

Disciplina: Aprendizado Profundo

Aula 2: O Perceptron e Modelos Lineares

Eliezer de Souza da Silva
sereliezer.github.io
eliezer.silva@ufc.br

Mestrado e Doutorado em Ciência da Computação, Universidade Federal do
Ceará (MDCC / UFC)

10 de Setembro de 2025

Roteiro da Aula de Hoje

- 1 Revisão e Roteiro
- 2 O Perceptron Clássico
 - Limitações: Problema do XOR
- 3 Redes de uma camada: regressão e classificação
- 4 A Perspectiva Estatística
 - Modelo Linear Generalizado (GLM)
 - Família de distribuições exponencial e GLM
- 5 Treinamento e Otimização
- 6 Aprendizado Linear Não-Supervisionado
- 7 Próximos Passos

Recapitulando a Aula 1

Conceitos-Chave

- **Aprendizado como Aproximação de Funções:** O objetivo é encontrar uma função h em um espaço de hipóteses \mathcal{H} que generalize bem para novos dados.

Recapitulando a Aula 1

Conceitos-Chave

- **Aprendizado como Aproximação de Funções:** O objetivo é encontrar uma função h em um espaço de hipóteses \mathcal{H} que generalize bem para novos dados.
- **Fundamento Probabilístico:** O princípio da Máxima Verossimilhança (MLE) nos permite derivar funções de custo a partir de suposições sobre os dados (Gaussiana \rightarrow Erro Quadrático; Bernoulli \rightarrow Cross-Entropy).

Recapitulando a Aula 1

Conceitos-Chave

- **Aprendizado como Aproximação de Funções:** O objetivo é encontrar uma função h em um espaço de hipóteses \mathcal{H} que generalize bem para novos dados.
- **Fundamento Probabilístico:** O princípio da Máxima Verossimilhança (MLE) nos permite derivar funções de custo a partir de suposições sobre os dados (Gaussiana \rightarrow Erro Quadrático; Bernoulli \rightarrow Cross-Entropy).
- **Redes Rasas vs. Profundas:** A grande vantagem do aprendizado profundo é a capacidade de *aprender* as representações dos dados (via funções de base) mais adequadas para o problema.

A Inspiração: O Neurônio Biológico e o Modelo Lógico I

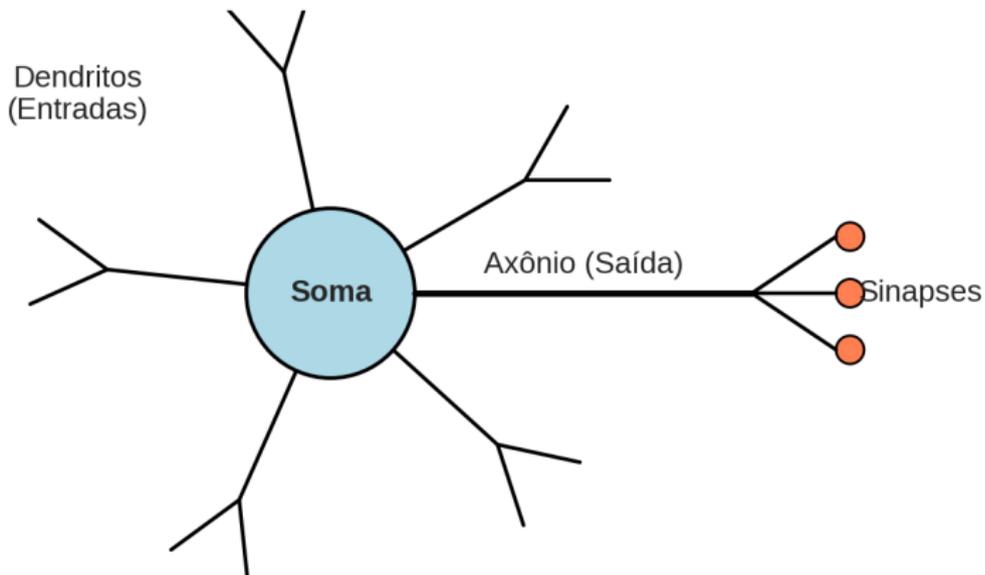


Figura: Modelo simplificado de um neurônio biológico

A Inspiração: O Neurônio Biológico e o Modelo Lógico II

O Neurônio de McCulloch & Pitts (1943)

O primeiro modelo matemático foi concebido como um **circuito lógico**.

- **Entradas e Saídas Booleanas:** $x_i, y \in \{0, 1\}$.
- **Pesos Fixos:** Os pesos w_i eram pré-definidos (manualmente) para implementar funções lógicas como AND, OR, NOT.
- **Ativação:** Um limiar θ fixo. A saída é 1 se $\sum w_i x_i \geq \theta$, senão 0.

A Inspiração: O Neurônio Biológico e o Modelo Lógico III

Computação, Mas Sem Aprendizado

A grande contribuição foi mostrar que redes desses neurônios poderiam, em tese, computar qualquer função booleana, ou seja, definindo a existência de equivalência entre essas redes neurais e classes de funções lógicas.

Leitura Adicional

Warren S. McCulloch e Walter Pitts (1943). “A Logical Calculus of the Ideas Immanent in Nervous Activity”. Em: *The Bulletin of Mathematical Biophysics* 5.4, pp. 115–133. DOI: [10.1007/bf02478259](https://doi.org/10.1007/bf02478259)

O Modelo Perceptron (Rosenblatt, 1958) I

As Duas Grandes Inovações

O Perceptron de Rosenblatt foi revolucionário por introduzir:

- 1 Pesos Reais e Ajustáveis:** Os pesos w e o viés b são números reais que podem ser modificados.
- 2 Um Algoritmo de Aprendizagem:** Uma regra para ajustar esses pesos automaticamente com base nos dados.

O Modelo Perceptron (Rosenblatt, 1958) II

Estrutura e Regra de Aprendizado

- **Ativação:** Função degrau (Heaviside), resultando em uma classificação binária.

$$\hat{y} = \sigma(w^T x + b) = \begin{cases} 1 & \text{se } w^T x + b \geq 0 \\ 0 & \text{se } w^T x + b < 0 \end{cases}$$

- **Regra de Aprendizado:** Para cada exemplo (x_n, y_n) , com $\tilde{x}_n = (x_n, 1)$, os pesos $w^{(t+1)} = \{w, b\}$ são ajustados com base no erro:

$$w^{(t+1)} \leftarrow w^{(t)} + \eta(y_n - \hat{y}_n)\tilde{x}_n$$

O Modelo Perceptron (Rosenblatt, 1958) III

Aprendizado por Supervisão: O Papel do Erro

O Perceptron é um exemplo canônico de aprendizado supervisionado.

- O “supervisor” é o rótulo correto, y_n , de cada exemplo.
- O aprendizado é **guiado pelo erro**. A cada passo, o modelo compara sua predição (\hat{y}_n) com o rótulo verdadeiro (y_n).
- O termo $(y_n - \hat{y}_n)$ gera um **sinal de erro**. Se a predição está correta, o erro é zero e os pesos não mudam. Se está errada, o erro é não-nulo e os pesos são ajustados para mover a fronteira de decisão.
- Esta ideia de **correção de erro iterativa** é um dos pilares do aprendizado de máquina.

Derivação da Regra de Aprendizado I

O Critério do Perceptron

A regra de atualização não é arbitrária. Ela pode ser derivada como uma forma de descida de gradiente estocástica em uma função de custo específica, o *Critério do Perceptron*.

Para simplificar a matemática, usamos alvos $y_n \in \{-1, 1\}$. Um ponto x_n é classificado incorretamente se o sinal da saída linear for diferente do sinal do alvo, ou seja, se $y_n(w^T x_n) < 0$.

O custo é a soma sobre todos os pontos mal classificados (\mathcal{M}):

$$J_P(w) = \sum_{n \in \mathcal{M}} -y_n(w^T x_n)$$

Nosso objetivo é minimizar este custo.

Derivação da Regra de Aprendizado II

Descida de Gradiente Estocástica

Calculamos o gradiente para um *único* ponto mal classificado:

$$\nabla_{\mathbf{w}} \left(-y_n (\mathbf{w}^T \mathbf{x}_n) \right) = -y_n \nabla_{\mathbf{w}} (\mathbf{w}^T \mathbf{x}_n) = -y_n \mathbf{x}_n$$

A regra de atualização da descida de gradiente é:

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} J_P = \mathbf{w}^{(t)} + \eta y_n \mathbf{x}_n$$

Esta é a regra de aprendizado do Perceptron para $y_n \in \{-1, 1\}$.

Derivação da Regra de Aprendizado III

Teorema da Convergência do Perceptron

Se o conjunto de treinamento for **linearmente separável**, o algoritmo do Perceptron tem a garantia de encontrar um hiperplano que separa os dados em um número finito de passos.

O Problema do XOR I

Dados Não-Linearmente Separáveis

Um Perceptron define uma fronteira de decisão linear (um hiperplano). Eles só funcionam se as classes puderem ser separadas por esta fronteira.

O Problema do XOR II

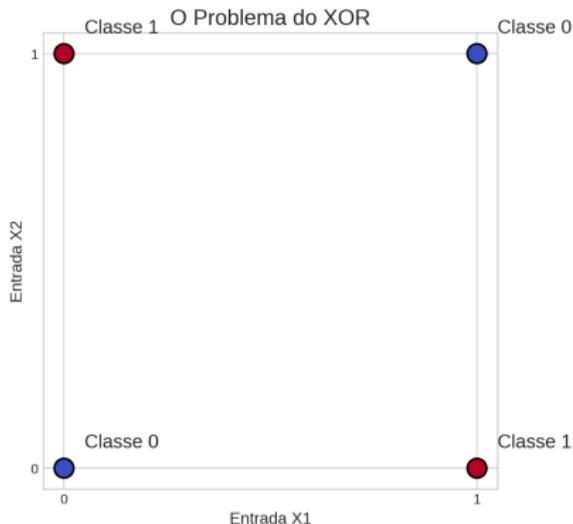


Figura: É impossível desenhar uma única linha reta para separar os pontos azuis dos vermelhos.

O Problema do XOR III

Laboratório Interativo no Google Colab

Exemplo mostrando o treinamento do Perceptron para dados linearmente e não-linearmente separáveis.

[Abrir Notebook no Colab](#)

A Controvérsia do Perceptron

Minsky e Papert, em seu livro de 1969, não apenas provaram a limitação do Perceptron simples, mas também fizeram um campanha de convencimento argumentando (de forma equivocada) que essas limitações se estenderiam a redes multicamadas, contribuindo para o primeiro “Inverno da IA”.

O Problema do XOR IV

Leitura Adicional

Para uma análise moderna e detalhada da controvérsia e seu impacto histórico:

Yuxi Liu (jan. de 2024). *The Perceptron Controversy*. Website.

URL:

<https://yuxi.ml/essays/posts/perceptron-controversy/>
(acesso em 10/09/2025)

Minsky e Rosenblatt eram conhecidos do ensino médio, que também aparentemente alimentou uma rivalidade e traz certas questões das motivações de Minsky.

Anatomia de um Modelo de Camada Única I

Estrutura Geral

Um modelo de camada única (como o Perceptron ou GLMs) segue uma estrutura comum:

- 1 Entradas:** Podem ser os dados brutos x ou um mapeamento para um espaço de features $\phi(x)$.
- 2 Combinação Linear:** Calcula-se uma soma ponderada das entradas. $z = w^T \phi(x) + b$
- 3 Função de Ativação:** Uma função (geralmente não-linear) σ é aplicada para produzir a saída. $\hat{y} = \sigma(z)$
- 4 Função de Custo (Loss):** $J(w, b)$ que mede a perda ou custo para cada valor dos parâmetros da rede.

Anatomia de um Modelo de Camada Única II

Modelagem

A escolha da função de ativação σ e da função de custo J depende da tarefa (regressão ou classificação).

Exemplo: Regressão Linear

O Modelo

- **Tarefa:** Prever um valor contínuo.
- **Entradas:** Usamos um mapeamento de features $\phi(x)$ (pode ser a identidade, $\phi(x) = x$, ou features polinomiais, etc.).
- **Ativação:** A função identidade, $\sigma(z) = z$.

$$\hat{y} = \mathbf{w}^T \phi(x) + b$$

- **Função de Custo:** Minimizamos o Erro da Soma dos Quadrados (derivado do MLE Gaussiano).

$$J(\mathbf{w}, b) = \sum_{n=1}^N (y_n - \hat{y}_n)^2$$

Classificação com Múltiplas Classes ($K > 2$)

Estrutura para K Classes

- **Saídas:** O modelo tem K saídas, uma para cada classe. Cada saída z_k é calculada por seu próprio vetor de pesos w_k .

$$z_k = w_k^T x + b_k$$

- **Codificação dos Alvos:** Os rótulos de classe y_n são convertidos para o formato **one-hot encoding**. Ex: para a classe 2 de 5, o alvo é $[0, 1, 0, 0, 0]$.
- **Ativação:** A função **Softmax**, que transforma as saídas lineares z_k em uma distribuição de probabilidade.

$$\hat{y}_k = \text{softmax}(z)_k = \frac{\exp(z_k)}{\sum_{j=1}^K \exp(z_j)}$$

Fronteiras de Decisão e Teoria da Decisão I

Fronteiras de Decisão Lineares

Para um classificador com saídas lineares (antes do softmax), a fronteira de decisão entre duas classes, C_i e C_j , é o conjunto de pontos onde suas ativações são iguais:

$$z_i(x) = z_j(x) \implies (w_i - w_j)^T x + (b_i - b_j) = 0$$

Esta é a equação de um hiperplano. Portanto, as fronteiras de decisão de um modelo linear são sempre lineares.

Fronteiras de Decisão e Teoria da Decisão II

Teoria da Decisão Bayesiana Mínima

A regra de decisão ótima é atribuir um ponto x à classe que maximiza a probabilidade posterior $p(C_k|x)$.

Nosso modelo, ao usar a função softmax e ser treinado com a cross-entropy, aprende a aproximar essas probabilidades posteriores.

A regra de decisão se torna:

Escolher classe k se $\hat{y}_k > \hat{y}_j$ para todo $j \neq k$

O Trade-off Viés-Variância I

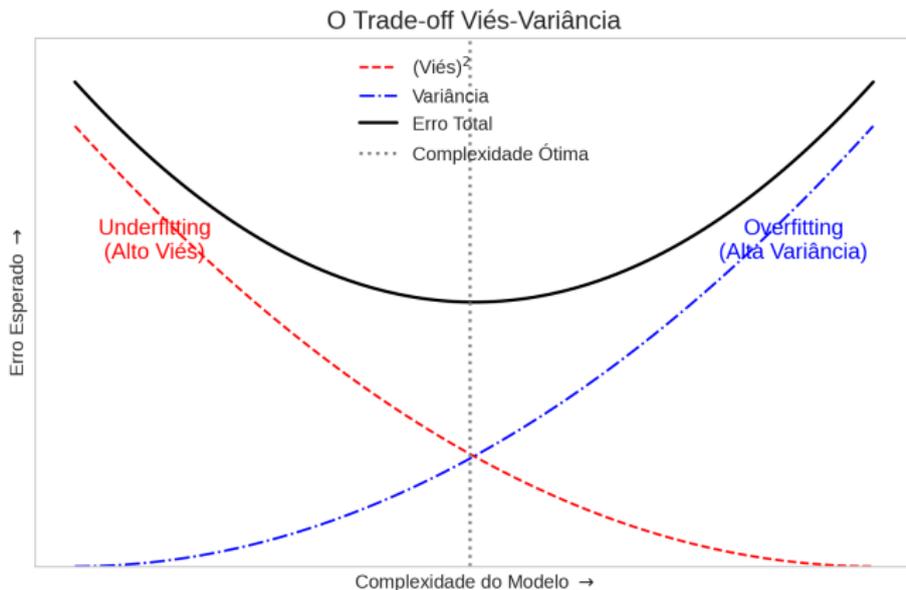
Decompondo o Erro de Generalização

O erro esperado de um modelo em dados não vistos pode ser decomposto em três componentes:

$$\text{Erro Esperado} = (\text{Viés})^2 + \text{Variância} + \text{Ruído Irreduzível}$$

- **Viés (Bias):** Erro devido a suposições simplistas do modelo. Um modelo com alto viés não consegue capturar a complexidade dos dados (underfitting).
- **Variância (Variance):** Erro devido à sensibilidade do modelo a flutuações nos dados de treino. Um modelo com alta variância se ajusta demais ao ruído (overfitting).

O Trade-off Viés-Variância II



O Trade-off Viés-Variância III

Figura: Ilustração do trade-off. A complexidade do modelo (e.g., grau do polinômio) controla o balanço.

O Que é um Modelo Linear Generalizado (GLM)?

GLMs são uma família de modelos estatísticos que generalizam a regressão linear – modelam uma entrada linear a uma saída com diferentes tipos de distribuições.

Componentes de uma GLM

Os Três Componentes de um GLM

- 1 Componente Aleatório:** Especifica a distribuição de probabilidade da variável de saída y (e.g., Gaussiana, Bernoulli, Poisson).

Componentes de uma GLM

Os Três Componentes de um GLM

- 1 Componente Aleatório:** Especifica a distribuição de probabilidade da variável de saída y (e.g., Gaussiana, Bernoulli, Poisson).
- 2 Componente Sistemático:** Um preditor linear que é uma combinação das variáveis de entrada.

$$\eta = \mathbf{w}^T \mathbf{x} + b$$

Componentes de uma GLM

Os Três Componentes de um GLM

- 1 Componente Aleatório:** Especifica a distribuição de probabilidade da variável de saída y (e.g., Gaussiana, Bernoulli, Poisson).
- 2 Componente Sistemático:** Um preditor linear que é uma combinação das variáveis de entrada.

$$\eta = \mathbf{w}^T \mathbf{x} + b$$

- 3 Função de Ligação (Link Function):** Uma função g que conecta a média do componente aleatório ($\mu = \mathbb{E}[y]$) com o componente sistemático $g(\mu) = \eta$.

Exemplo: Regressão Logística como um GLM

Construindo a Regressão Logística passo a passo

- **Componente Aleatório:** A saída y é binária (0 ou 1), então assumimos uma distribuição de **Bernoulli**. A média é a probabilidade de sucesso, $\mathbb{E}[y] = p$.

Exemplo: Regressão Logística como um GLM

Construindo a Regressão Logística passo a passo

- **Componente Aleatório:** A saída y é binária (0 ou 1), então assumimos uma distribuição de **Bernoulli**. A média é a probabilidade de sucesso, $\mathbb{E}[y] = p$.
- **Componente Sistemático:** O preditor linear, $\eta = w^T x + b$.

Exemplo: Regressão Logística como um GLM

Construindo a Regressão Logística passo a passo

- **Componente Aleatório:** A saída y é binária (0 ou 1), então assumimos uma distribuição de **Bernoulli**. A média é a probabilidade de sucesso, $\mathbb{E}[y] = p$.
- **Componente Sistemático:** O preditor linear, $\eta = w^T x + b$.
- **Função de Ligação:** A função de ligação canônica para a distribuição de Bernoulli é a função **Logit**.

$$g(p) = \ln \left(\frac{p}{1-p} \right) = \eta$$

Conexão com redes neurais

Da Função de Ligação para a Função de Ativação

Para obter a probabilidade p , precisamos da **inversa** da função de ligação:

$$p = g^{-1}(\eta) = \frac{1}{1 + e^{-\eta}} = \sigma(\eta)$$

Esta é a **função Sigmóide**, que serve como a "função de ativação" na Regressão Logística.

Perceptron vs. Regressão Logística (GLM) I

Perceptron

- **Saída:** Decisão “dura” (0 ou 1).
- **Ativação:** Função degrau (não-diferenciável).
- **Custo:** Critério do Perceptron (focado em erros de classificação).
- **Treino:** Descida de Gradiente.

Regressão Logística

- **Saída:** Probabilidade (valor contínuo entre 0 e 1).
- **Ativação:** Sigmóide (suave e diferenciável).
- **Custo:** Negativo do Log-Likelihood (Cross-Entropy).
- **Treino:** Descida de Gradiente.

Conexão com a Família Exponencial de Distribuições I

Por que os GLMs são tão especiais?

Muitas das distribuições mais comuns (Gaussiana, Bernoulli, Poisson, Categórica, etc.) pertencem a uma classe chamada **Família Exponencial**.

Uma distribuição pertence a esta família se sua função de probabilidade pode ser escrita na forma:

$$p(y|\eta) = h(y) \exp(\eta^T u(y) - a(\eta))$$

- η : o “parâmetro natural” da distribuição.
- $u(y)$: a “estatística suficiente”.

Conexão com a Família Exponencial de Distribuições II

A Conexão com GLMs

Em um GLM, se a distribuição de y pertence à família exponencial, a **função de ligação canônica** é aquela que mapeia a média μ para o parâmetro natural η .

Essa conexão garante belas propriedades matemáticas (como a convexidade da função de log-likelihood negativo), o que torna a otimização muito mais bem comportada.

Um Catálogo de Funções de Ativação

Degrau (Step):

$\sigma(z) = 1[z \geq 0]$ Usada no Perceptron original.

Linear (Identidade): $\sigma(z) = z$

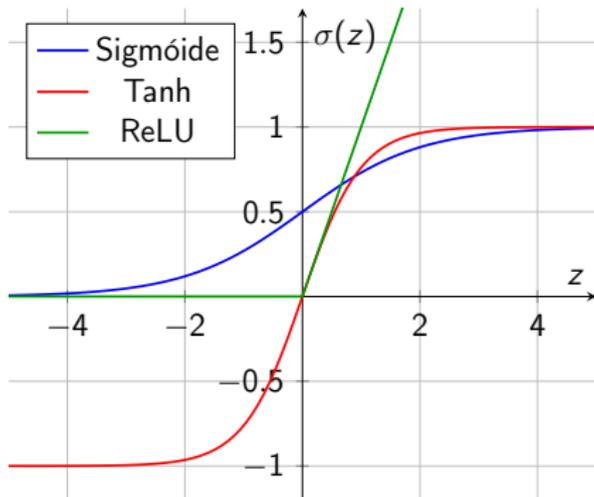
Sigmóide (Logistic):

$\sigma(z) = \frac{1}{1+e^{-z}}$ Boa para probabilidades, mas sofre do problema de saturação dos gradientes.

Tangente Hiperbólica (Tanh):

$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
 Similar à sigmóide, mas no intervalo $(-1, 1)$. É zero-centrada, o que pode ajudar na otimização.

Ativações Clássicas e Modernas



ReLU (Rectified Linear Unit):

$\sigma(z) = \max(0, z)$ Extremamente eficiente computacionalmente e não satura para valores positivos. É a ativação padrão para camadas ocultas.

Conectando GLMs e Redes Neurais

Função de Ligação vs. Função de Ativação

A tabela abaixo resume a conexão conceitual entre o framework estatístico dos GLMs e as escolhas de design em uma rede neural de camada única.

Tarefa	Distribuição da Saída (y)	Função de Ligação ($g(\mu) = \eta$)	Função de Ativação ($\hat{y} = \sigma(\eta)$)
Regressão	Gaussiana	Identidade	Linear (Identidade)
Classificação Binária	Bernoulli	Logit	Sigmóide
Classificação Multiclasse	Catagórica	Logit Generalizado	Softmax
Contagem	Poisson	Log	Exponencial (e^η)

Tabela: A função de ativação é, frequentemente, a inversa da função de ligação canônica da distribuição assumida para os dados.

Treinamento via Descida de Gradiente I

A Intuição

Imagine que a função de custo $J(w)$ é uma paisagem montanhosa. Nosso objetivo é encontrar o ponto mais baixo (o mínimo).

- O **gradiente** $\nabla_w J(w)$ é um vetor que aponta na direção de *maior* subida.
- Para minimizar o custo, damos um pequeno passo na direção **oposta** ao gradiente.

Treinamento via Descida de Gradiente II

A Regra de Atualização Geral

A descida de gradiente é um processo iterativo para ajustar os pesos:

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}^{(t)})$$

- η (eta) é a **taxa de aprendizado**, que controla o tamanho do passo.
- Este algoritmo é a base para treinar quase todas as redes neurais.

Mergulho Raso: O Gradiente de um Neurônio I

Calculando o Gradiente com a Regra da Cadeia

Para treinar com descida de gradiente, precisamos de $\nabla_w J$. Vamos derivá-lo para um único neurônio e um único dado (x, y) .

A computação acontece em etapas:

- 1 Entrada linear: $z = w^T x + b$
- 2 Ativação: $\hat{y} = \sigma(z)$
- 3 Custo: $J(\hat{y}, y)$

Usando a regra da cadeia para a derivada em relação a um peso w_i :

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Mergulho Raso: O Gradiente de um Neurônio II

A Fórmula Geral do Gradiente

Cada termo tem uma interpretação clara:

- $\frac{\partial J}{\partial \hat{y}}$: O quanto o custo muda com a saída (sinal de erro).
- $\frac{\partial \hat{y}}{\partial z} = \sigma'(z)$: O quanto a ativação muda com a entrada linear.
- $\frac{\partial z}{\partial w_i} = x_i$: O quanto a entrada linear muda com o peso.

Juntando tudo, o gradiente para o vetor de pesos w é:

$$\nabla_w J = \underbrace{\frac{\partial J}{\partial \hat{y}} \sigma'(z)}_{\text{Sinal de erro propagado}} \cdot \mathbf{x}^T$$

Descida de Gradiente: Batch, Estocástico e Mini-Batch I

O gradiente “verdadeiro” é a média sobre todo o dataset.
Diferentes estratégias existem para estimá-lo.

Batch GD

Todos os dados de treino:

$$\nabla J = \frac{1}{N} \sum_{n=1}^N \nabla J_n$$

- **Pró:** Passos precisos.
- **Contra:** Muito caro para datasets grandes.

Estocástico (SGD)

Um único dado de treino por passo:

$$\nabla J \approx \nabla J_n$$

- **Pró:** Rápido, pode escapar de mínimos locais.
- **Contra:** Caminho de otimização errático.

Mini-Batch SGD

Pequeno lote (mini-batch):

$$\nabla J \approx \frac{1}{B} \sum_{n \in \mathcal{B}} \nabla J_n$$

- **Pró:** Rápido e se beneficia de GPUs.
- **Contra:** hiperparâmetro (tamanho do lote).

Descida de Gradiente: Batch, Estocástico e Mini-Batch II

O Padrão em Deep Learning

Quase todos os modelos de aprendizado profundo são treinados com alguma variante do **Mini-Batch Stochastic Gradient Descent**.

Batch GD vs. SGD: O Caminho para o Mínimo I

Comparativo

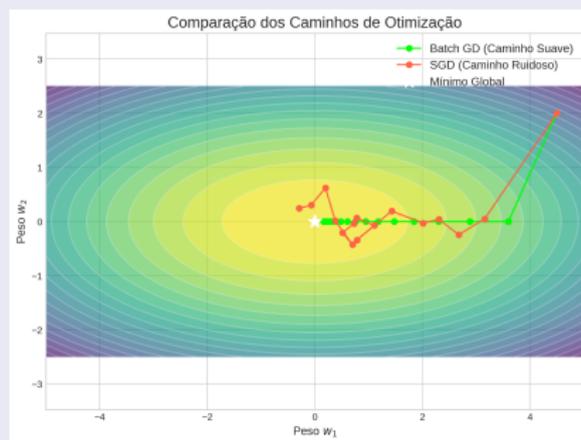


Figura: Comparação dos caminhos de otimização.

Batch GD vs. SGD: O Caminho para o Mínimo II

Batch Gradient Descent

O caminho é **suave e direto**. Como o gradiente é calculado sobre todo o dataset, cada passo é uma estimativa precisa da melhor direção para o mínimo. É como descer uma montanha em um dia claro.

Stochastic Gradient Descent

O caminho é **ruidoso e errático**. Cada passo é baseado em um único exemplo, então a direção pode não ser a ideal. É como descer a montanha em um nevoeiro denso.

- **Vantagem:** O ruído pode ajudar a escapar de mínimos locais rasos e a convergência é muito mais rápida em termos de dados processados.

O Papel da Taxa de Aprendizado (η) I

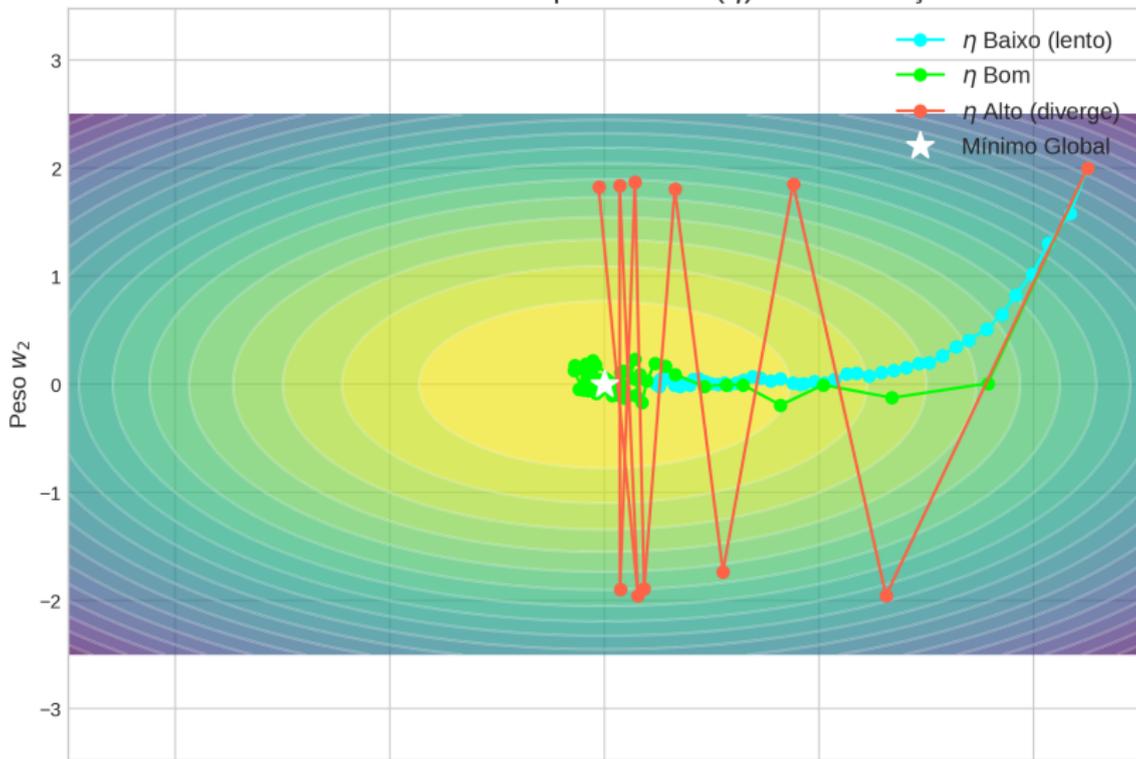
Controlando o Tamanho do Passo

A taxa de aprendizado, η , é um dos hiperparâmetros mais importantes. Ela determina o quão grande é o passo que damos na direção oposta ao gradiente.

- η **muito baixo**: Convergência muito lenta.
- η **muito alto**: Otimização pode oscilar, "pular" o mínimo e até divergir.
- η **ideal**: Convergência eficiente e estável.
- η **adaptativo**: Sequência de Robbins–Monro

O Papel da Taxa de Aprendizado (η) II

O Efeito da Taxa de Aprendizado (η) na Otimização



Indo Além do Aprendizado Supervisionado I

Uma Pergunta Central

Já vimos que um neurônio (ou uma camada de neurônios) pode separar dados. Mas será que essa mesma estrutura linear pode ser usada para aprender algo sobre a **estrutura intrínseca** dos dados?

Aprendizado de Representação

O objetivo é transformar os dados brutos x em uma nova representação h que seja mais útil, mais compacta ou que revele fatores importantes de variação dos dados.

$$h = f(x)$$

Aplicação: Remoção de Ruído (Denoising) I

Denoising Autoencoder

Uma variação poderosa é treinar o autoencoder para reconstruir uma versão *limpa* de uma entrada *ruidosa*.

- **Entrada:** $\tilde{x} = x_{limpo} + \text{ruído}$.
- **Alvo (Target):** x_{limpo} .
- **Objetivo:** Aprender a função $g(\tilde{x}) \approx x_{limpo}$.

Aplicação: Remoção de Ruído (Denoising) II

Anatomia de um Denoising Autoencoder



Figura: O modelo é forçado a aprender a estrutura essencial dos dados para conseguir remover o ruído e reconstruir a imagem original.

Outro Exemplo: Fatoração de Matrizes I

O Problema: Sistemas de Recomendação

O objetivo é prever entradas ausentes em uma matriz de interações R (e.g., notas que usuários deram para filmes).

A Solução: Fatoração

A ideia é decompor a matriz grande e esparsa R ($m \times n$) em duas matrizes menores e densas:

- P ($m \times k$): Uma matriz de **features latentes** para cada usuário.
- Q ($n \times k$): Uma matriz de **features latentes** para cada item.

O objetivo é que o produto delas aproxime a matriz original:
 $R \approx PQ^T$.

Outro Exemplo: Fatoração de Matrizes II

Conexão com Modelos Lineares

A predição para um único usuário u e item i é simplesmente o **produto escalar** de seus vetores de features latentes:

$$\hat{r}_{ui} = p_u^T q_i$$

Isto pode ser visto como um modelo linear onde as **features latentes (embeddings)** p_u e q_i são os pesos que o modelo aprende. É equivalente a uma rede neural rasa com duas camadas de embedding cujas saídas são combinadas por um produto escalar.

Anatomia de um Autoencoder I

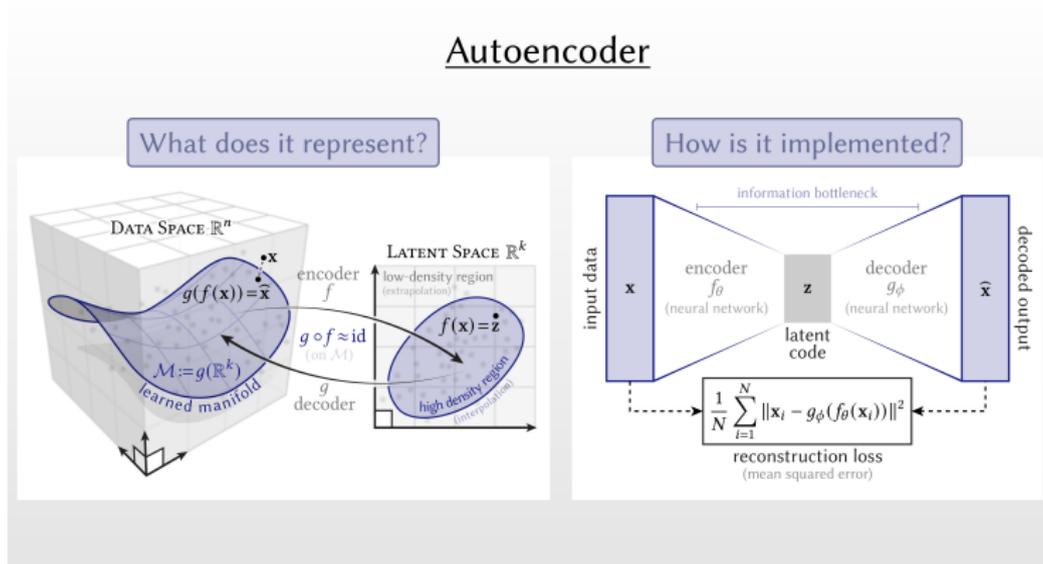


Figura: Fonte: Keenan Crane

Anatomia de um Autoencoder II

1. Encoder (f)

Mapeia a entrada de alta dimensão $x \in \mathbb{R}^n$ para uma **representação latente** (código) de baixa dimensão $z \in \mathbb{R}^k$.

O modelo é treinado para minimizar o **erro de reconstrução**, forçando a representação latente z a capturar a estrutura essencial dos dados.

2. Decoder (g)

Reconstrói a entrada original a partir do código latente, $\hat{x} = g(z)$.

PCA e o Autoencoder Linear I

O que acontece se o Encoder e o Decoder são lineares?

- **Encoder:** $z = W_{enc}x$
- **Decoder:** $\hat{x} = W_{dec}z = W_{dec}W_{enc}x$

Se treinarmos esta rede para minimizar o erro de reconstrução quadrático, $\sum_n \|x_n - \hat{x}_n\|^2$, o subespaço gerado pelas colunas de W_{dec} (ou pelas linhas de W_{enc}) converge para o mesmo subespaço dos **componentes principais (PCA)** dos dados.

PCA e o Autoencoder Linear II

PCA como Aprendizado de Features

A Análise de Componentes Principais (PCA) encontra as direções de maior variância nos dados. Um autoencoder linear aprende a projetar os dados nessas direções, mostrando que mesmo um modelo simples pode **aprender features lineares** ótimas para compressão.

Conexão: Autoencoder Linear, PCA e a Regra de Oja I

O Autoencoder Linear e seu Custo

Relembrando o autoencoder linear com pesos “amarrados”

($W_{dec} = W_{enc}^T = W$):

- **Encoder:** $h = Wx$
- **Decoder:** $\hat{x} = W^T h = W^T Wx$

O objetivo é minimizar o **erro de reconstrução quadrático médio**:

$$J(W) = \mathbb{E} \left[\|x - W^T Wx\|^2 \right]$$

Conexão: Autoencoder Linear, PCA e a Regra de Oja II

Minimizar o Erro \iff Maximizar a Variância (PCA)

É um resultado clássico da álgebra linear que minimizar o erro de reconstrução é equivalente a maximizar a variância da representação projetada h .

A solução para este problema é que as colunas de W devem formar uma base ortonormal para o subespaço gerado pelos k **componentes principais** dos dados (os autovetores da matriz de covariância com os maiores autovalores).

Conexão: Autoencoder Linear, PCA e a Regra de Oja III

A Conexão Final: Três Visões, Uma Solução

- **PCA (Estatística)**: Encontra a projeção linear ótima (os componentes principais) através da decomposição da matriz de covariância.
- **Autoencoder Linear (Otimização)**: Encontra o mesmo subespaço ao minimizar o erro de reconstrução via descida de gradiente.
- **Regra de Oja/GHA (Aprendizado Neural)**: É uma regra de aprendizado online e local que faz com que os pesos dos neurônios convirjam para esses mesmos componentes principais.

Conexão: Autoencoder Linear, PCA e a Regra de Oja IV

Isso nos mostra uma bela unificação de ideias: uma regra de aprendizado biologicamente inspirada resolve um problema de otimização de redes neurais que, por sua vez, é equivalente a um dos métodos mais fundamentais da estatística.

Laboratório Interativo no Google Colab

Exemplo mostrando a conexão entre esses três métodos no conjunto de dados MNIST.

[Abrir Notebook no Colab](#)

Qual “Custo” a Regra de Oja Minimiza? I

Maximização da Variância

Objetivo: Maximizar a variância da saída do neurônio.

$$\max_{\mathbf{w}} \mathbb{E}[y^2] = \max_{\mathbf{w}} \mathbb{E}[(\mathbf{w}^T \mathbf{x})^2] = \max_{\mathbf{w}} \mathbf{w}^T \Sigma \mathbf{w}$$

Onde $\Sigma = \mathbb{E}[\mathbf{x}\mathbf{x}^T]$ é a matriz de covariância dos dados.

Qual “Custo” a Regra de Oja Minimiza? II

A Necessidade de uma Restrição

Sem uma restrição, a solução para maximizar a variância seria trivial e inútil: $w \rightarrow \infty$. Para encontrar uma direção significativa, precisamos restringir a "energia" do neurônio. A restrição implícita é que a norma do vetor de pesos seja constante:

$$\|w\|^2 = 1$$

Este é exatamente o problema de encontrar o **primeiro componente principal** dos dados.

Mergulho Raso: Derivação Formal via Lagrangeanos I

O Problema de Otimização com Restrição

O objetivo da Regra de Oja é encontrar a direção de máxima variância nos dados. Matematicamente, isso se traduz em:

- **Maximizar:** A variância da saída, $J(\mathbf{w}) = \mathbb{E}[y^2] = \mathbf{w}^T \Sigma \mathbf{w}$, onde $\Sigma = \mathbb{E}[\mathbf{x}\mathbf{x}^T]$.
- **Sujeito a:** Uma restrição que impede os pesos de crescerem indefinidamente, $\|\mathbf{w}\|^2 = \mathbf{w}^T \mathbf{w} = 1$.

Mergulho Raso: Derivação Formal via Lagrangeanos II

Construindo o Lagrangeano

Usamos um multiplicador de Lagrange λ para incorporar a restrição à função objetivo:

$$\mathcal{L}(w, \lambda) = w^T \Sigma w - \lambda(w^T w - 1)$$

Mergulho Raso: Derivação Formal via Lagrangeanos III

Encontrando a Solução Ótima

Para encontrar o máximo, derivamos em relação a w e igualamos a zero:

$$\nabla_w \mathcal{L} = 2\Sigma w - 2\lambda w = 0 \implies \Sigma w = \lambda w$$

Esta é a **equação de autovetores!** A solução w deve ser um autovetor da matriz de covariância Σ . Para maximizar $J(w) = w^T(\lambda w) = \lambda$, devemos escolher o autovetor correspondente ao **maior autovalor**, que é, por definição, o **primeiro componente principal**.

Mergulho Raso: Da Solução Analítica à Regra de Oja I

Revisitando a Ascensão de Gradiente

Uma regra de atualização simples para maximizar a variância (sem restrição) seria a ascensão de gradiente estocástica, que leva à **regra de Hebb pura**:

$$w^{(t+1)} \leftarrow w^{(t)} + \eta(yx)$$

Sabemos que esta regra é instável.

Mergulho Raso: Da Solução Analítica à Regra de Oja II

A Conexão com a Regra de Oja

A regra de Oja pode ser vista como uma **aproximação estocástica** da ascensão de gradiente no Lagrangeano.

$$w^{(t+1)} \leftarrow w^{(t)} + \eta \left(\underbrace{yx}_{\approx \Sigma w} - \underbrace{y^2 w}_{\approx \lambda w} \right)$$

Mergulho Raso: Da Solução Analítica à Regra de Oja III

A Intuição

- O termo Hebbiano ($y\mathbf{x}$) é uma estimativa de um passo na direção do gradiente de $w^T \Sigma w$.
- O termo de esquecimento ($-y^2 w$) age como uma aproximação do termo do multiplicador de Lagrange. Como $y^2 = (w^T \mathbf{x})^2$ é uma estimativa da variância e a variância ótima é o autovalor λ , este termo efetivamente subtrai uma componente na direção de w , impedindo que sua norma cresça e forçando a convergência para o autovetor principal.

A Regra de Oja é uma forma elegante e computacionalmente simples de resolver um problema de otimização com restrição de forma online.

Encerramento e Perguntas I

Resumo da Aula

- Revisitamos o Perceptron clássico e sua regra de aprendizado.
- Conectamos modelos lineares ao framework estatístico dos GLMs.
- Comparamos Perceptron e Regressão Logística, motivando o uso de funções de custo suaves e da Descida de Gradiente.
- Vimos que modelos lineares podem ser usados para aprender representações úteis dos dados.

Encerramento e Perguntas II

Para a Próxima Aula

- **Tópico:** O Multilayer Perceptron (MLP) e Backpropagation.
- **Objetivo:** Entender como empilhar camadas para superar as limitações dos modelos lineares e aprender features não-lineares.

Perguntas?

Referências I

-  Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer.
-  Bishop, Christopher M. e Hugh Bishop (2023). *Deep Learning: Foundations and Concepts*. Springer.
-  Buchanan, Sam et al. (ago. de 2025). *Learning Deep Representations of Data Distributions*.
<https://ma-lab-berkeley.github.io/deep-representation-learning-book/>. Online.
-  Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". Em: *Mathematics of control, signals and systems* 2.4, pp. 303–314.

Referências II

-  Gefter, Amanda (fev. de 2015). “The Man Who Tried to Redeem the World with Logic”. Em: *Nautilus*. URL: <https://nautil.us/the-man-who-tried-to-redeem-the-world-with-logic-235253/>.
-  Grünwald, Peter D. (2007). *The Minimum Description Length Principle*. The MIT Press.
-  Jordan, Michael I (2019). “Artificial intelligence—the revolution hasn’t happened yet”. Em: *Harvard Data Science Review* 1.1.
-  Krause, Andreas e Jonas Hübner (2025). *Probabilistic Artificial Intelligence*. arXiv: 2502.05244 [cs.AI].
-  Liu, Yuxi (jan. de 2024). *The Perceptron Controversy*. Website. URL: <https://yuxi.ml/essays/posts/perceptron-controversy/> (acesso em 10/09/2025).

Referências III

-  McCulloch, Warren S. e Walter Pitts (1943). “A Logical Calculus of the Ideas Immanent in Nervous Activity”. Em: *The Bulletin of Mathematical Biophysics* 5.4, pp. 115–133. DOI: [10.1007/bf02478259](https://doi.org/10.1007/bf02478259).
-  Mercer, James (1909). “Functions of positive and negative type, and their connection with the theory of integral equations”. Em: *Philosophical transactions of the Royal Society of London. Series A* 209.441-458, pp. 415–446.
-  Minsky, Marvin e Seymour A Papert (1969). *Perceptrons: An introduction to computational geometry*. MIT press.
-  Murphy, Kevin Patrick (2023). *Probabilistic Machine Learning: Advanced Topics*. The MIT Press.

Referências IV

-  Oja, Erkki (1982). “A simplified neuron model as a principal component analyzer”. Em: *Journal of mathematical biology* 15.3, pp. 267–273.
-  Prince, Simon J.D. (2023). *Understanding Deep Learning*. The MIT Press.
-  Rasmussen, Carl Edward e Christopher K. I. Williams (2006). *Gaussian processes for machine learning*. The MIT Press.
-  Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.”. Em: *Psychological review* 65.6, p. 386.
-  Schölkopf, Bernhard e Alexander J Smola (2002). *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press.

Referências V

-  Shannon, C. E. (1988). “Programming a computer for playing chess”. Em: *Computer Chess Compendium*. Berlin, Heidelberg: Springer-Verlag, pp. 2–13. ISBN: 0387913319.
-  Shannon, Claude E. (1948). *A Mathematical Theory of Communication*. Vol. 27, pp. 379–423, 623–656.
-  Turing, Alan M. (1969). “Intelligent Machinery”. Em: *Machine Intelligence 5*. Ed. por Bernard Meltzer e Donald Michie. Edinburgh University Press, pp. 3–23.