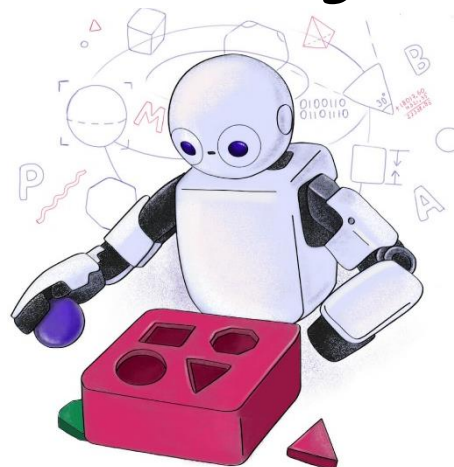


T320 - Introdução ao Aprendizado de Máquina II: *Redes Neurais Artificiais (Parte IV)*



Inatel

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

Recapitulando

- Na última aula, discutimos como as redes neurais aprendem.
- Vimos que isso é feito através da minimização de uma 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 multi-classe e a **focal loss** para o treinamento de detectores de objetos.
- Aprendemos que a minimização da função de custo é realizada iterativamente com a retropropagação do erro até que não haja mais melhoria na performance da rede neural.
- Analisamos como a retropropagação funciona através de um exemplo.
- Nesta aula, iremos discutir algumas visões práticas para o treinamento de redes neurais.

Algumas visões práticas de algoritmos de aprendizado

- Podemos dizer que os ***elementos básicos do aprendizado de máquina*** através de ***redes neurais*** foram apresentados até aqui.
- Porém, existem importantes aspectos práticos que devem ser comentados de modo que vocês fiquem mais familiarizados com as práticas atuais.
- Portanto, começamos relembrando sobre a questão do cálculo do ***vetor gradiente***.

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch


- 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 sinápticos*** que minimizem a ***função de custo***.
- 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, conseqüentemente, o ***erro de saída***.
 - Etapa reversa (***retropropagação/backpropagation***) em que se calculam as derivadas parciais necessárias ao longo das camadas da rede.

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- Vimos também que se calcula o gradiente associado a cada exemplo de entrada e saída da rede e que a média de todos esses **gradientes locais** leva ao gradiente estimado para o conjunto total de exemplos.

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

- 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, surge aqui 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**?

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- Nesse questionamento, existem duas abordagens opostas: o cálculo **online** (ou seja, exemplo-a-exemplo) e o cálculo em batelada (**batch**) do gradiente.
- Vejamos inicialmente a noção geral de **adaptação dos pesos** (sinápticos e bias) com o cálculo **online** do gradiente, como mostra o algoritmo abaixo.

- Defina valores iniciais para o vetor de pesos \mathbf{w} e um passo de aprendizagem α pequeno.
- Faça $k = 0$ (épocas), $t = 0$ (iterações) e calcule $J(\mathbf{w}(k))$.
- Enquanto o critério de parada não for atendido, faça:
 - Ordene aleatoriamente os exemplos de entrada e saídas correspondentes.
 - Para l variando de 1 até N , faça:
 - Apresente o exemplo l de entrada à rede.
 - Calcule $J_l(\mathbf{w}(t))$ e $\nabla J_l(\mathbf{w}(t))$.
 - $\mathbf{w}(t + 1) = \mathbf{w}(t) - \alpha \nabla J_l(\mathbf{w}(t))$.
 - $t = t + 1$.
 - $k = k + 1$.
 - Calcule $J(\mathbf{w}(k))$.

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- O outro extremo seria utilizar todo o conjunto de exemplos para calcular o gradiente antes de atualizar os pesos.
- Essa é a ideia por trás da abordagem em **batelada (batch)**. O algoritmo abaixo ilustra a operação correspondente.

- Defina valores iniciais para o vetor de pesos \mathbf{w} e um passo de aprendizagem α pequeno.
- Faça $k = 0$ (épocas) e calcule $J(\mathbf{w}(k))$.
- Enquanto o critério de parada não for atendido, faça:
 - Para l variando de 1 até N , faça:
 - Apresente o exemplo l de entrada à rede.
 - Calcule $J_l(\mathbf{w}(k))$ e calcule e armazene $\nabla J_l(\mathbf{w}(k))$.
 - $\mathbf{w}(k+1) = \mathbf{w}(k) - \frac{\alpha}{N} \sum_{l=1}^N \nabla J_l(\mathbf{w}(k))$.
 - $k = k + 1$.
 - Calcule $J(\mathbf{w}(k))$.

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- Nas **redes neurais profundas** (ou **deep learning**), usadas com muita frequência em problemas com enormes conjuntos de dados, a regra é adotar o caminho do meio, usando a abordagem com **mini-batches**.
- Nesse caso, a adaptação dos **pesos** é realizada com um gradiente calculado a partir de conjunto com mais de um e menos de N exemplos.
- **OBS.:** As amostras que compõem o **mini-batch** devem ser **aleatoriamente** escolhidas a partir do conjunto de treinamento. O algoritmo abaixo ilustra isso.

- Defina valores iniciais para o vetor de pesos \mathbf{w} , um passo de aprendizagem α pequeno e o tamanho m do mini-batch.
- Faça $k = 0$ (época) e calcule $J(\mathbf{w}(k))$.
- Enquanto o critério de parada não for atendido, faça:
 - Para l variando de 1 até m , faça:
 - Apresente o exemplo l de entrada, amostrado aleatoriamente sem reposição do conjunto de treinamento, à rede.
 - Calcule $J_l(\mathbf{w}(k))$ e calcule e armazene $\nabla J_l(\mathbf{w}(k))$.
 - $\mathbf{w}(k + 1) = \mathbf{w}(k) - \frac{\alpha}{m} \sum_{l=1}^m \nabla J_l(\mathbf{w}(k))$.
 - $k = k + 1$.
 - Calcule $J(\mathbf{w}(k))$.

Variações dos algoritmos de otimização dos pesos

- Existem vários algoritmos baseados no **gradiente** que podem ser empregados para otimizar os **pesos** de uma rede neural.
- Aqui, vamos nos ater aos métodos mais usuais na literatura moderna, que se encontra bastante focada no **apredizado profundo**.

➤ Método do Gradiente Estocástico (*Stochastic Gradient Descent, SGD*)

- Nos slides anteriores, nós vimos que o aprendizado **online** utiliza um único exemplo (tomado aleatoriamente) para **estimar** o gradiente da **função custo**.
- Este tipo de estimador é o que gera a noção de **gradiente estocástico**.
- Caso utilizemos **mini-batches**, também teremos uma estimativa do **gradiente**, o qual, a rigor, seria determinístico apenas se usássemos todos os dados (no caso do **batch**).
- Por esse motivo, esses métodos de **primeira ordem** (ou seja, métodos baseados na derivada parcial de primeira ordem), como o **online**, são conhecidos como métodos de **gradiente descendente estocástico**.

Variações dos algoritmos de otimização dos pesos

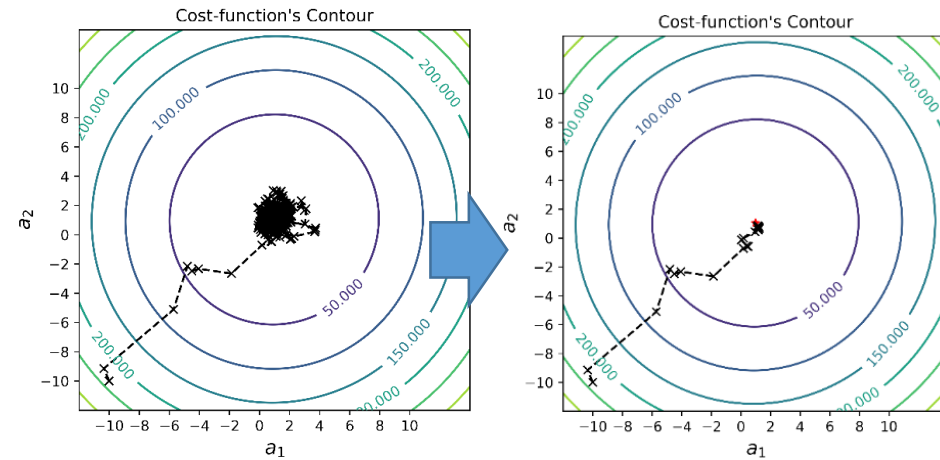
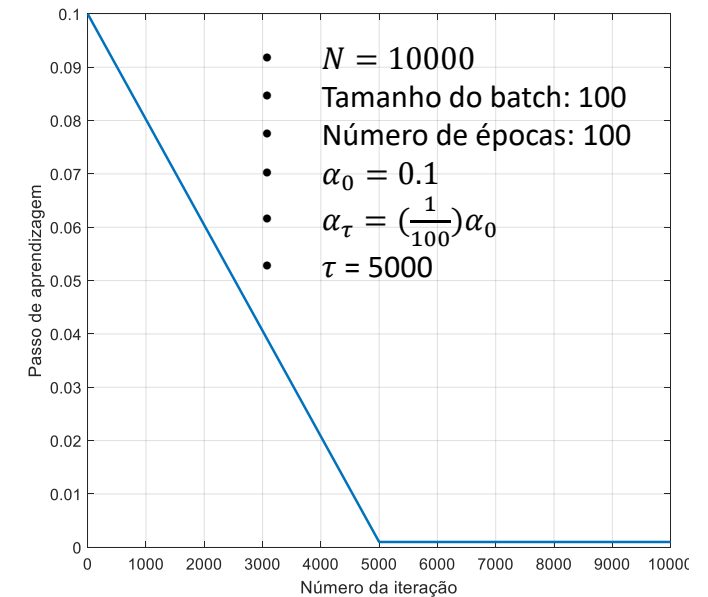
➤ Redução programada do passo de aprendizagem

- A escolha do ***passo de aprendizagem*** é complicada e exige um compromisso entre velocidade de convergência e estabilidade/precisão.
- Pode-se usar um valor fixo, mas geralmente para o GDE e MB, se adota uma variação decrescente de um valor α_0 a um valor α_τ (i.e., da iteração 0 à iteração τ):

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

onde j é o número da iteração de treinamento.

- Após a τ -ésima iteração, deixa-se o valor do passo de aprendizagem fixo, como mostrado na figura ao lado.
- Porém, a definição dos hiperparâmetros α_0 e α_τ , é mais um problema ***a ser tratado caso-a-caso***.



Variações dos algoritmos de otimização dos pesos

➤ Momentum

- O ***termo momento*** é adicionado à equação de atualização dos pesos para trazer ***informação de gradientes anteriores acumulados*** ao seu ajuste.
- Esse termo tem o potencial de aumentar a velocidade de convergência das versões online e em mini-lotes do gradiente descendente e deixá-las mais estáveis.
- A ***atualização dos pesos*** com o ***termo momento*** é dada por

$$w \leftarrow w + v,$$

onde v é a ***velocidade***, a qual é atualizada da seguinte forma

$$v \leftarrow \mu v - \alpha g,$$

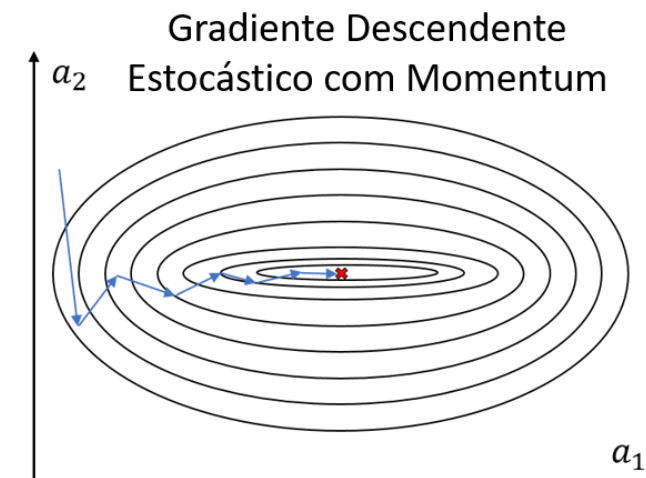
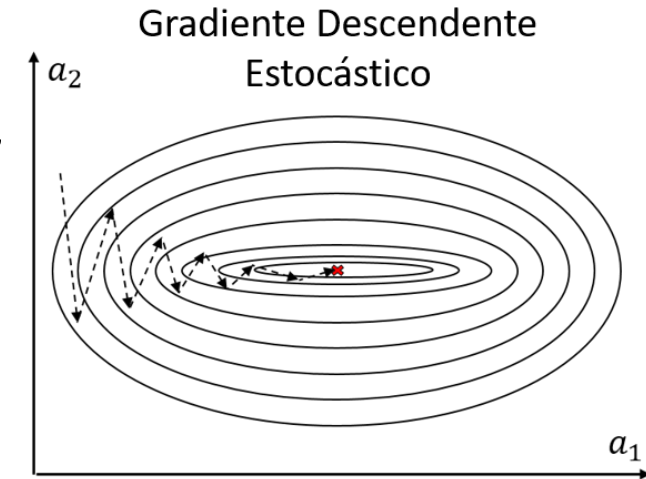
g é o ***vetor gradiente***, α é o ***passo de aprendizagem*** e $\mu \in [0,1)$ é o ***coeficiente de momento*** e determina com que rapidez as contribuições de gradientes anteriores decaem (ou seja, μ é um termo de memória).

- Quanto maior for μ , maior será a influência de gradientes anteriores na direção atual.
- v dá a ***direção*** e a ***velocidade*** na qual os pesos se movem pelo espaço de pesos.

Variações dos algoritmos de otimização dos pesos

➤ Momentum

- Em física, **momento** é igual a **massa de uma partícula vezes sua velocidade**.
- No algoritmo do momento, assumimos que a massa é unitária, então o vetor velocidade \mathbf{v} também pode ser considerado como o momento da partícula.
- O termo momento adiciona uma fração μ de atualizações anteriores dos pesos à atualização corrente.
 - Quando o gradiente aponta na mesma direção por várias iterações, o termo aumenta o tamanho dos passos dados naquela direção.
 - Quando o gradiente muda de direção a cada nova iteração, o termo momento suaviza as variações (figura ao lado).
 - Como resultado, temos convergência mais rápida e oscilação reduzida.



Variações dos algoritmos de otimização dos pesos

➤ **Momento de Nesterov**

- O método do ***momento de Nesterov*** pode ser visto, essencialmente, como uma variação do ***método do momento*** em que o cálculo do ***vetor gradiente*** não é feito sobre o vetor de pesos w , mas sim sobre $w + \mu v$.
- Esse termo adicional funciona como um fator de correção que pode aumentar, em alguns casos, a velocidade de convergência do algoritmo.

➤ **Modelos com Passo de Aprendizagem Adaptativo**

- O ***passo de aprendizagem*** é um hiperparâmetro difícil de se ajustar otimamente e bastante relevante para o sucesso do treinamento de uma rede neural.
- Isso motivou o surgimento de um conjunto de métodos com mecanismos capazes de ajustá-lo dinamicamente.
- O passo é ajustado de acordo com o desempenho da rede, i.e., informação dos gradientes passados.
- Além disso, pode-se ter ***passos diferentes para cada peso do modelo***, os quais são atualizados de forma independente.
- Portanto, são adequados para redes neurais, onde a superfície de erro é diferente em diferentes dimensões, tornando a atualização dos pesos mais efetiva.
- Dentre as técnicas mais populares dessa classe estão ***AdaGrad***, ***RMSProp*** e ***Adam***.

Inicialização dos Pesos

- Uma vez que os métodos de treinamento de **redes neurais MLP** são iterativos, eles dependem de uma **inicialização dos pesos**.
- Como os métodos são de **busca local**, a inicialização pode afetar drasticamente a qualidade da solução obtida.
- O **ponto de inicialização** pode determinar se o algoritmo converge, sendo alguns pontos iniciais tão instáveis que o algoritmo encontra dificuldades numéricas (representações numéricas: **underflow** e **overflow**) e falha completamente em convergir (e.g., desaparecimento e explosão dos gradientes).
- A inicialização também pode fazer com que ocorram variações expressivas na **velocidade de convergência** (e.g., platôs, pontos de sela).
- Um ponto importante da inicialização é “**quebrar a simetria**” entre os **nós**, ou seja, **nós** com a mesma **função de ativação** e conectados às mesmas entradas, devem ter pesos iniciais diferentes.
- Isso, portanto, sugere uma **abordagem de inicialização aleatória**.

Inicialização dos Pesos

- Os pesos iniciais são tipicamente obtidos a partir de ***distribuições gaussianas*** ou ***uniformes***.
- A ordem de grandeza desses pesos levanta algumas discussões:
 - Pesos de maior magnitude criam maior distinção entre ***nós*** (i.e., a ***quebra de simetria***). Por outro lado, isso pode causar problemas de instabilidade.
 - Pesos de maior magnitude favorecem a propagação de informação, porém, por outro lado, causam preocupações do ponto de vista de regularização.
 - Pesos de magnitude elevada podem levar os ***nós*** (no caso de ***funções de ativação*** do tipo sigmóide como a tangente hiperbólica e a função logística) a operarem na região de saturação, comprometendo a convergência do algoritmo.
 - Por outro lado, pesos de magnitude muito reduzida podem reduzir drasticamente o aprendizado das redes neurais.
- Portanto, na sequência listamos algumas ***heurísticas*** para inicialização dos pesos.

Inicialização dos Pesos

- A ideia por trás delas é manter a média das ativações igual a **zero** e a variância das ativações **constante** 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	Nenhuma (Linear), Tanh, Logística, Softmax	$r = \sqrt{\frac{6}{m+n}}$	$\sigma^2 = \frac{2}{m+n}$
He	ReLU e variantes	$r = \sqrt{\frac{6}{m}}$	$\sigma^2 = \frac{2}{m}$
LeCun	SELU	$r = \sqrt{\frac{3}{m}}$	$\sigma^2 = \frac{1}{m}$

- Uma heurística para a inicialização dos **pesos de bias** é inicializá-los com **valores nulos**. Esta heurística se mostra bastante eficiente na maioria dos casos.

Redes Neurais MLP com SciKit-Learn

- Como vimos anteriormente, a biblioteca SciKit-Learn disponibiliza algumas classes para o treinamento de redes neurais multi-layer perceptron.
- Entretanto, suas implementações não se destinam a aplicações de larga escala.
- Em particular, a biblioteca SciKit-Learn não oferece suporte a GPUs.
- Para implementações de ***modelos de aprendizado profundo*** escaláveis, muito mais rápidos, flexíveis e baseados em GPU, devemos utilizar bibliotecas como:
 - ***Tensorflow***: biblioteca para desenvolvimento de aplicações eficientes e escaláveis de machine learning.
 - ***keras***: biblioteca de alto-nível para desenvolvimento de aplicações Deep Learning de forma simples. É capaz de rodar sobre TensorFlow, Theano ou Apache MXNet.
 - ***skorch***: biblioteca para a criação de redes neurais compatíveis com o SciKit-Learn que encapsula a biblioteca PyTorch.
 - Entre outras: https://scikit-learn.org/stable/related_projects.html#related-projects

Tarefas

- **Quiz:** *“T320 - Quiz – Redes Neurais Artificiais (Parte VII)”* que se encontra no MS Teams.
- **Projeto:** [Projeto #2](#).
 - Pode ser feito em grupos de no máximo 3 alunos.
 - **Entrega:** 26/06/2021.
 - Vídeo com a explicação sobre o projeto se encontra na pasta “Projetos” em “Arquivos”.
 - 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..



Deep Learning



What society thinks I do



What my friends think I do



What other computer
scientists think I do



What mathematicians think I do



What I think I do



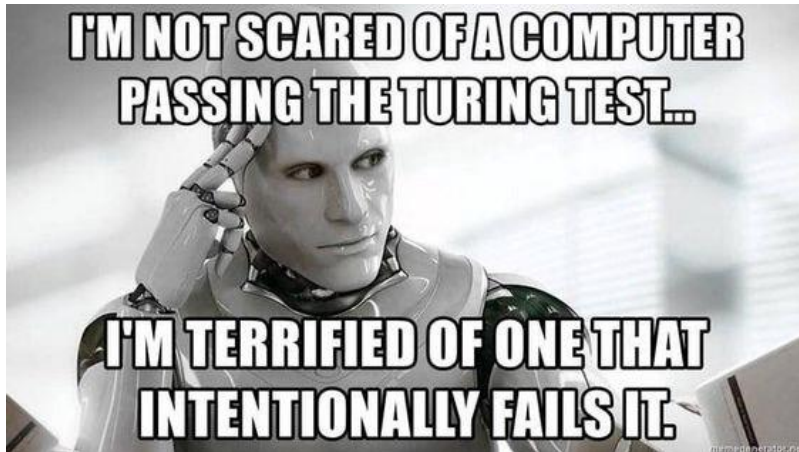
What I actually do

SO YOU ARE TELLING ME



**TO TRAIN DEEP LEARNING
MODELS IN THE BROWSER?**

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



**I'M TERRIFIED OF ONE THAT
INTENTIONALLY FAILS IT.**

Dog



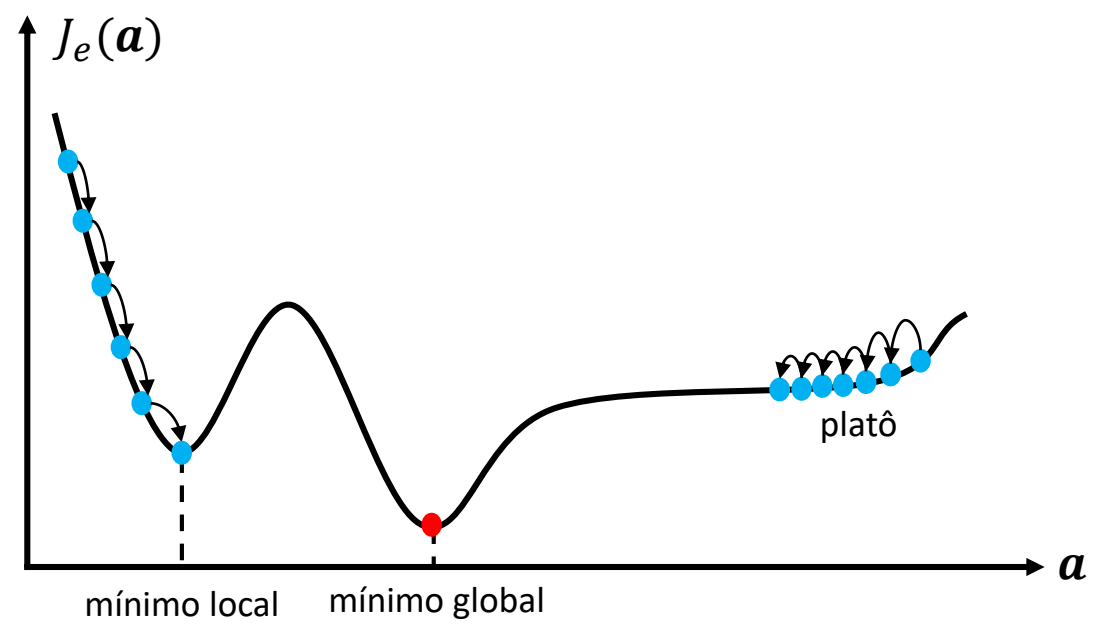
**I NEED GPU
FOR MY DUMB
NEURAL NETWORK**

ONE DOES NOT SIMPLY



**GENERATE MEMES USING DEEP
LEARNING**

Figuras



Algumas visões práticas de algoritmos de aprendizado

Versão Online

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