Luis Fred
luisfred.com.br

Deep Learning

Uma introdução ao deep learning com foco em NLP

Porque você deveria estudar deep learning?

Porque você deveria estudar deep learning

Deep learning tem superado os modelos de machine learning convencionais em termos de performance, com um maior potencial de generalização que tem viabilizado casos de uso em diversas áreas:

- Energia renovável
- Agricultura
- Medicina
- Monitoramento urbano
- Mobilidade urbana
- Controle de mudanças climáticas
- Varejo
- E contando...

ML convencional requer mais esforço manual com engenharia de features

Técnicas de machine learning convencionais são restritas na maneira como os dados brutos são processados. Durante muito tempo, o processo de construção de modelos preditivos envolveu um considerável esforço manual e expertise no domínio de cada problema sendo tratado. A maior parte desse trabalho manual era direcionada para a construção de extratores de features com o objetivo de obter as devidas transformações nos dados brutos, representando-os de tal modo que se adequassem aos algoritmos de aprendizagem.

Deep learning não requer engenharia de features

Por outro lado, algoritmos de deep learning são capazes de extrair features automaticamente a partir dos dados, sem a necessidade de esforço manual. Ou seja, você não precisa escolher um conjunto de regras e algoritmos para extrair as features dos dados, pois o algoritmo de deep learning aprende sobre estas features automaticamente durante o processo de treino.

Processo de aprendizagem hierárquico

Modelos de deep learning utilizam **redes neurais** no processo de aprendizagem, onde cada problema é visto de maneira hierárquica. Deste modo, as camadas inferiores do modelo codificam as representações mais básicas do problema, ao passo em que as camadas de mais alto nível codificam conceitos mais complexos. A última camada é normalmente aquela responsável por trazer a resposta obtida pelo modelo.

Como as redes neurais funcionam?

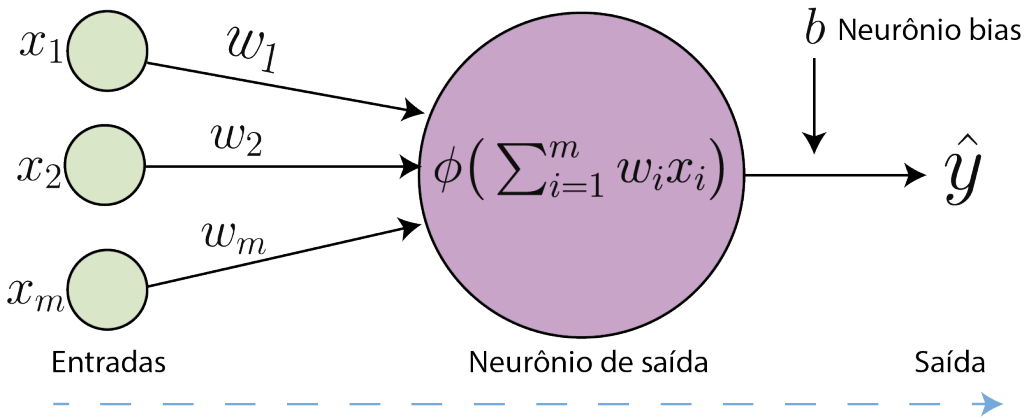
Como uma rede neural trabalha

O objetivo de uma rede neural é obter uma aproximação de alguma função f^*. No caso da aprendizagem supervisionada, $y = f^*(x)$ mapeia vetores de entrada x para um vetor de variáveis categóricas y por meio de uma sequência de transformações que ocorrem ao longo de várias camadas. As transformações que cada camada aplica às suas entradas são parametrizadas por um conjunto de pesos $\theta = w, b$. Deste modo, o modelo define um mapeamento $y = f(x; \theta)$ e encontra os valores de θ que resultam na melhor aproximação, ou que têm o menor erro associado.

Perceptron

A Perceptron

A rede neural mais simples se chama *Perceptron*. Ela contém apenas uma camada (de entrada) e um nó (neurônio) de saída, onde um conjunto de entradas é diretamente mapeado para uma saída usando uma generalização de uma função linear.



A Perceptron

Considere uma situação na qual as amostras de treino estejam no formato $(\mathbf{x}, y)$, onde cada instância $\mathbf{x} = [x_1, ..., x_m]$ contém m features e $y \in \{-1, +1\}$ contém os valores observados das classes binárias. O objetivo do modelo é predizer os valores das classes com respeito aos dados não observados. A camada de entrada contém m nós, que transmitem para o nó de saída as m features $\mathbf{x} = [x_1, ..., x_m]$ com os pessos $\mathbf{w} = [w_1, ..., w_m]$ associados. A função linear $\mathbf{w} \cdot \mathbf{x}$ é computada no neurônio de saída. Esse calculo gera um valor real e o sinal desse valor (ativação) é usado para mapear esse valor real para $+1$ ou -1:

$$\hat{y} = \phi(\mathbf{w} \cdot \mathbf{x}) = \phi(\sum_{i=1}^{m} w_i x_i) + b$$

Como uma rede neural trabalha

O termos bias b é incorporado na rede como um peso adicional que sempre transmote o valor 1 para a saída e serve para lidar com comportamentos invariantes nas predições. A predição feita pelo modelo durante um instante qualquer possui um erro associado $E = y - \hat{y}$. O aprendizado do modelo consiste em ajustar os valores dos pesos de modo a encontrar uma combinação ótima que minimize estes erros. Esse processo de minimização é feito com o auxílio de uma função de perda, o que será mencionado nas próximas aulas. Além disso, os pesos costumam ser atualizados na direção contrária a do gradiente do erro. Nós também veremos isso em detalhes ao longo do curso.

Como uma rede neural trabalha

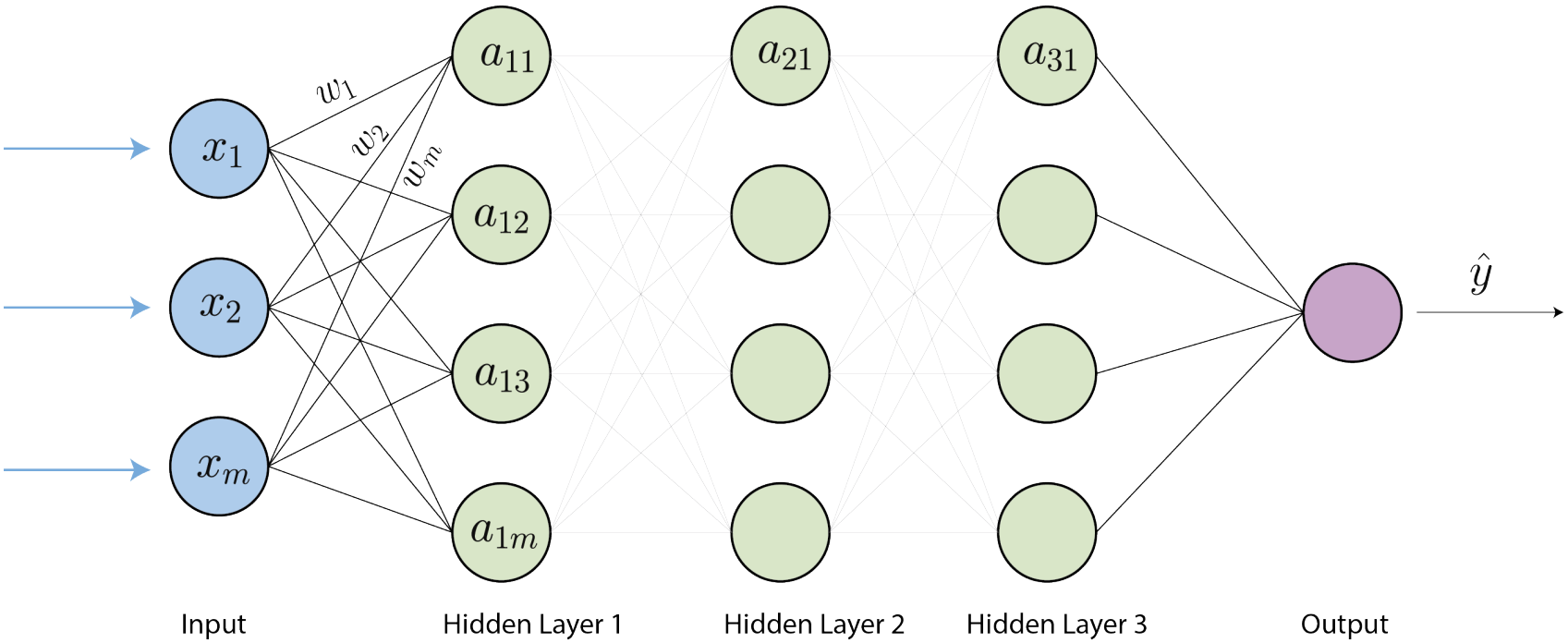
Da mesma forma que ocorre com um cérebro biológico, as redes neurais artificiais precisam de estímulos externos para conseguirem aprender algo. Estes estímulos vêm de um conjunto de dados de aprendizagem, que contêm os pares entrada-saída responsáveis por guiar o treino do modelo. Esses dados fornecem o feedback para a correção dos pesos na rede neural, dependendo de quão bem a saída prevista para uma entrada específica corresponde ao rótulo presente no conjunto de aprendizagem.

Rede neural Multi-Camadas

Rede neural multi-camadas

Na rede neural multi-camadas, os neurônios são arranjados em camadas, onde as camadas de entrada e de saída são separadas por um grupo de camadas ocultas.

Como uma rede neural trabalha



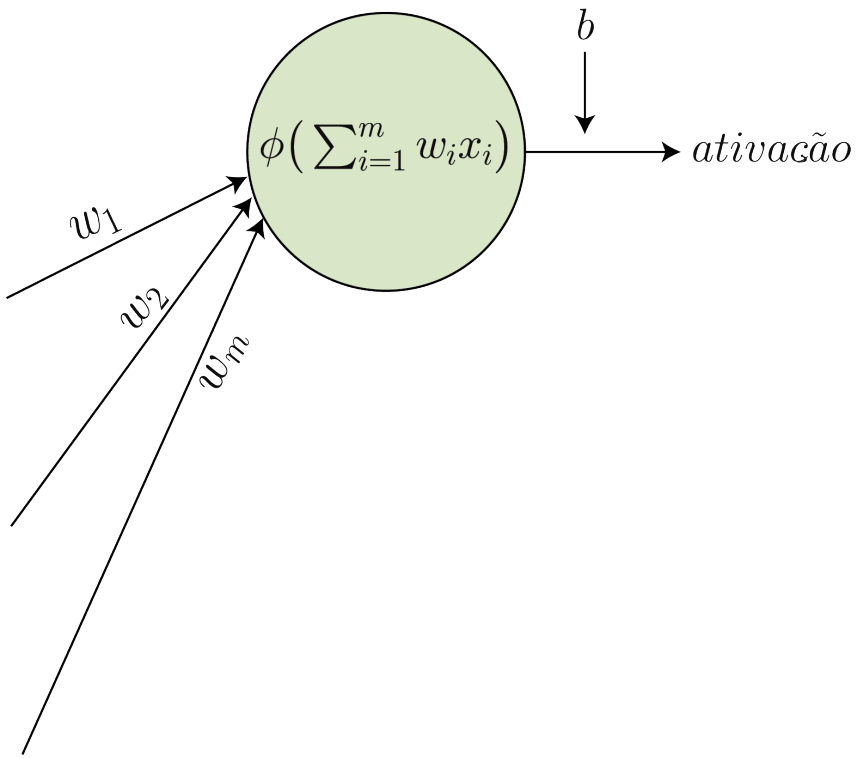
Como uma rede neural trabalha

Cada um desses neurônios é uma unidade computacional que trabalha em paralelo. O verdadeiro potencial das redes neurais vem da combinação dessas unidades, permitindo que o modelo aprenda funções mais complexas. Cada camada subsequente de uma rede neural se conecta à saída da camada anterior e a rede vai aprendendo conceitos mais complexos, incrementalmente, à medida em que avançam as camadas (profundidade). Quanto maior o nível da camada, mais abstratas são as representações obtidas. Veremos um exemplo disso mais adiante.

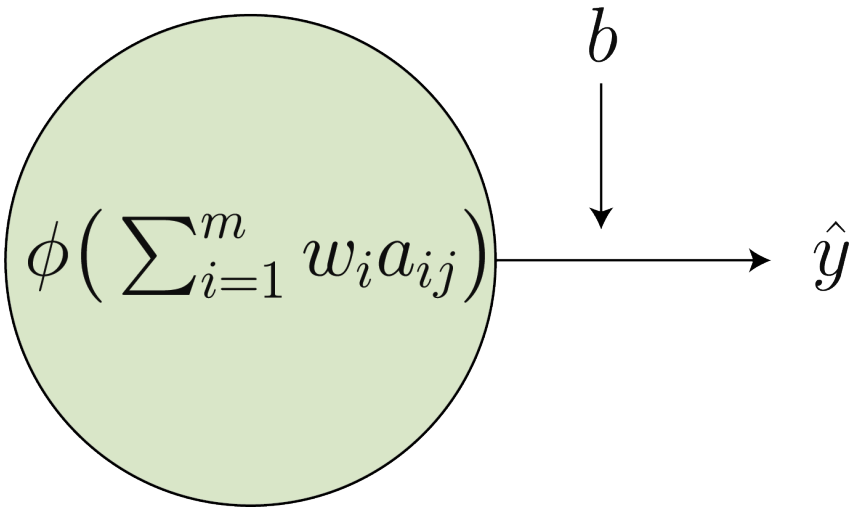
Como uma rede neural trabalha

Cada unidade presente na rede lembra um neurônio biológico por receber as entradas de várias outras unidades e computar suas próprias ativações. A escolha da função de ativação é, então, fundamental para guiar o aprendizado do modelo ao computar os valores das camadas intermediárias. Falaremos sobre funções de ativação nas próximas aulas.

Como uma rede neural trabalha



Como uma rede neural trabalha



Como uma rede neural trabalha

A quantidade de unidades presentes na camada de saída reflete o numero de classes que você está tentando modelar.

Como uma rede neural trabalha

Essas redes são normalmente representadas por uma composição de várias funcões diferentes. Por exemplo, nós poderíamos ter uma composição de cinco funções $f^{(1)}, f^{(2)}, f^{(3)}, f^{(4)}$ e $f^{(5)}$ encadeadas para formar uma rede assim:

$$f(\mathbf{x}) = f^{(5)}(f^{(4)}(f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))))$$

Neste caso, $f^{(1)}$ corresponde à primeira camada da rede, enquanto que $f^{(5)}$ corresponde à quinta camada. Lembre-se, pode haver $f^{(n)}$ camadas, com n representando a profundidade do modelo. É comum termos n na ordem das dezenas ou até centenas - ***é por isso que se chama "deep learning"***.

Como uma rede neural trabalha

Durante a fase de treino, nós queremos que o $f(\mathbf{x})$ chegue tão próximo de $f^*(\mathbf{x})$ quanto possível, com a rede produzindo uma saída $\hat{y}$ que é bastante próxima do y presente no conjunto de aprendizagem. Assim, tal aproximação terá sempre um erro associado durante cada iteração (que é calculado ao comparar $\hat{y}$ com y) e o nosso objetivo é minimizar este erro gradativamente por meio de ajustes infinitesimais aplicados nos pesos.

Como uma rede neural trabalha

Assim, os pesos são ajustados em resposta aos erros de predição cometidos pela rede. O objetivo de alterar os pesos é modificar a função computada com a intenção de melhorar os acertos nas iterações subsequentes. Ao ajustar esses parâmetros sucessivamente, a função computada pela rede é refinada ao longo do tempo, até que seja capaz de obter boas generalizações.

Como uma rede neural trabalha

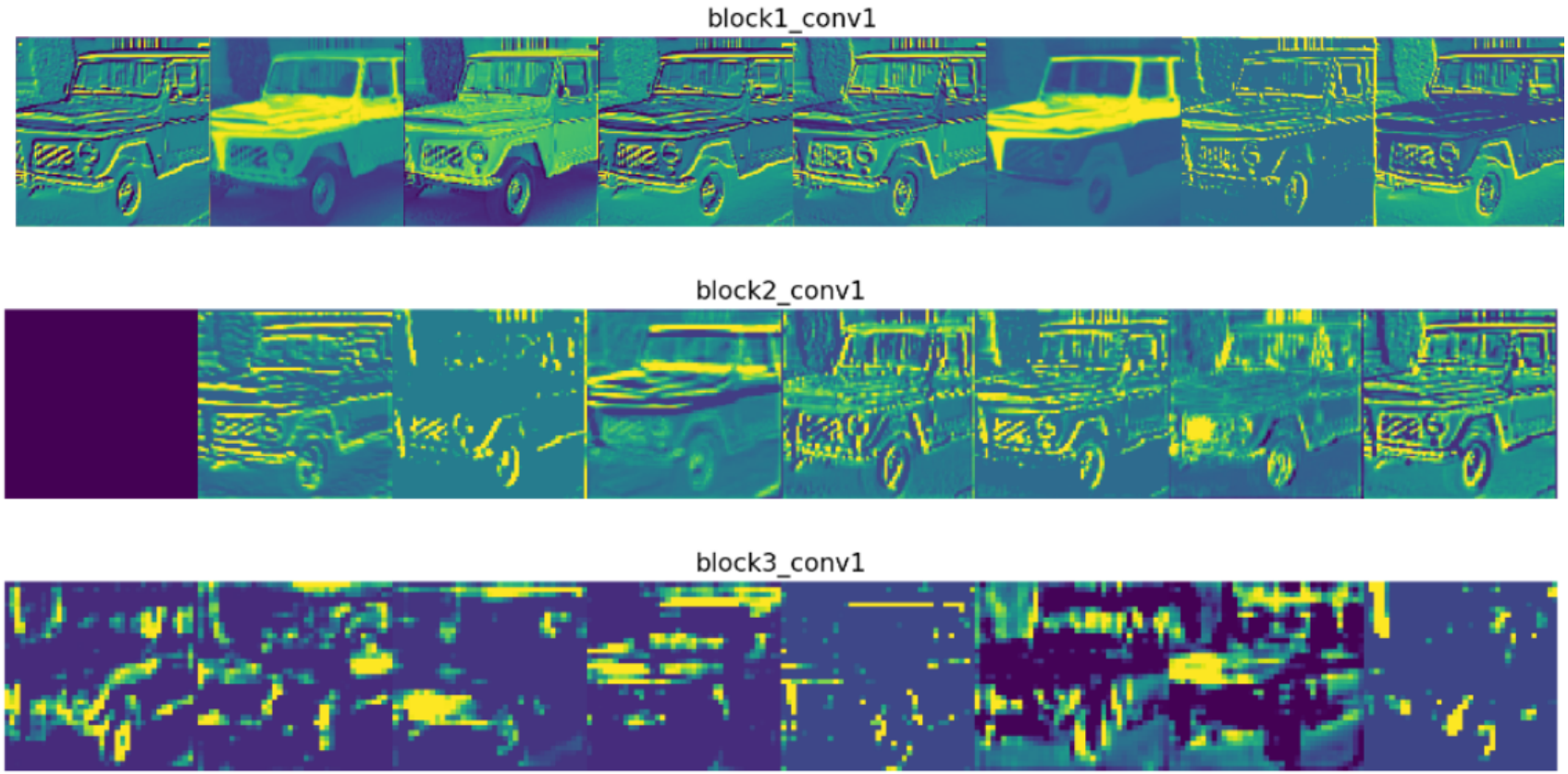
Diferente do que acontece na camada de saída, as camadas intermediárias não sofrem influência direta dos dados de entrada. Ou seja, diferente da camada de saída, as camadas intermediárias não conhecem o valor de y e o algoritmo deve decidir como usar estas camadas para obter a melhor aproximação f^*.

Cada camada de uma rede neural aprende um conjunto específico de conceitos

Processo de aprendizagem hierárquico

Como um modelo de deep learning para classificação de imagens aprenderia os padrões da imagem abaixo?

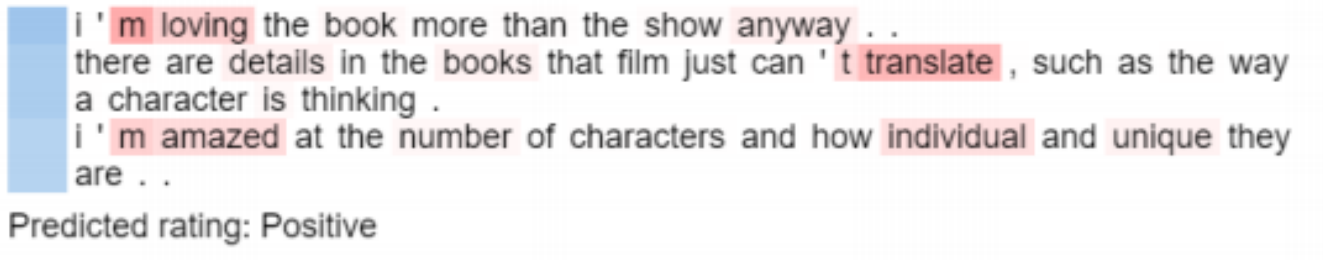
Processo de aprendizagem hierárquico



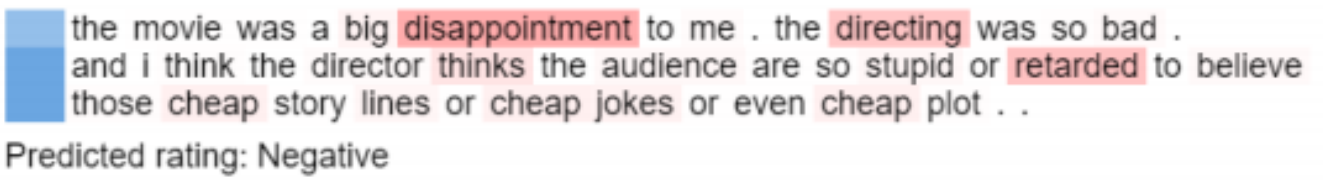
Processo de aprendizagem hierárquico

- A primeira camada extrai uma parte expressiva da informação presente na imagem, agindo na identificação de cantos, contornos, bordas, etc
- Conforme avançamos pelas camadas, notamos que a rede se concentra em maior medida nas representações mais abstratas, codificando informações de mais alto nível, como partes mais específicas do objeto que o distinguem.

Como uma rede neural extrai as features de um texto?



(a) A positive example of visualization of a strong word in the sentence.



(b) A negative example of visualization of a strong word in the sentence.

Figure: Exemplo de como uma rede neural pode "enxergar" as features de um texto levando em conta todo o contexto presente na vizinhança de cada palavra para inferir o seu signifidcado. Fonte: Abreu, J., Fred, L., et al

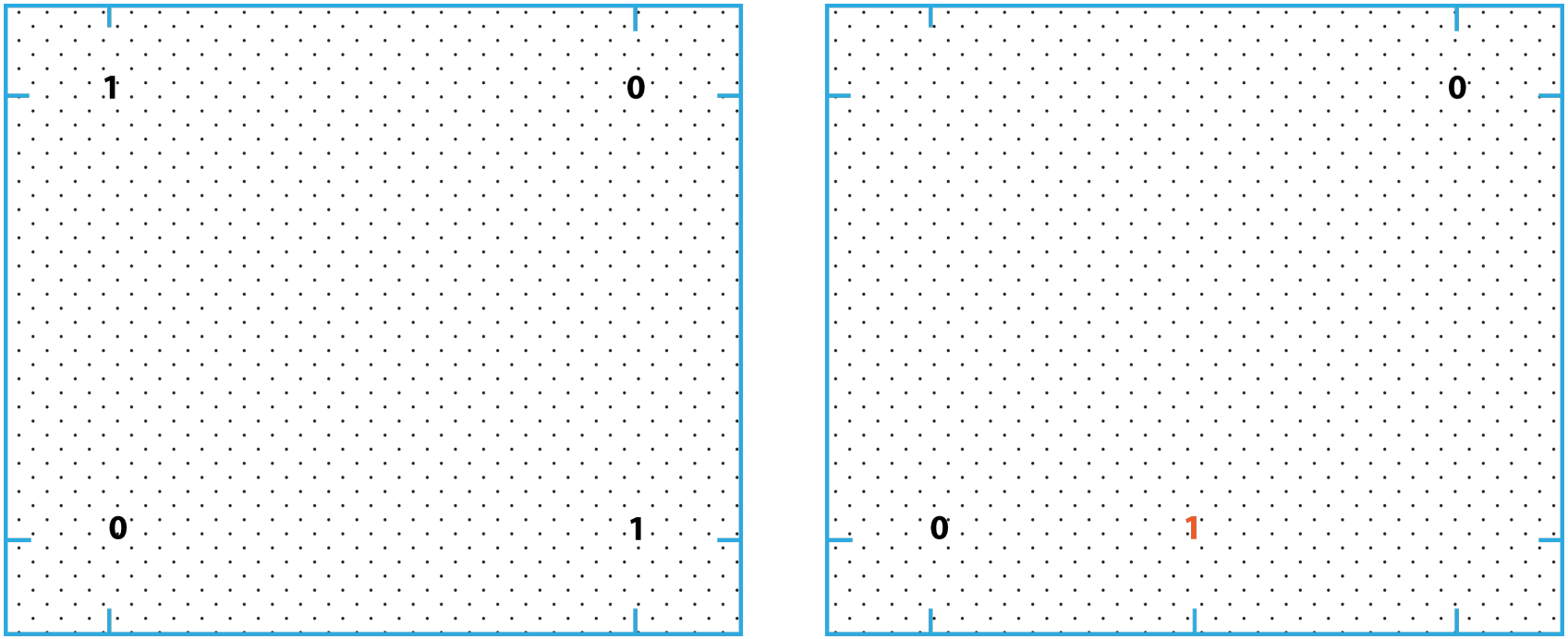
Funções de ativação

$$y = \phi(\mathbf{w} \cdot \mathbf{x})$$

Funções de ativação

A função de ativação é um componente crítico do projeto de qualquer rede neural e sua escolha apropriada costuma ter um impacto significativo na performance final do modelo. Ela aplica uma transformação no nível de ativação de um determinado neurônio, enquadrando os valores das ativações dentro de intervalos específicos, como $[0, 1]$ ou $[-1, 1]$. As funções de ativação, quando combinadas ao longo das camadas, também garantem a separação linear entre os pontos de dados das diferentes classes, ao criar um mapeamento não linear entre eles.

Funções de ativação



Espaço original

Espaço transformado

Figure: As features não lineares foram mapeadas para um único ponto no espaço de features, permitindo uma separação linear.

Funções de ativação

$$y = \phi(\mathbf{w} \cdot \mathbf{x})$$

Funções de ativação

São exemplos de funções de ativação usadas atualmente em redes neurais:

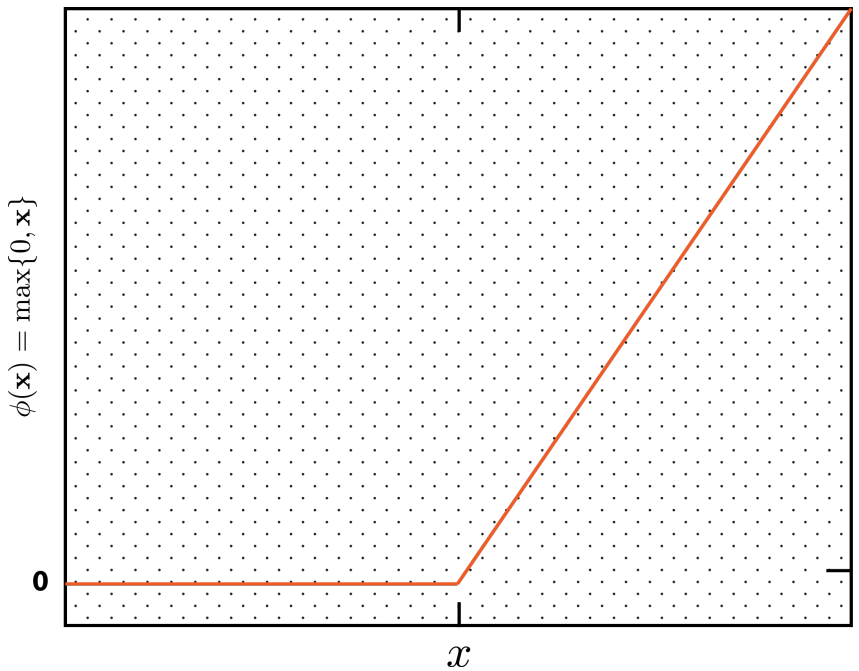
- Rectified Linear Unit (ReLU)
- LeakyReLU
- Softmax

Rectified Linear Unit (ReLU)

A ReLU é a função de ativação padrão de quase toda rede neural moderna. Ela tem derivada parcial igual a $\mathbf{1}$ diante de valores positivos e $\mathbf{0}$ para valores negativos, pois ela satura abaixo de $\mathbf{0}$. Por ser muito próxima de uma função linear, ela compartilha das mesmas propriedades que fazem com que os modelos lineares sejam fáceis de otimizar usando métodos baseados em gradiente.

Rectified Linear Unit (ReLU)

$$\phi(\mathbf{x}) = \max\{0, \mathbf{x}\}$$

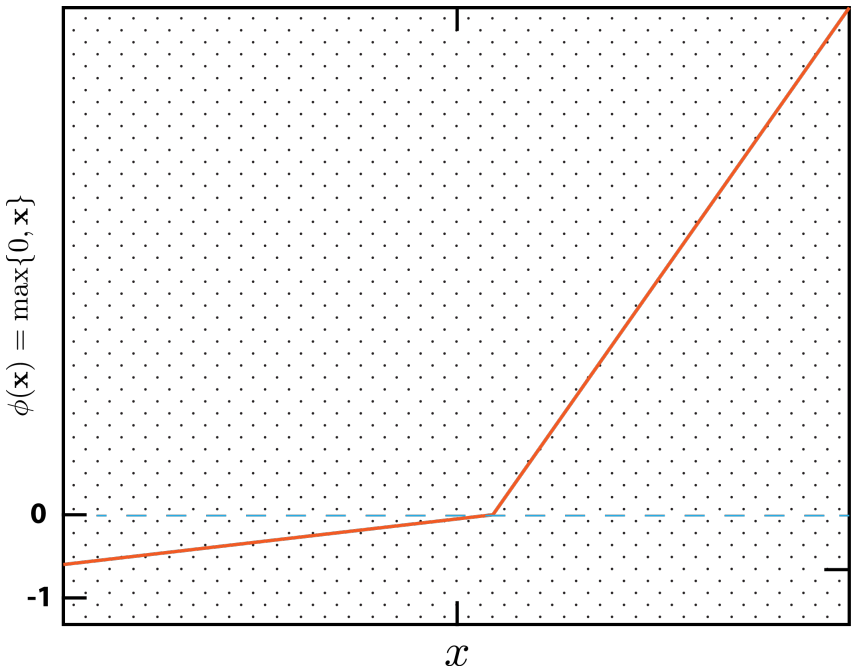


LeakyReLU

A ReLU tem o inconveniente de ignorar valores negativos, o que pode ser prejudicial para o aprendizado da rede neural em algumas situações. Por exemplo, pode haver situações onde o neurônio nunca entra em ativação, impedindo o ajuste dos pesos. A **LeakyReLU** é uma resposta para esse problema, computando gradientes de uma pequena fração da parte negativa, permitindo que as unidades ativem para uma determinada faixa de valores negativos.

LeakyReLU

$$\phi(\mathbf{x}) = \max\{0, \mathbf{x}\} = \begin{cases} \mathbf{x} \text{ se } \mathbf{x} > 0 \\ 0.01\mathbf{x} \text{ se } \mathbf{x} \leq 0 \end{cases}$$



Softmax

Frequentemente usada na camada de saída de uma rede neural para classificação (é mais indicada para problemas envolvendo múltiplas classes). Ela aplica uma transformação nos valores de saída dos neurônios, para que o valor **de cada unidade** esteja no intervalo $[\mathbf{0}, \mathbf{1}]$. Ela também garante que a soma de todas as saídas não ultrapasse o valor de $\mathbf{1}$. A saída da softmax é equivalente a uma distribuição de probabilidades categóricas e nos diz a probabilidade de cada uma das classes ser a verdadeira.

Softmax

$$\phi(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{n} e^{x_k}}$$

Softmax

$$\phi\left(\begin{bmatrix}1.2\\0.9\\0.4\end{bmatrix}\right) = \begin{bmatrix}0.46\\0.34\\0.20\end{bmatrix}$$

Retropropagação e Gradiente descendente

Como uma rede neural é treinada

$$x^* = \arg\min f(x)$$

Gradiente descendente Retropropagação

A tarefa de treinar uma rede neural é formulada em termos de minimizar uma função de custo $f(x)$, a qual calcula o custo de um conjunto de pesos quando eles estão em um determinado ponto do espaço de soluções a cada iteração do algoritmo no conjunto de aprendizagem. O objetivo é encontrar um conjunto ideal de valores para os pesos, de tal forma que á função de custo produza o menor valor possível.

Gradiente descendente Retropropagação

A **retropropagação** trata de propagar no sentido contrário, da última camada para a primeira, o custo produzido na saída, utilizando-o para computar o gradiente e atualizar os pesos. Esse processo usa a **regra da cadeia do cálculo** com **derivação parcial**, sendo repetido várias vezes até que o erro seja minimizado. Nesta técnica, todos os pesos são inicializados aleatoriamente ou inicializados usando alguma distribuição de probabilidades. Os pesos da última camada são atualizados primeiro, depois os da camada anterior e assim por diante, até chegar na primeira camada e atualizar os pesos dela também. Uma vez que o backpropagation tenha computado o gradiente, o algoritmo do gradiente descendente otimiza a função custo com o valor desse gradiente.

Gradiente descendente Retropropagação

O algoritmo da retropropagação, por meio do gradiente descendente que cumpre o papel de otimizador, aplica em cada peso um ajuste que é proporcional à derivada parcial da função de custo com respeito àquele peso. Por meio da regra da cadeia, a derivada parcial é usada para determinar a direção na qual buscar o melhor ajuste de cada peso dentro de um amplo espaço de possibilidades.

Função de custo

Uma função custo muito conhecida e utilizada nesse processo é a dos *erros médios quadráticos*:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$$

onde m representa a quantidade de amostras disponível no conjunto de aprendizagem e θ são os parâmetros (ou pesos) que queremos ajustar com o objetivo de minimizar o custo da função acima. Além disso, $\hat{y}^{(i)}$ é a predição feita pela rede na i-ésima amostra de treino com base nos parâmetros que vão sendo ajustados iterativamente.

Gradiente descendente Retropropagação

O **Gradiente descendente** é a técnica de otimização normalmente usada para calcular os gradientes necessários para atualizar, de forma iterativa, os valores dos pesos da rede e orientar a função de custo a seguir na direção de valores muito pequenos. Ele mede o gradiente local dessa função com respeito a um vetor de pesos θ e procura seguir, iterativamente, na direção que produza um valor de gradiente cada vez menor, até que ele seja zero.

Gradiente descendente Retropropagação

Mas o que é **GRADIENTE** e como usamos isso em redes neurais?

Gradiente descendente Retropropagação

O gradiente ∇f é uma função de valor vetorial que armazena valores de suas derivadas parciais.

$$\nabla \mathrm{f} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \\ \frac{\partial f}{\partial y} \\ \\ \vdots \end{bmatrix}$$

Se nos imaginarmos em um ponto (x_0, y_0) no espaço de entrada de f, o vetor gradiente $\nabla f(x_0, y_0)$ nos diria em qual direção deveríamos nos movimentar para aumentar o valor de f mais rapidamente. Se quisermos minimizar o valor de f, o vetor $-\nabla f(x_0, y_0)$ nos ajudaria com a direção.

Gradiente descendente Retropropagação

No contexto das redes neurais, f é uma função de custo nos parâmetros θ, frequentemente denotada por $J(\theta)$ e $(x, y, z, ...)$ representam esses parâmetros. Na literatura, os parâmetros costumam ser referenciados como $\theta = \{w, b\}$. O vetor gradiente $\nabla J(\theta)$, que é computado via retropropagação, contém todas as derivadas parciais da função custo, sendo uma para cada peso do modelo.

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\theta) \end{bmatrix}$$

Gradiente descendente Retropropagação

Nós estamos representando $\nabla_\theta J(\theta)$ como um vetor coluna apenas com o objetivo de simplificar a demonstração. O gradiente é na verdade composto por uma matriz com várias derivadas parciais de primeira ordem, chamada **Matriz jacobiana**. Cada elemento de $\nabla_\theta J(\theta)$ é computado usando a regra da cadeia:

$$\frac{\partial J(\theta)}{\partial \theta_{ji}} = \frac{\partial J(\theta)}{\partial e_j} \frac{\partial e_j}{\partial \phi_j} \frac{\partial \phi_j}{\sum_{i=0}^{m} w_{ji}\phi_i} \frac{\sum_{i=0}^{m} w_{ji}\phi_i}{\partial \theta_{ji}}$$

Onde θ_{ji} representa o peso j na camada i, e_j representa o erro cometido pelo neurônio j e ϕ_j é a ativação do neurônio j.

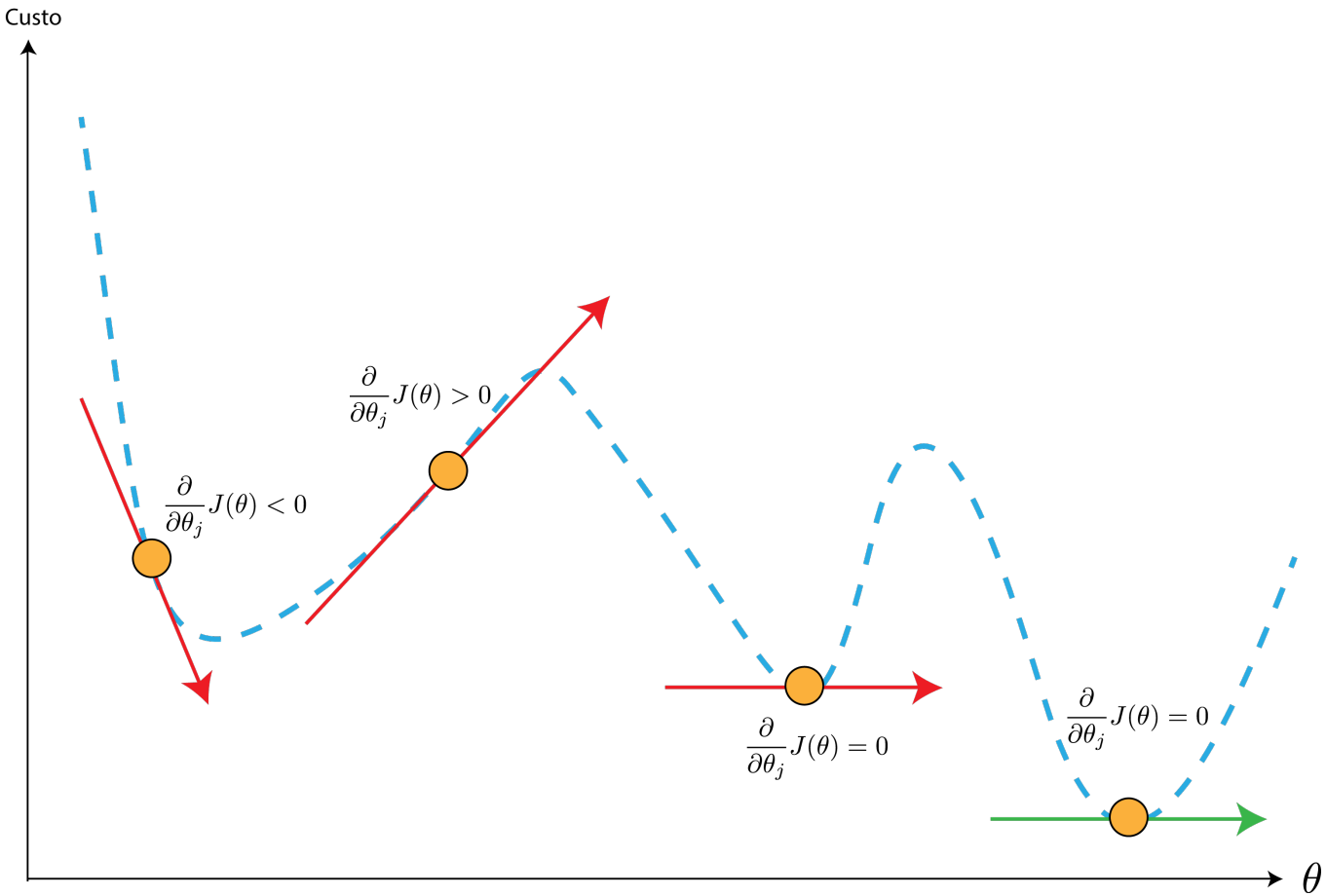
Gradiente descendente Retropropagação

O vetor de pesos θ normalmente começa sendo preenchido com **valores aleatórios** e nós vamos ajustando esses pesos gradativamente, na tentativa de diminuir o valor produzido pela função de custo. O tamanho do ajuste é controlado por um parâmetro externo η chamado **taxa de aprendizagem**, tipicamente tendo um valor pequeno como 0.001. É preciso computar o gradiente da função de custo com respeito a cada um dos pesos θ_j, pois precisamos saber o tamanho do impacto que uma pequena alteração nos pesos causa na função custo. É por isso que usamos derivação parcial.

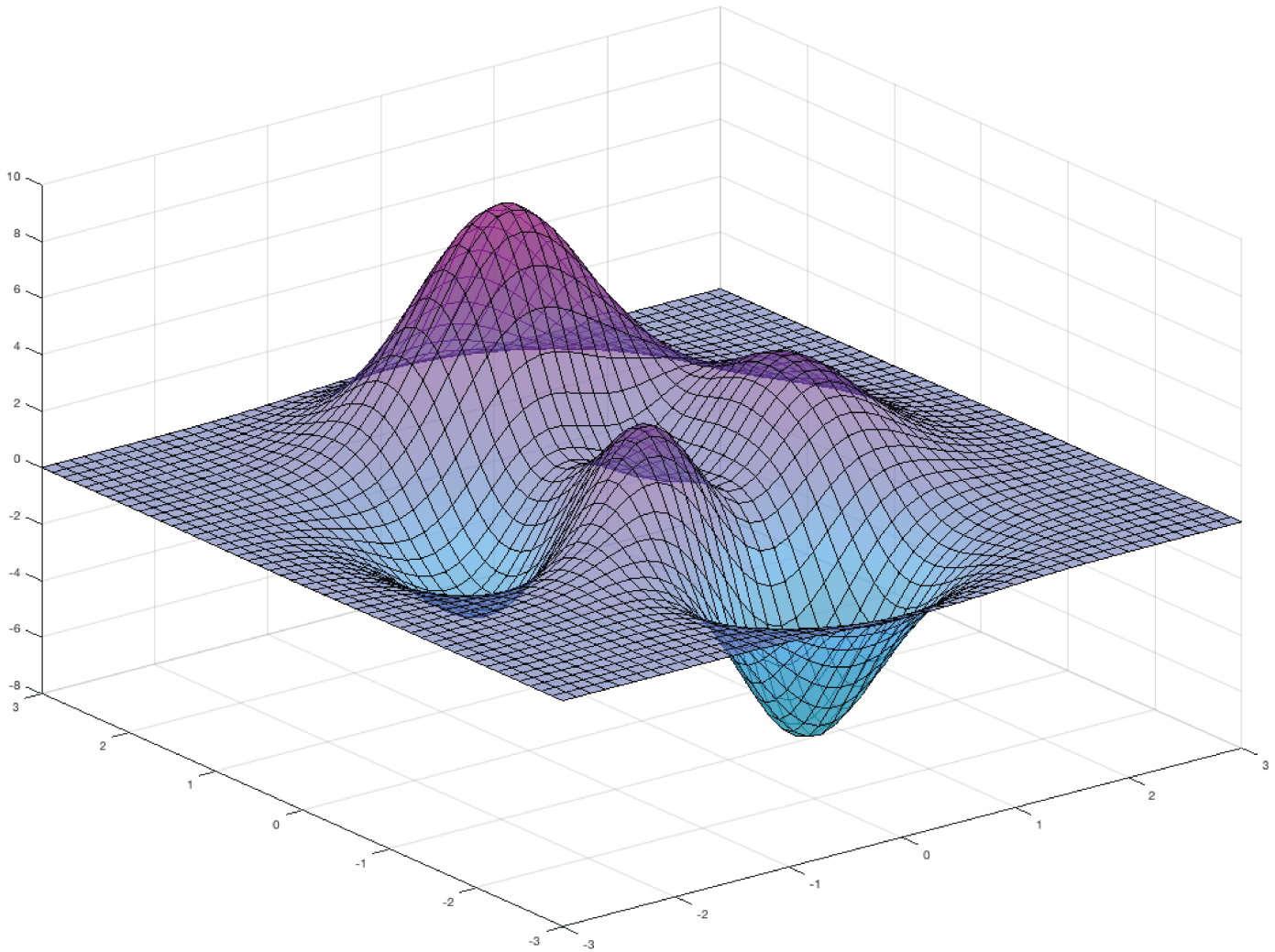
Gradiente descendente Retropropagação

A derivada é usada porque nos fornece a inclinação da superfície durante cada posição da função custo no espaço n-dimensional de entrada, nos dizendo quando a função está aumentando ou diminuindo neste ponto. A derivada basicamente representa um linha que é tangencial a algum ponto em uma superfície. Sabendo a inclinação atual, fica mais fácil estimar em que direção dá o passo seguinte. É como descer um vale com os olhos fechados, tentando encontrar o melhor percurso a cada passo até chegar na parte de baixo, com a taxa de aprendizagem controlando o tamanho de cada um dos seus passos. Você pode dá um passo em qualquer direção, mas qual seria a melhor direção?

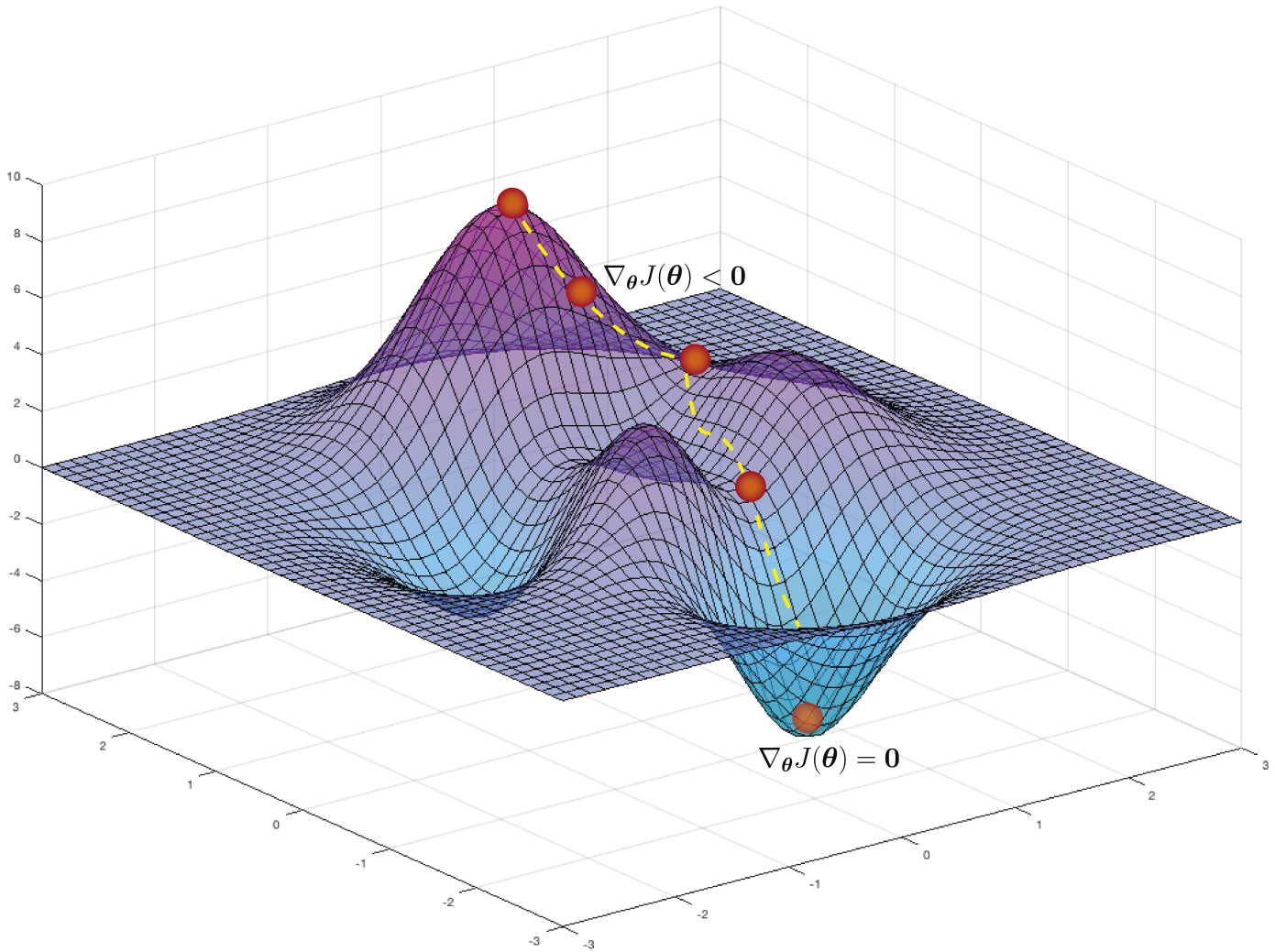
Gradiente descendente Retropropagação



Gradiente descendente Retropropagação



Gradiente descendente Retropropagação



Gradiente descendente Retropropagação

Uma vez tendo o vetor gradiente calculado, o qual aponta pra cima por padrão, basta percorrer na direção oposta. Isto significa subtrair $\nabla_\theta J(\theta)$ de θ. É aqui onde a taxa de aprendizado η desemprenha seu papel. Ao multiplicar o vetor gradiente $\nabla_\theta J(\theta)$ por η, nós conseguimos ponderar o tamanho do ajuste a ser promovido nos pesos:

$$\theta^{\text{(next step)}} = \theta - \eta \nabla_\theta J(\theta)$$

Gradiente descendente Retropropagação

$$f(\theta_0, \theta_1) = \theta_0^2 + 3\theta_0\theta_1 + 7\theta_0 - \theta_1^2$$

$$\nabla \mathrm{f} = \begin{bmatrix} 2\theta_0 + 3\theta_1 + 7 \\ 3\theta_0 - 2\theta_1 \end{bmatrix}$$

Gradiente descendente Retropropagação

$$\text{Seja } \theta_0 = 4, \theta_1 = 5, \eta = 0.5$$

$$\nabla \text{f} = \begin{bmatrix} 2 * 4 + 3 * 5 + 7 \\ 3 * 4 - 2 * 5 \end{bmatrix} = \begin{bmatrix} 30 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} \theta_0^{(\text{next step})} \\ \theta_1^{(\text{next step})} \end{bmatrix} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} - \eta \nabla \text{f}$$

$$\begin{bmatrix} \theta_0^{(\text{next step})} \\ \theta_1^{(\text{next step})} \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix} - 0.5 \begin{bmatrix} 30 \\ 2 \end{bmatrix} = \begin{bmatrix} -11 \\ 4 \end{bmatrix}$$

Gradiente descendente Retropropagação

A função de custo, então, é computada para os novos valores dos pesos $\begin{bmatrix} -11 \\ 4 \end{bmatrix}$ e espera-se que tenha um valor inferior ao custo computado para $\begin{bmatrix} 4 \\ 5 \end{bmatrix}$

PONTOS CRÍTICOS

Gradiente descendente Retropropagação

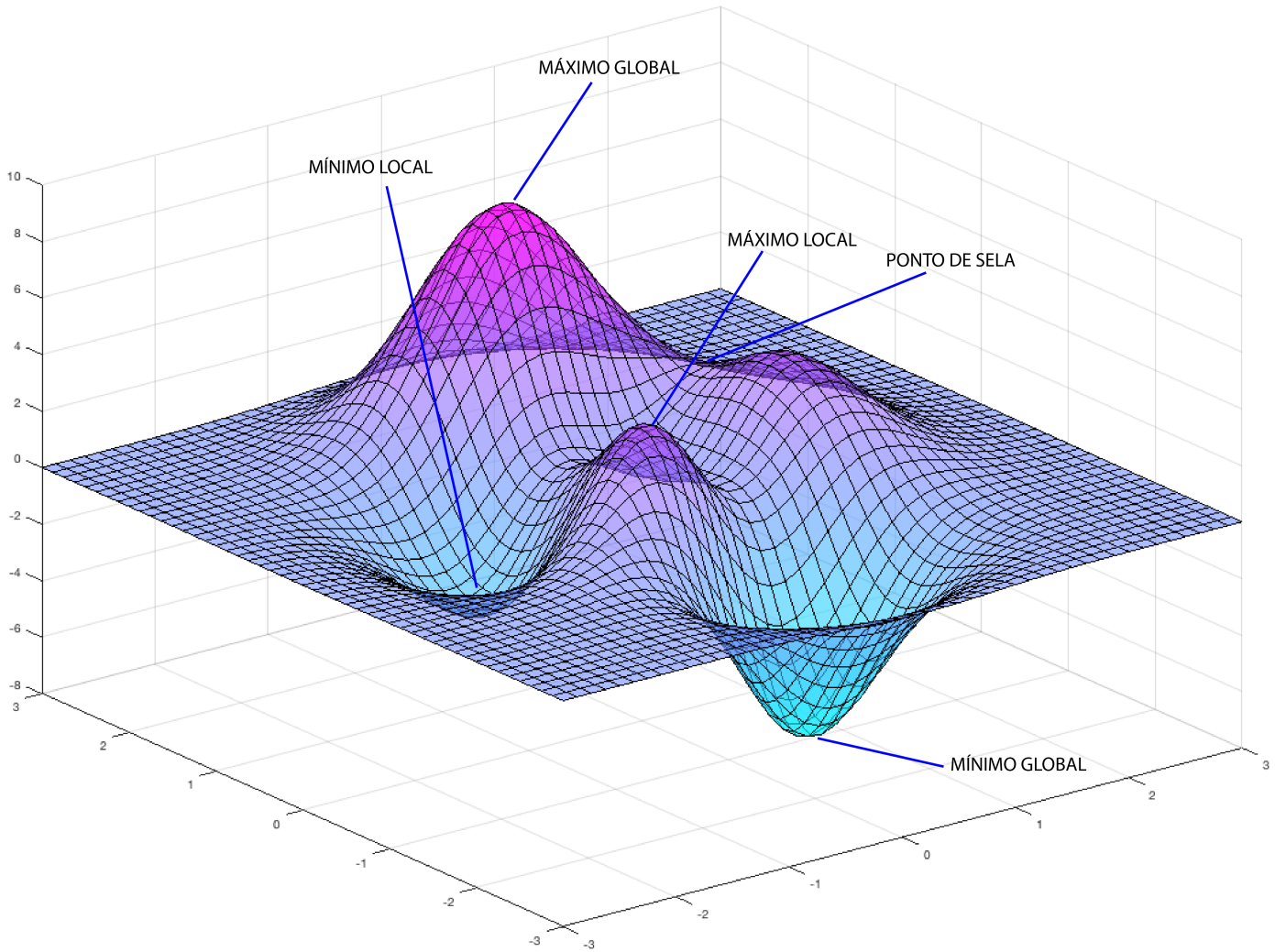
Pontos críticos:

- Máximo local
- Mínimo local
- Ponto de sela

Gradiente descendente Retropropagação

Quando a derivada em um determinado ponto for nula, ela não fornece informações sobre em qual direção mover a função custo. Esses pontos onde a derivada é nula são conhecidos como pontos críticos. Um **mínimo local** é um ponto que tem um $f(x)$ menor do que seus pontos vizinhos. Um **máximo local** é um ponto que tem $f(x)$ maior do que todos os seus vizinhos. Pontos críticos que não são nem mínimos e nem máximos locais são conhecidos como **pontos de sela**. O problema com esses pontos é que o gradiente pode ficar preso em um deles e o valor do custo nunca diminuir. É neste momento que entra mais um componente das redes neurais conhecido como **otimizador**. Falaremos sobre otimizadores ao longo deste curso.

Gradiente descendente Retropropagação



Gradiente descendente Retropropagação

- Stochastic Gradient Descent
- Mini-batch Gradient Descent

Stochastic Gradient Descent

O problema com o gradiente descendente é que ele usa todo o conjunto de aprendizagem de uma vez a cada iteração para computar os gradientes e atualizar os pesos. É algo que pode exigir um esforço computacional bastante elevado se o conjunto de aprendizagem for muito grande. O **Stochastic Gradient Descent**, por outro lado, pega apenas uma instância aleatória (estocástico) do conjunto de aprendizagem a cada iteração e computa os gradientes com base nesta instância (eles faz o mesmo com todas as instâncias), no lugar da base toda de uma vez. Assim, é evidente que o SGD é muito mais eficiente e escalável computacionalmente, considerando que ele precisa de menos dados para manipular durante cada iteração.

Stochastic Gradient Descent

A desvantagem do SGD vem justamente da sua natureza estocástica, que o torna bem menos regular do que o gradiente descendente comum, que é também conhecido como *Batch Gradient Descent*. No lugar de diminuir a função custo suavemente até alcançar o valor mínimo, o SGD provoca flutuações nela. No fim das contas, isso faz com que a função de custo caia apenas próximo do valor mínimo, flutuando em seu entorno sem nunca tocá-lo. Assim, os valores finais dos parâmetros são apenas aceitáveis, não ótimos. Contudo, reduzir a taxa de aprendizagem gradualmente pode ajudar a contornar esse problema.

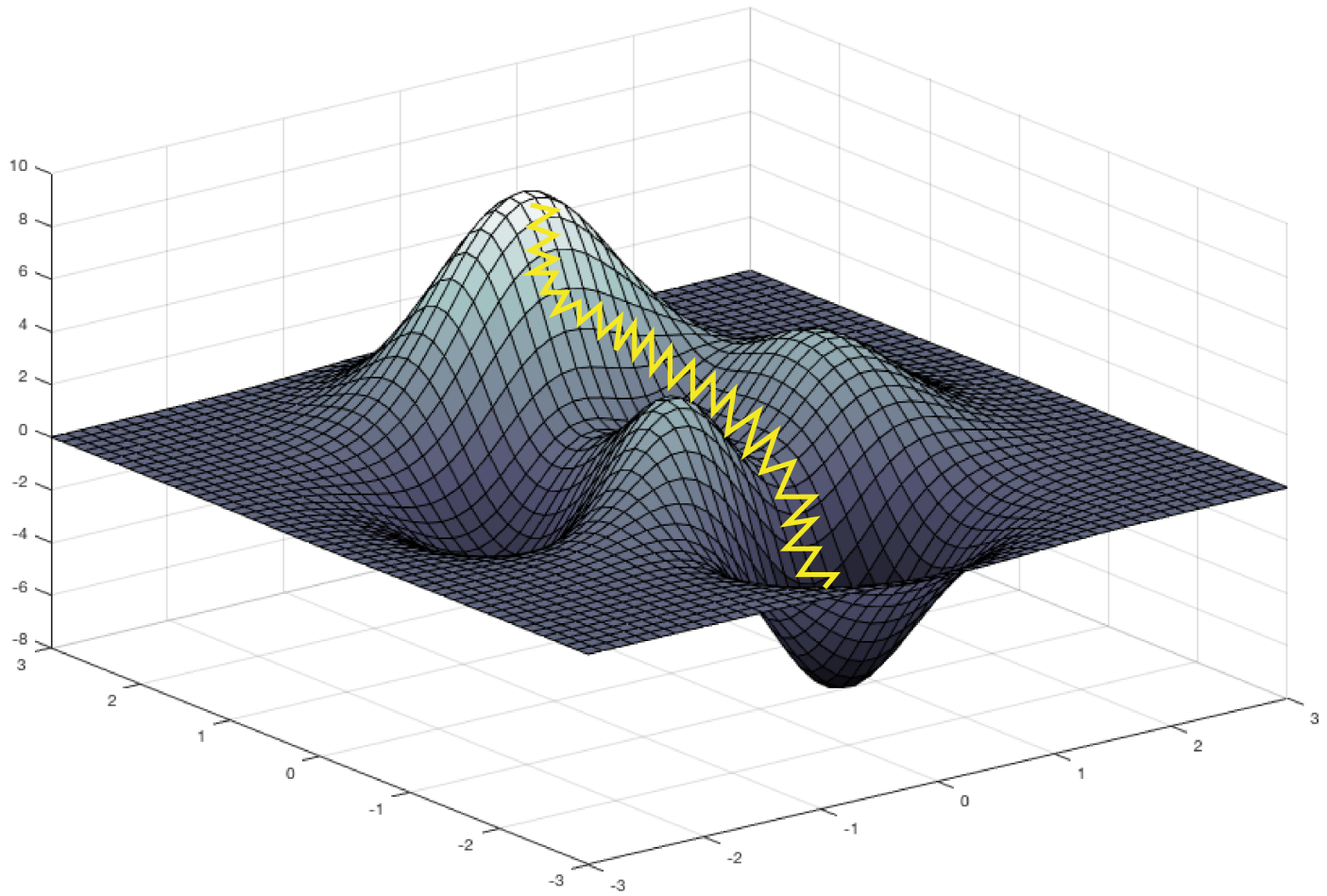
Stochastic Gradient Descent

Além disso, esse comportamento "arisco" do SGD tem uma grande utilidade, pois pode ajudar o algorítmo a passar por cima um mínimos locais (ele ainda pode ficar preso em algum), dando ao SGD melhores chances de alcançar o mínimo global.

Mini-batch Gradient Descent

No lugar de usar todo o conjunto de aprendizagem para computar o gradiente, ou de usar uma instância por vez, o **Mini-batch Gradient Descent** computa o gradiente em pequenas frações aleatórias (também não é suave como o GD) dos dados denominadas *mini-batches*. Trata-se de uma combinação do GD com o SGD. A principal vantagem disso é poder tirar proveito de operações em GPU, agilizando o treinamento do modelo. Além disso, o uso de mini-batches torna este método menos errático do que o SGD, fazendo com que o custo caia mais próximo do mínimo.

Gradiente descendente estocástico



Funções de perda

Função de perda

Treinar uma rede neural também significa minimizar iterativamente os erros de generalização cometidos pelo modelo. Esses erros são medidos por meio de uma função objetivo $f(x)$ que precisa diminuir a diferença entre $\hat{y}$ e y a cada iteração.

Função de perda

No contexto das redes neurais para classificação, objeto de estudo durante esse curso, a função de perda mais popular tem sido a *Cross-Entropy Loss*. Há duas versões dela:

- Binary Cross-Entropy loss
- Categorical Cross-Entropy loss

Função de perda

Binary Cross-Entropy

Binary Cross-Entropy loss

A *Binary Cross-Entropy loss* é útil sempre que quisermos que a rede produza uma probabilidade condicional de classe para um problema de classificação binária. Assim, assume-se que a camada de saída passe por uma transformação usando a função logística. Assumindo que $y \in \{0, 1\}$, a saída do classificador $\tilde{y}$ passa primeiro por uma transformação usando a função logística $\sigma(x) = 1/(1 + e^{-x})$, de tal forma que seja forçada a permanecer dentro do intervalo $[0, 1]$. Em seguida, é interpretada como uma probabilidade condicional $\hat{y} = \sigma(\tilde{y}) = P(y = 1|\mathbf{x})$.

Binary Cross-Entropy loss

A regra assumida pela *Binary Cross-Entropy* para fazer uma classificação é:

$$prediction = \begin{cases} 0 & \hat{y} < 0.5 \\ 1 & \hat{y} \geq 0.5 \end{cases}$$

A rede neural é treinada para maximizar a probabilidade condicional logarítmica $P(y = 1|\mathbf{x})$ para cada exemplo $(\mathbf{x}, y)$ presente no conjunto de aprendizagem. A Binary Cross-Entropy é, então, definida como:

Binary Cross-Entropy loss

$$L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Função de perda

Categorical Cross-Entropy

Categorical Cross-Entropy loss

A *Categorical Cross-Entropy loss* é muito conhecida na literatura das redes neurais e adequada à situações onde desejamos interpretar probabilisticamente os scores obtidos. Indicada para classificadores multi-classe, essa função não apenas ajuda a fornecer o rótulo da classe mais provável, como também fornece uma distribuição de probabilidades dos possíveis rótulos. Ela também requer que a saída do classificador seja transformada, mas agora por uma função *softmax*.

Categorical Cross-Entropy loss

Seja $\mathbf{y} = \{y_1, y_2, ..., y_n\}$ um vetor respresentando n rótulos de um conjunto de aprendizagem composto por n classes. Seja $\hat{\mathbf{y}} = \{\hat{y}_1, \hat{y}_2, ..., \hat{y}_n\}$ a saída do classificador, que é transformada por uma função softmax e representa uma distribuição condicional de classes $\hat{\mathbf{y}}_\mathbf{i} = P(y = i|\mathbf{x})$. A Categorical Cross-Entropy mede a dissimilaridade entre a distribuição dos verdadeiros valores dos rótulos $\mathbf{y}$ e a distribuição dos rótulos estimada $\hat{\mathbf{y}}$ pela rede. Também conhecida como *verossimilhança logaritmica negativa*, a Categorical Cross-Entropy é definida assim:

$$L_{\text{cross-entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i \mathbf{y}_i \log(\hat{\mathbf{y}}_i)$$

Categorical Cross-Entropy loss

Quando os scores $\hat{\mathbf{y}}$ são transformados pela função sofmax, eles passam a ser positivos e seu somatório é igual a $\mathbf{1}$. Assim, se aumentassemos o score atribuído à classe verdadeira, todos os outros valores diminuiriam e o somatório de todos os valores permaneceria igual a $\mathbf{1}$.

OUTROS OTIMIZADORES

Outros otimizadores

Otimizadores tradicionais como o Gradiente Descendente ou o Gradiente Descendente Estocástico podem ser bastante lentos, dependendo do modelo que você estiver tentando obter. E quando o gradiente fica preso em algum mínimo local, ou ponto de sela, dificilmente obteremos uma solução ótima usando esses otimizadores. Para solucionar problemas dessa natureza, outros otimizadores mais rápidos foram concebidos:

- Gradiente descendente com Momento
- RMSProp
- Adam

Gradiente descendente com Momento

O gradiente descendente comum possui oscilações e avança a pequenos passos em direção ao mínimo global. Esse comportamento pode transformar o processo de treino da rede neural em algo bastante lento, principalmente com taxas de aprendizado mais altas.

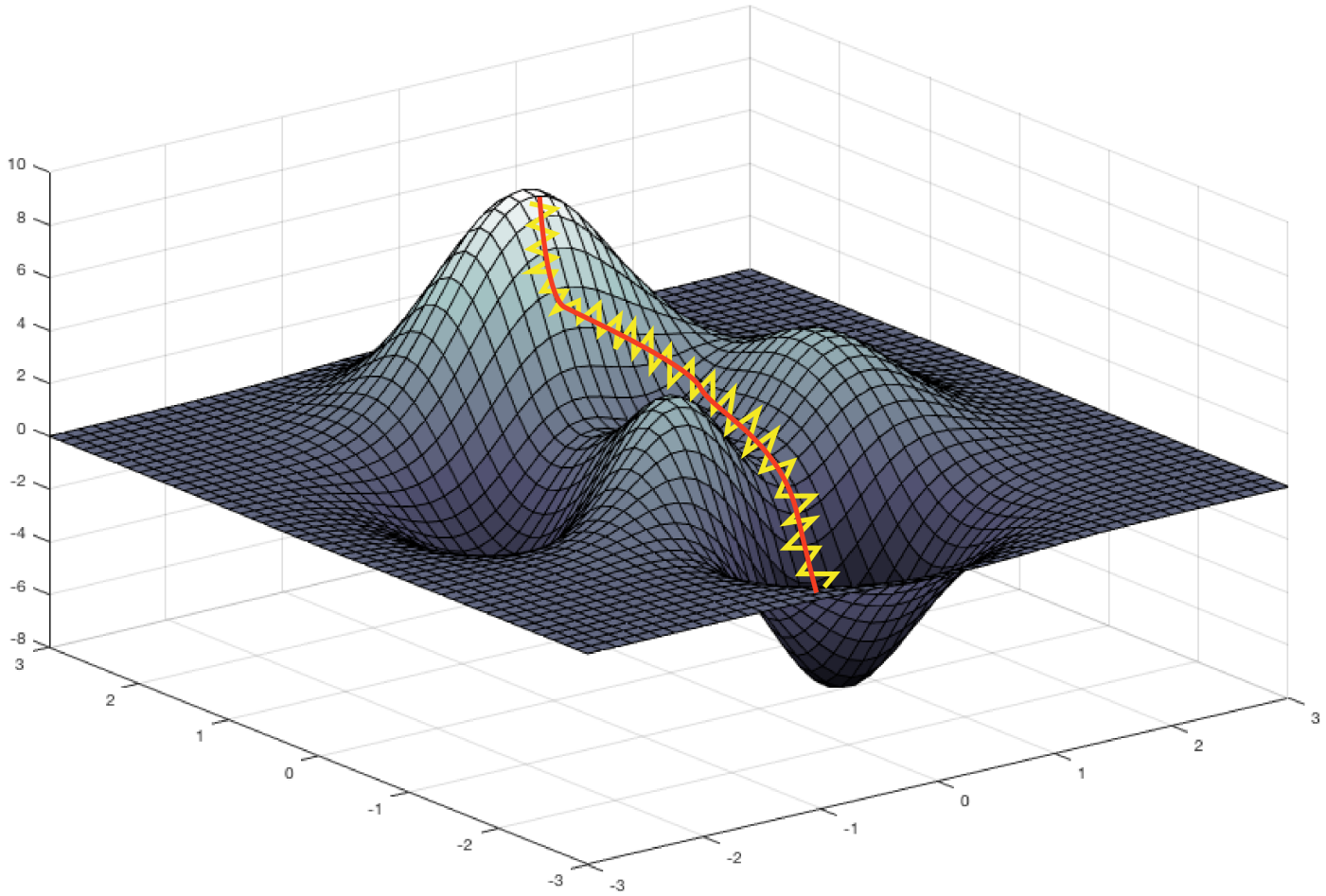
Gradiente descendente com Momento

O **Gradiente descendente com Momento** oferece uma solução para esse problema, ao aplicar uma média móvel nos gradientes computados e usar isto para atualizar os pesos, evitando oscilações e resultando num ganho significativo de aceleração no processo de otimização. Essa é a velocidade necessária para que o gradiente consiga escapar de mínimos locais muito mais rapidamente, caso fique preso em algum, fazendo a função convergir mais rápido. O algoritmo usa um hiperparâmetro β para ponderar o tamanho do momento, controlando a velocidade.

$$\mathbf{m} = \beta\mathbf{m} - \eta\nabla_\theta J(\theta)$$

$$\theta = \theta + \mathrm{m}$$

Gradiente descendente estocástico



RMSProp

Alguns otimizadores incluem métodos de decaimento controlado da taxa de aprendizagem, que ocorre mais rapidamente em pontos mais íngremes da superfície. O problema com o decaimento da taxa de aprendizagem é que a taxa de aprendizagem pode diminuir muito, ao ponto de a convergência parar antes de chegar a um ponto ótimo.

RMSProp

O **RMSProp** tenta evitar isso ao acumular os gradientes apenas das iterações mais recentes. Para isso, a cada iteração, ele armazena em um vetor $\mathbf{s}$ os quadrados das derivadas parciais da função custo com respeito aos pesos e essa operação é ponderada por uma **taxa de decaimento** β, que controla o decaimento da taxa de aprendizagem.

$$\mathbf{s} = \beta\mathbf{s} + (1 - \beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$\theta = \theta \text{ - } \eta\nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

onde ϵ é um termo que assume um valor muito pequeno, apenas para evitar problemas como divisões por zero.

Adam

O **Adaptive moment estimation - Adam**, combina os conceitos dos otimizadores com Momento e do RMSProp, sendo computacionalmente mais eficiente e praticamente não exigindo o *tuning* de hiperparâmetros. Ele calcula a média móvel exponencial tanto do gradiente quando do quadrado do gradiente, controlando as taxas de decaimento dessas médias com base em dois parâmetros: β_1 (taxa de decaimento do primeiro momento) e β_2 (taxa de decaimento do segundo momento). Ao funcionar bem com conjuntos de aprendizagem grandes e redes neurais maiores e mais complexas, o Adam se tornou bastante popular, sendo uma escolha padrão no design de redes neurais atualmente. Além disso, o algoritmo costuma funcionar ligeiramente melhor do que o RMSProp.