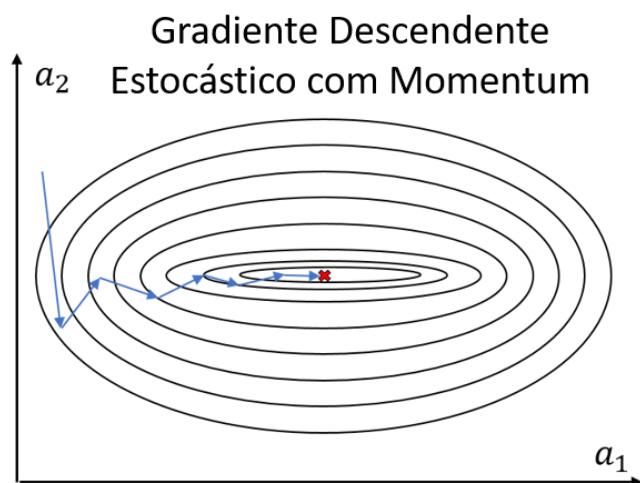
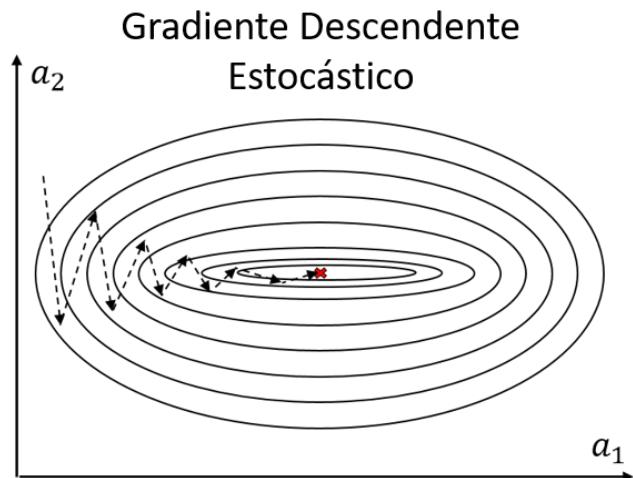


- Como vimos antes, o termo momentum adiciona uma **média móvel de estimativas do vetor gradiente**, ν , à equação de atualização dos pesos, **tornando as atualizações menos ruidosas** e, consequentemente, **acelerando a convergência e aumentando a estabilidade** do algoritmo.

$$\begin{aligned}\nu(i) &= \mu\nu(i - 1) + (1 - \mu)\nabla\hat{J}_e(\mathbf{w}(i)), \\ \mathbf{w}(i + 1) &= \mathbf{w}(i) - \alpha\nu(i).\end{aligned}$$

onde $\nabla\hat{J}_e(\mathbf{w}(i))$ é a **estimativa do vetor gradiente** e $\mu \in [0,1)$ (**coeficiente de momentum**) determina a quantidade de estimativas anteriores que são consideradas no cálculo da média.

Termo momentum



- O termo momentum adiciona uma média das estimativas dos gradientes anteriores à atualização corrente.
 - Quando as **estimativas apontam na mesma direção** por várias iterações, o termo faz com que o tamanho dos passos dados naquela direção aumentem, ou seja, o **modelo ganha impulso**.
 - Quando as **estimativas mudam de direção** a cada nova iteração, o termo **suaviza as variações**.
 - Como resultado, temos **convergência mais rápida e oscilação reduzida**.
- A **desvantagem** é que nós precisamos encontrar as valores ideais dos **hiperparâmetros** α e μ .

Momento de Nesterov

- O método do **momento de Nesterov** é uma variação do **termo momentum** em que o cálculo da **estimativa do vetor gradiente** não é feito em relação ao vetor de pesos atual, $w(i)$, mas em **relação ao próximo vetor de pesos**, ou seja, em relação ao valor do vetor de pesos após sua atualização com o termo momentum,

$$w(i + 1) = w(i) - \alpha v(i).$$

- Essa mudança no cálculo da estimativa do vetor gradiente faz com que o **momento de Nesterov** apresente **convergência mais rápida e ajustes mais precisos dos pesos** do que o termo momentum, especialmente em regiões onde a **superfície de erro se assemelha à forma de um vale**.

Passo de aprendizagem adaptativo

- Na *variação adaptativa*, o passo de aprendizagem é *ajustado adaptativamente* de acordo com a *inclinação da superfície de erro*.
- Além disso, usa *passos de aprendizagem diferentes para cada peso* do modelo, *os atualizando de forma independente* de acordo com a inclinação da superfície na direção dos pesos.
- Assim, esses métodos são adequados para redes neurais, onde a *superfície de erro é bastante irregular e diferente em diferentes dimensões, tornando a atualização dos pesos mais efetiva*.
- Uma *vantagem* é que na maioria dos casos, *não é necessário se ajustar manualmente nenhum hiperparâmetro*.
- As técnicas mais conhecidas são RMSProp, AdaGrad e Adam.

Inicialização dos pesos

- Um outro aspecto prático que é importante discutirmos é a *inicialização dos pesos de uma rede neural*.
- Como os métodos de treinamento de *redes neurais* são de *busca local*, eles dependem da *inicialização dos pesos*.
- Porém, a inicialização pode afetar drasticamente a qualidade da solução obtida.
- O *ponto de inicialização dos pesos* pode afetar a velocidade de convergência do algoritmo.
- Alguns *pontos de inicialização* fazem com que a rede alcance uma *boa solução mais rapidamente*, enquanto outros pontos podem levar a uma *convergência mais lenta* (e.g., algoritmo pode ser inicializado em um ponto de sela ou em uma região de platô).

Inicialização dos pesos

- Alguns ***pontos de inicialização*** são tão instáveis que o algoritmo pode encontrar dificuldades numéricas (***underflow*** e ***overflow***), falhando completamente em convergir (***desaparecimento*** ou ***explosão*** dos gradientes).
- Uma questão importante da inicialização dos pesos é ***quebrar a simetria*** entre os ***nós***, ou seja, ***nós*** com a ***mesma função de ativação*** e ***conectados aos mesmos nós***, devem ter pesos iniciais diferentes, caso contrário, eles terão os mesmos pesos ao longo do treinamento (i.e., aprendem a mesma coisa).
- Portanto, como veremos a seguir, para ***quebrar a simetria e evitar problemas de convergência***, utilizamos algumas ***heurísticas de inicialização aleatória dos pesos***.

Inicialização dos pesos

- Os pesos iniciais são tipicamente obtidos a partir de *distribuições gaussianas ou uniformes*, não importando muito qual delas é usada.
- No entanto, a *escala de variação da distribuição de inicialização dos pesos* tem um efeito significativo no *resultado da otimização* e, consequentemente, na *capacidade de generalização* da rede neural.
- Sendo assim, a *escala de variação* da inicialização dos pesos levanta algumas discussões.
- Distribuições com *grande escala variação* tendem a *reduzir o problema da simetria*, pois a probabilidade de valores iniciais bastante distintos é maior.

Inicialização dos pesos

- Porém, se as *magnitudes dos valores iniciais forem muito grandes*, podemos ter problemas de *instabilidade*.
- Pesos com magnitudes muito grandes podem levar os *nós* com *funções de ativação* do tipo
 - Sigmoide a operarem na região de saturação, causando o *desaparecimento do gradiente*.
 - ReLU à *explosão do gradiente*.
- Por outro lado, distribuições com *escala de variação muito pequena* têm *maiores chances causar a simetria entre nós* e também podem apresentar *instabilidade ou lentidão* durante o treinamento.
 - Por exemplo, redes com *pesos muito pequenos e com nós usando função de ativação ReLU*, podem ter problemas com o *desaparecimento do gradiente*.
- Na sequência veremos algumas *heurísticas* para inicialização dos pesos.

Heurísticas de inicialização dos pesos

- A ideia por trás destas heurísticas de inicialização dos pesos é **manter a média das ativações dos nós igual a zero e suas variâncias constantes ao longo das várias camadas da rede**, pois desta forma evita-se o desaparecimento ou a explosão do gradiente.
- Considerando uma camada com m entradas e n saídas, temos as seguintes **heurísticas** para inicializar os **pesos sinápticos*** de seus nós.

Inicialização	Funções de ativação	Distribuição Uniforme $U(-r, r)$	Distribuição Normal $N(0, \sigma^2)$
Xavier/Glorot	Linear (i.e., nenhuma), Tanh, Logística, Softmax	$r = \sqrt{\frac{6}{m+n}}$	$\sigma^2 = \frac{2}{m+n}$
He	ReLU e suas variantes	$r = \sqrt{\frac{6}{m}}$	$\sigma^2 = \frac{2}{m}$
LeCun	SELU	$r = \sqrt{\frac{3}{m}}$	$\sigma^2 = \frac{1}{m}$

*Em geral, inicializa-se os **pesos de bias** com **valores iguais a 0**, pois se mostra uma inicialização bastante eficiente na maioria dos casos.

[\[2\] How to find appropriate initialization values](#)

Redes neurais com a biblioteca SciKit-Learn



- A biblioteca SciKit-Learn **disponibiliza apenas dois tipos de arquiteturas** de redes neurais, MLP e **máquina de Boltzmann restrita**.
- A **máquina de Boltzmann** é implementada através da classe Bernoulli *restricted Boltzmann machine* (BernoulliRBM) e é usada para redução de dimensionalidade, classificação, geração de imagens, etc.
- Além disso, suas implementações **não são flexíveis** e **não se destinam a aplicações de larga escala**.
 - Por exemplo, a biblioteca *SciKit-Learn* não oferece suporte a GPUs.

Redes neurais com a biblioteca SciKit-Learn

- Para implementações de ***modelos de aprendizado profundo*** escaláveis, muito mais rápidos, flexíveis e baseados em GPU, devemos utilizar bibliotecas como:
 - ***Tensorflow***: criada pela equipe *Google Brain* do *Google*.
 - ***PyTorch***: criada pela *Meta AI* (antigo *Facebook*).
 - ***Matlab***
 - ***MXNet***: criada pela *Apache*.
 - ***Caffe***: criada na universidade da Califórnia em Berkeley.
 - ***Theano***: criada pela Universidade de Montreal (primeira versão) e mantida posteriormente pela equipe de desenvolvedores do pacote PyMC sob o nome de Aesara.
 - Entre outras:
https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Tarefas

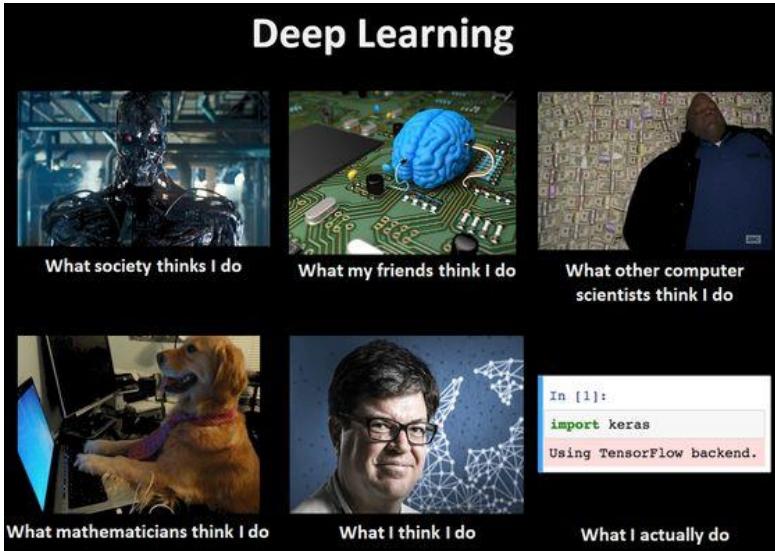
- **Quiz:** “*T320 - Quiz – Redes Neurais Artificiais (Parte VII)*” que se encontra no MS Teams.
- **Projeto:** Projeto #2.
 - Projeto está no github e pode ser feito em grupos de no máximo 3 alunos.
 - **Entrega: 07/12/2025 até às 23:59.**
 - Leiam os enunciados atentamente.
 - Apenas um integrante do grupo precisa fazer a entrega.
 - **Mas não se esqueçam de colocar os nomes de todos os integrantes do grupo.**

Obrigado!

People with no idea
about AI, telling me my
AI will destroy the world



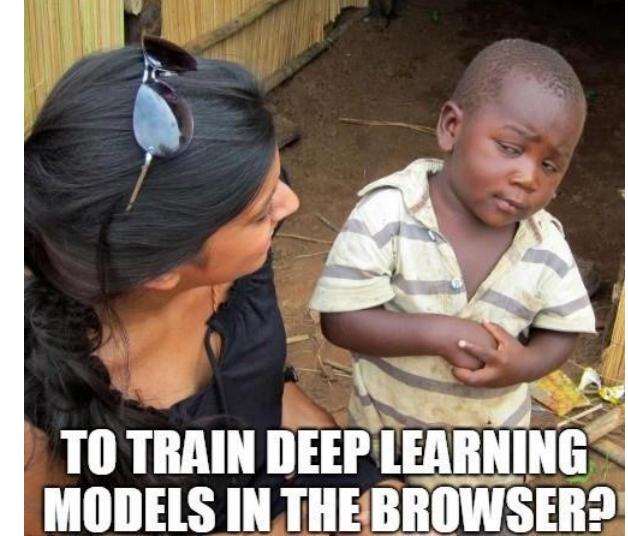
Me wondering why my
neural network is
classifying a cat as a dog..



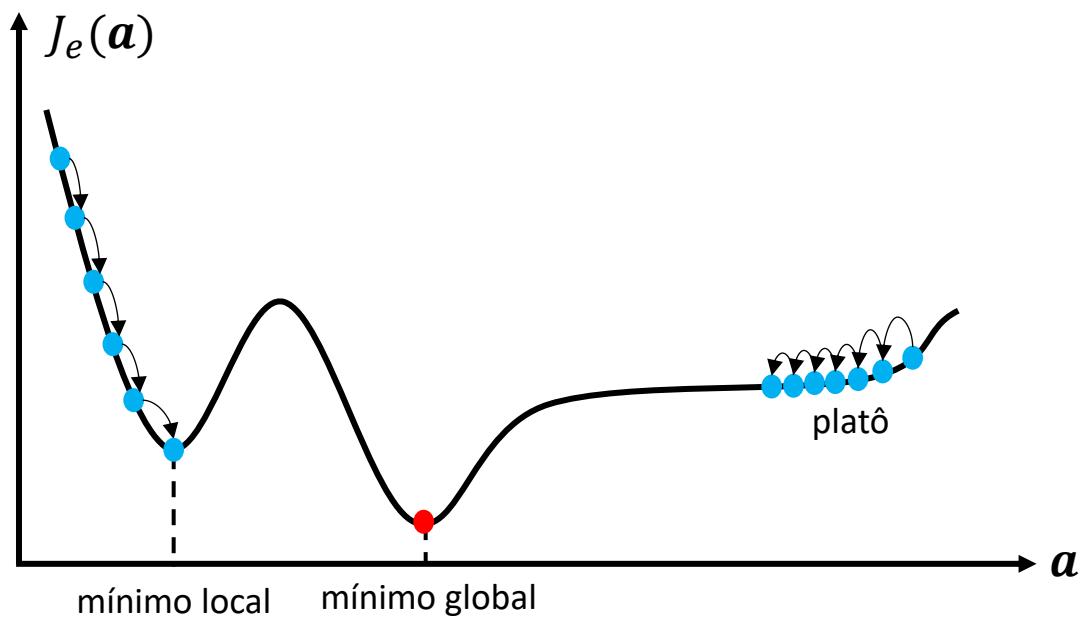
I'M NOT SCARED OF A COMPUTER
PASSING THE TURING TEST...



SO YOU ARE TELLING ME



Figuras



Algumas visões práticas de algoritmos de aprendizado

Versão Online

$$\frac{\partial J(\mathbf{x}(n) \mid \mathbf{w}(k))}{\partial w_{i,j}^m} = \frac{1}{N_M} \sum_{j=1}^{N_M} \frac{\partial(d_j(n) - y_j(n) \mid \mathbf{w}(k))^2}{\partial w_{i,j}^m} = \frac{1}{N_M} \sum_{j=1}^{N_M} \frac{\partial e_j^2(n \mid \mathbf{w}(k))}{\partial w_{i,j}^m} = \nabla J_n(\mathbf{w}(k)).$$