Deep Learning

Funções de Ativação e Funções de Custo

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Visão Geral

1. **Funções de Ativação**
2. **Funções de Custo**
3. **Métricas de Avaliação**

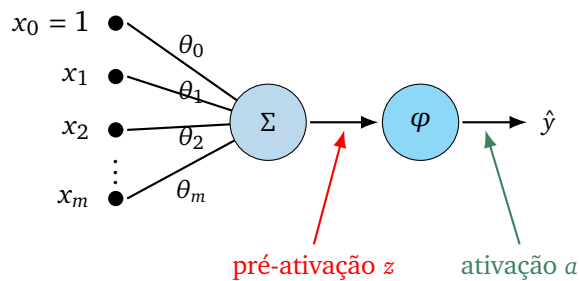
Visão Geral

1. **Funções de Ativação**
2. Funções de Custo
3. Métricas de Avaliação

Pré-ativação z vs ativação a

- Cada unidade computa uma soma ponderada das entradas (*pré-ativação*).
- O resultado passa por uma função φ (*ativação*) que determina a saída.
- Toda a riqueza de comportamento da rede depende da escolha de φ.

$$z = \boldsymbol{\theta}^\top \mathbf{x}, \qquad a = \varphi(z)$$



Camadas ocultas vs camada de saída

- Em **camadas ocultas**, φ adiciona não-linearidade: sem ela, qualquer rede colapsa em uma transformação afim.
- Em **camada de saída**, φ é escolhida conforme a tarefa (regressão, classificação binária, multiclasse, multilabel).
- As ativações de saída interagem diretamente com a função de custo.

O que esperamos de uma boa ativação

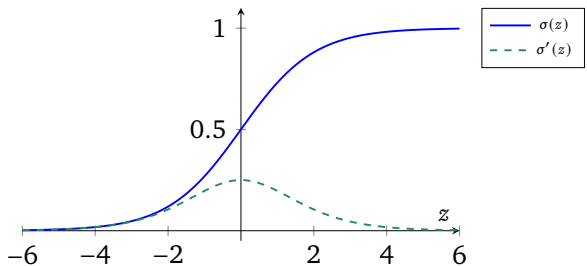
- Contínua e diferenciável (quase em todo lugar): a descida de gradiente exige φ'.
- $\varphi'(z)$ não nula em uma faixa ampla, para que o gradiente não se anule.
- Computação barata: avaliada milhões de vezes por iteração.
- Idealmente sem saturação severa: gradientes preservam ordem de magnitude entre camadas.

Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z))$$

- Imagem em $(0, 1)$.
- Derivada máxima 0.25 em $z = 0$.
- Histórica em redes rasas; saturação atrapalha em redes profundas.

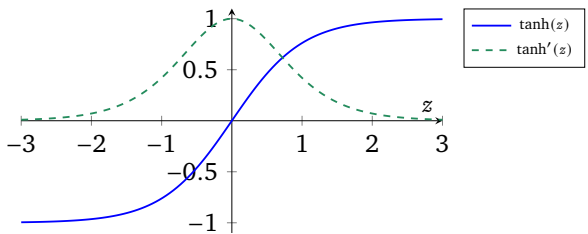


Tangente hiperbólica

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

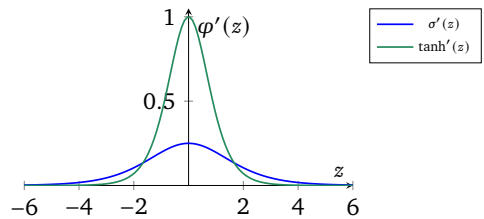
$$\tanh'(z) = 1 - \tanh^2(z)$$

- Imagem em $(-1, 1)$, centrada em zero.
- Mais simétrica que a sigmoid.
- Sofre do mesmo problema de saturação.



Saturação: dissipação de gradiente

- Para $|z|$ grande, $\sigma'(z) \to 0$ e $\tanh'(z) \to 0$.
- A regra da cadeia em uma rede profunda multiplica essas derivadas camada a camada.
- Resultado: gradientes vão a zero, camadas iniciais quase não aprendem.
- Esse fenômeno é o *vanishing gradient problem*.

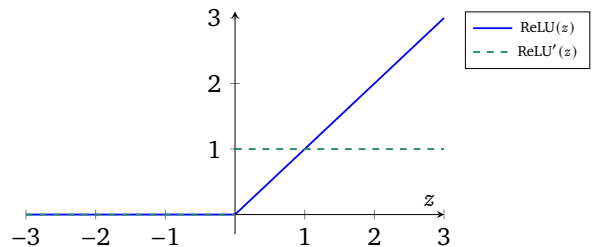


ReLU[1]

$$\mathrm{ReLU}(z) = \max(0, z)$$

$$\mathrm{ReLU}'(z) = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \end{cases}$$

- *Rectified Linear Unit*.
- Não satura no lado positivo: gradiente preservado.
- Esparsidade natural (zera o lado negativo).
- Padrão em redes profundas modernas.



[1] Vinod Nair e Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". Em: ICML. 2010, pp. 807–814.

O problema do *dying ReLU*

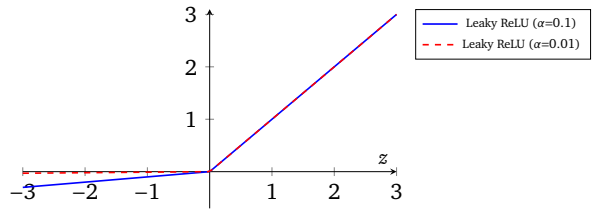
- Se uma unidade entra em $z < 0$ para todos os exemplos do dataset, $\text{ReLU}'(z) = 0$ sempre.
- Sem gradiente passando, os pesos não atualizam: a unidade fica morta.
- Em redes muito largas, é comum uma fração significativa das unidades estar morta em algum momento do treinamento.
- Causas usuais: taxa de aprendizado muito alta, inicialização ruim.

Leaky ReLU[2] e PReLU[3]

$$\varphi(z) = \max(\alpha z,\, z), \quad \alpha \in (0, 1)$$

$$\varphi'(z) = \begin{cases} \alpha & z < 0 \\ 1 & z > 0 \end{cases}$$

- Leaky ReLU: α fixo (tipicamente 0.01).
- PReLU: α é treinável.
- Mata o problema do *dying ReLU* ao manter gradiente α no lado negativo.



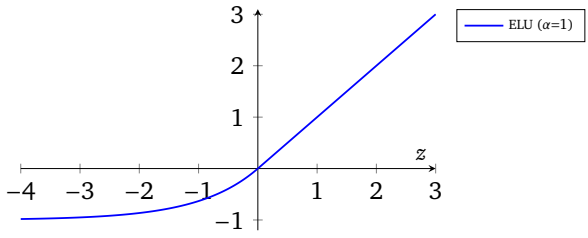
[2] Andrew L. Maas, Awni Y. Hannun e Andrew Y. Ng. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". Em: ICML Workshop on Deep Learning for Audio, Speech and Language Processing. 2013.

[3] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". Em: ICCV. 2015.

ELU[4]

$$\varphi(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}$$

$$\varphi'(z) = \begin{cases} 1 & z > 0 \\ \alpha\, e^z & z < 0 \end{cases}$$



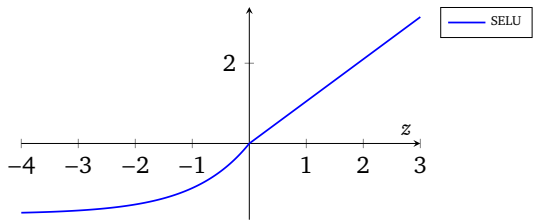
- *Exponential Linear Unit*.
- Suave em torno de 0; sai do regime de unidade morta exponencialmente.
- Empurra a média das ativações para perto de zero.

[4] Djork-Arné Clevert, Thomas Unterthiner e Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". Em: ICLR. 2016.

SELU[5]

$$\varphi(z) = \lambda \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}$$

- *Scaled ELU*: ELU multiplicada por uma constante λ.
- Constantes $\alpha \approx 1.6733$ e $\lambda \approx 1.0507$ são escolhidas para que a estatística das ativações se auto-normalize ao longo das camadas.
- Útil quando não usamos *batch norm*.

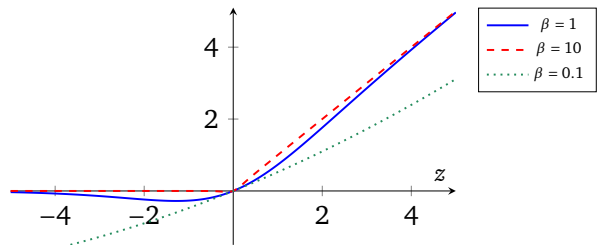


[5] Günter Klambauer et al. "Self-Normalizing Neural Networks". Em: NeurIPS. 2017.

Swish[6]

$$\varphi(z) = z\,\sigma(\beta z)$$

- Não monotônica: passa por um pequeno mínimo negativo antes de subir.
- Diferenciável em todo o domínio.
- Limite $\beta \to 0$: $\varphi(z) \approx z/2$ (linear).
- Limite $\beta \to \infty$: $\varphi(z) \to \mathrm{ReLU}(z)$.
- Com $\beta = 1$ é a **SiLU** (*Sigmoid Linear Unit*), `nn.SiLU` no PyTorch.
- Descoberta por busca automática de ativações.

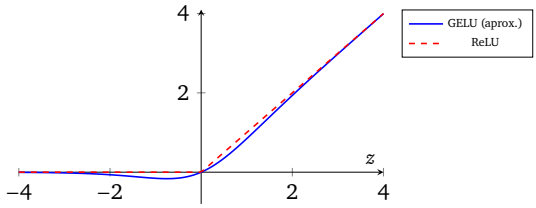


[6] Prajit Ramachandran, Barret Zoph e Quoc V. Le. “Searching for Activation Functions”. Em: arXiv preprint arXiv:1710.05941 (2017).

GELU[7]

$$\varphi(z) = z\,\Phi(z), \quad \Phi(z) = \Pr(Z \le z),\ Z \sim \mathcal{N}(0,1)$$

- *Gaussian Error Linear Unit*.
- Multiplica z pela probabilidade acumulada gaussiana $\Phi(z)$.
- Aproximação prática: $\varphi(z) \approx z\,\sigma(1.702\,z)$ (próxima de Swish).
- Padrão em *transformers* modernos (BERT, GPT-2/3, ViT).



[7] Dan Hendrycks e Kevin Gimpel. “Gaussian Error Linear Units (GELUs)”. Em: arXiv preprint arXiv:1606.08415 (2016).

SwiGLU[8]

- Não é uma ativação ponto-a-ponto: combina duas projeções lineares com gating.

$$\text{SwiGLU}(\mathbf{x}) = \text{Swish}_\beta(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{x}\mathbf{W}_2 + \mathbf{b}_2)$$

- $\odot$ é o produto elemento a elemento; uma das projeções faz *gating* sobre a outra.
- Substitui o bloco *feed-forward* típico em arquiteturas modernas como LLaMA, Mistral e PaLM.
- Empiricamente: melhora qualidade do modelo com mesmo orçamento de parâmetros e cálculo.

Custo

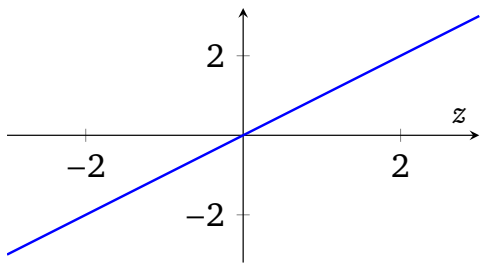
Adiciona uma terceira projeção ao bloco FFN (duas projeções de entrada em vez de uma), cerca de 50% mais parâmetros; na prática reduz-se a dimensão oculta para manter o orçamento.

[8] Noam Shazeer. "GLU Variants Improve Transformer". Em: arXiv preprint arXiv:2002.05202 (2020).

Identity (Linear)

$$\varphi(z) = z, \qquad \varphi'(z) = 1.$$

- Sem domínio limitado: $\varphi(z) \in \mathbb{R}$.
- Não adiciona não-linearidade: usar Linear em camadas ocultas faz a rede colapsar.
- Faz sentido apenas na **camada de saída**, em tarefas de regressão sem restrição de sinal.



Qual usar em uma camada oculta?

- **Default moderno**: ReLU para CNNs e MLPs simples; GELU para *transformers*.
- **Quando precisa de saída centrada em zero**: tanh em redes recorrentes clássicas.
- **Quando o *dying ReLU* aparece**: trocar para Leaky ReLU ou ELU.
- **Em LLMs grandes**: blocos FFN passam a usar SwiGLU.
- **Sigmoid em camada oculta**: praticamente nunca em redes profundas.

A camada de saída define a interface com a tarefa

- Regressão real: queremos $\hat{y} \in \mathbb{R}$ → **Linear**.
- Classificação binária: queremos $\hat{y} \in [0, 1]$ (probabilidade) → **Sigmoid**.
- Multilabel: K rótulos independentes, cada um com probabilidade própria → K **unidades Sigmoid**.
- Multiclasse exclusiva: K classes mutuamente exclusivas, distribuição categórica → **Softmax** (atua sobre a camada inteira).

Sigmoid na camada de saída

- Combinada com BCE (binary cross-entropy) na função de custo.
- Domínio do alvo: $y \in \{0, 1\}$.
- Saída: $\hat{y} = \sigma(z) = \Pr(y = 1 \mid x)$.
- Para multilabel basta uma sigmoid por classe; cada uma se comporta como um classificador binário independente.

Softmax: ativação que age na camada inteira

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Cada saída i depende de todas as pré-ativações, não só de z_i.
- Saídas são positivas e somam 1: distribuição de probabilidade.
- Combina com a *categorical cross-entropy* na loss.

z_i	$\text{softmax}(\mathbf{z})_i$
1.8	0.7500
0.7	0.2497
−6.0	0.0003

$$\text{softmax}\begin{pmatrix} 1.8 \\ 0.7 \\ -6.0 \end{pmatrix} \approx \begin{pmatrix} 0.7500 \\ 0.2497 \\ 0.0003 \end{pmatrix}$$

Visão Geral

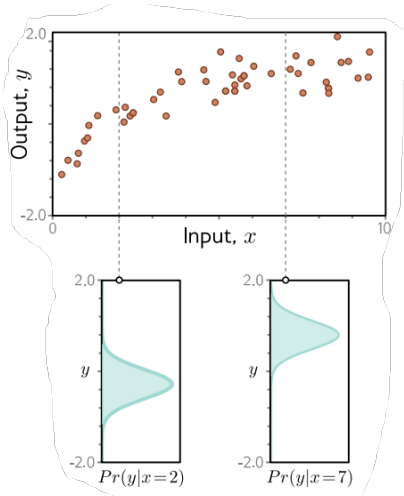
1. Funções de Ativação
2. **Funções de Custo**
3. Métricas de Avaliação

Mudança de paradigma

- Até aqui: a rede $f(\mathbf{x}; \boldsymbol{\theta})$ produz uma saída $\hat{y}$.
- A partir de agora: a rede produz parâmetros ϕ de uma distribuição de probabilidade sobre y.
- Loss vai medir o quão bem essa distribuição predita explica os dados observados.
- Esse é o ponto de partida da família de *maximum likelihood estimators*.

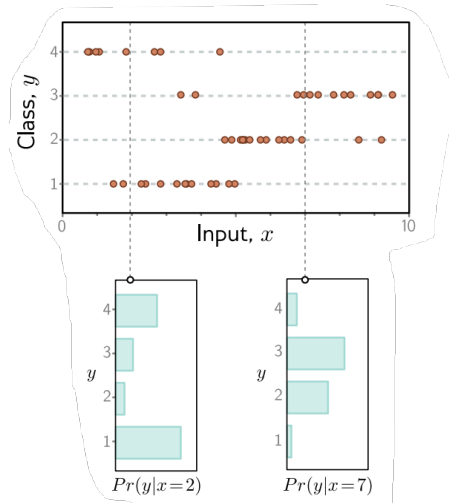
$\Pr(y \mid x)$ em uma tarefa contínua

- Tarefa de regressão: $y \in \mathbb{R}$.
- Para cada x, a rede estima a média de uma gaussiana sobre y.
- As gaussianas verticais à direita ilustram $\Pr(y \mid x{=}2)$ e $\Pr(y \mid x{=}7)$.



$\Pr(y \mid x)$ em uma tarefa discreta

- Tarefa de classificação multiclasse: $y \in \{1, 2, 3, 4\}$.
- Para cada x, a rede estima as probabilidades de cada classe.
- Distribuição categórica (Bernoulli generalizada).



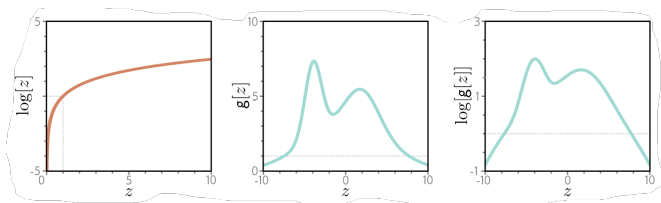
Estimador de máxima verossimilhança

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N} \Pr\left(y^{(i)} \mid f(x^{(i)}; \boldsymbol{\theta})\right)$$

- Maximiza a probabilidade conjunta dos rótulos observados, dada a entrada.
- Hipóteses implícitas:
  - As amostras $(x^{(i)}, y^{(i)})$ são **identicamente distribuídas**.
  - São **independentes** entre si (justifica o produto).
- Em uma frase: *i.i.d.*

Por que aplicar log à verossimilhança?

- log é monotônica crescente: preserva o ponto de máximo.
- Transforma produto em soma, que é estável numericamente.
- Soma de log-probabilidades ≤ 0, fácil de manipular.



Log-verossimilhança

$$
\begin{aligned}
\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N} \Pr\left(y^{(i)} \mid f(x^{(i)}; \boldsymbol{\theta})\right) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \log\left[\prod_{i=1}^{N} \Pr\left(y^{(i)} \mid f(x^{(i)}; \boldsymbol{\theta})\right)\right] \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \Pr\left(y^{(i)} \mid f(x^{(i)}; \boldsymbol{\theta})\right)
\end{aligned}
$$

Convenção: minimizar a *Negative Log-Likelihood*

$$
\begin{aligned}
\hat{\boldsymbol{\theta}} &= \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \Pr\left(y^{(i)} \mid f(x^{(i)}; \boldsymbol{\theta})\right) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} \left[- \sum_{i=1}^{N} \log \Pr\left(y^{(i)} \mid f(x^{(i)}; \boldsymbol{\theta})\right) \right] \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})
\end{aligned}
$$

- $\mathcal{L}(\boldsymbol{\theta})$ é a **Negative Log-Likelihood** (NLL).
- Diferentes escolhas de distribuição $\Pr(y \mid \cdot)$ produzem diferentes funções de custo conhecidas.

Receita geral para construir uma loss

1. Escolher uma distribuição $\Pr(y \mid \phi)$ apropriada ao domínio de y.
2. Fazer a rede $f(\mathbf{x}; \boldsymbol{\theta})$ predizer os parâmetros ϕ dessa distribuição.
3. Substituir e simplificar a NLL para obter uma loss $\mathcal{L}(\boldsymbol{\theta})$.
4. Treinar minimizando $\mathcal{L}$ via descida de gradiente.

O que vem agora

Vamos aplicar essa receita em três casos canônicos: regressão (gaussiana), classificação binária (Bernoulli) e classificação multiclasse (categórica).

Passo 1: escolher a distribuição

Para $y \in \mathbb{R}$ a escolha natural é a gaussiana univariada:

$$\Pr(y \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right].$$

- μ é a média (centro da distribuição).
- σ^2 é a variância (largura).

Passo 2: rede prediz a média

$$\Pr\big(y \mid f(\mathbf{x};\boldsymbol{\theta}), \sigma^2\big) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y - f(\mathbf{x};\boldsymbol{\theta}))^2}{2\sigma^2}\right].$$

- A rede passa a estimar $\mu = f(\mathbf{x};\boldsymbol{\theta})$.
- A variância σ^2 é tratada como hiperparâmetro (constante).
- A camada de saída é **Linear** (média pode assumir qualquer valor real).

Passo 3: substituir na NLL e simplificar

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}) &= -\sum_{i=1}^{N} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\,\exp\left(-\frac{(y^{(i)} - f(x^{(i)};\boldsymbol{\theta}))^2}{2\sigma^2}\right)\right] \\
&= -\sum_{i=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{i=1}^{N} \frac{(y^{(i)} - f(x^{(i)};\boldsymbol{\theta}))^2}{2\sigma^2} \\
&\propto \sum_{i=1}^{N} \left(y^{(i)} - f(x^{(i)};\boldsymbol{\theta})\right)^2
\end{aligned}
$$

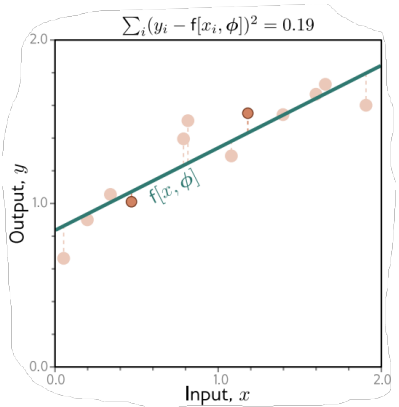
- O primeiro termo não depende de $\boldsymbol{\theta}$; a divisão por $2\sigma^2$ é constante.
- Sobra exatamente o erro quadrático.

Resultado: erro quadrático médio

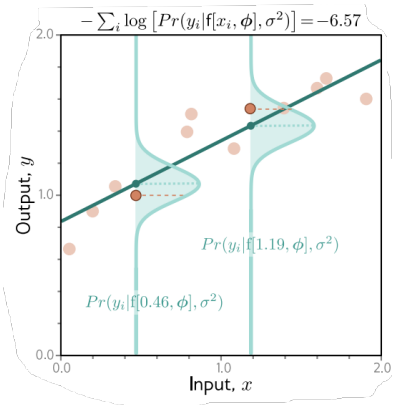
$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(y^{(i)} - f(x^{(i)}; \boldsymbol{\theta})\right)^2$$

- Esta é a L_2 **loss** (também chamada de MSE quando dividida por N).
- Sai naturalmente de duas hipóteses: instâncias *i.i.d.* amostradas de uma gaussiana univariada com variância fixa.

Visual: ajuste e gaussianas verticais



Soma dos quadrados dos resíduos.



$-\sum_i \log \Pr(y_i \mid f(x_i; \boldsymbol{\theta}), \sigma^2)$.

A mesma receita com outra distribuição: Laplace → MAE

Trocando a gaussiana pela distribuição de **Laplace**, com $\mu = f(\mathbf{x}; \boldsymbol{\theta})$ e escala b constante:

$$\Pr(y \mid \mu, b) = \frac{1}{2b} \exp\left(-\frac{|y - \mu|}{b}\right).$$

Substituindo na NLL:

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log\left[\frac{1}{2b} \exp\left(-\frac{|y^{(i)} - f(x^{(i)}; \boldsymbol{\theta})|}{b}\right)\right] \propto \sum_{i=1}^{N} \left|y^{(i)} - f(x^{(i)}; \boldsymbol{\theta})\right|$$

- O resultado é a L_1 **loss** (MAE): penaliza desvios linearmente, não quadraticamente.
- A Laplace tem caudas mais pesadas que a gaussiana: erros grandes são menos surpreendentes, logo a loss é mais robusta a outliers.

Passo 1: distribuição Bernoulli

Para $y \in \{0, 1\}$ usamos a Bernoulli(λ):

$$\Pr(y \mid \lambda) = \begin{cases} 1 - \lambda & y = 0 \\ \lambda & y = 1 \end{cases} = (1 - \lambda)^{1-y} \lambda^{y}.$$

- $\lambda \in [0, 1]$ é a probabilidade da classe positiva.
- Forma compacta usando expoentes evita o uso explícito de cases.

Passo 2: rede prediz λ via sigmoid

- Saída crua da rede $f(\mathbf{x}; \boldsymbol{\theta}) \in \mathbb{R}$ não respeita $\lambda \in [0, 1]$.
- Aplicamos sigmoid na saída: $\lambda = \sigma(f(\mathbf{x}; \boldsymbol{\theta}))$.

$$\Pr(y \mid \mathbf{x}; \boldsymbol{\theta}) = \big(1 - \sigma(f(\mathbf{x}; \boldsymbol{\theta}))\big)^{1-y} \big(\sigma(f(\mathbf{x}; \boldsymbol{\theta}))\big)^{y}.$$

Passo 3: NLL e simplificação

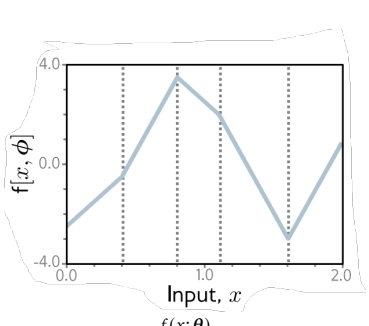
$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log\Big[\big(1-\sigma(f(x^{(i)};\boldsymbol{\theta}))\big)^{1-y^{(i)}}\ \big(\sigma(f(x^{(i)};\boldsymbol{\theta}))\big)^{y^{(i)}}\Big]$$

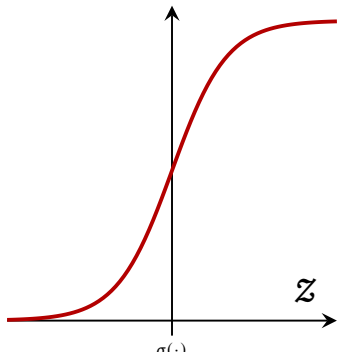
$$= \sum_{i=1}^{N}\Big[-\big(1-y^{(i)}\big)\log\big(1-\sigma(f(x^{(i)};\boldsymbol{\theta}))\big) - y^{(i)}\log\big(\sigma(f(x^{(i)};\boldsymbol{\theta}))\big)\Big]$$

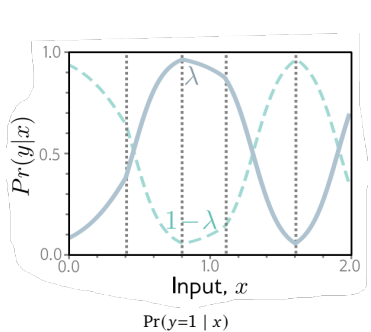
Resultado: Entropia Cruzada Binária (BCE)

$$\boxed{\mathcal{L}_{\text{BCE}}(\boldsymbol{\theta}) = -\sum_{i=1}^{N}\Big[y^{(i)}\log\hat{\lambda}^{(i)} + \big(1-y^{(i)}\big)\log\big(1-\hat{\lambda}^{(i)}\big)\Big]}$$

Visual: *piecewise* linear → sigmoid → $\Pr(y \mid x)$







- A pré-ativação da camada de saída é uma função *piecewise* linear.
- Após a sigmoid, a saída fica em $[0, 1]$ e oscila como uma probabilidade.

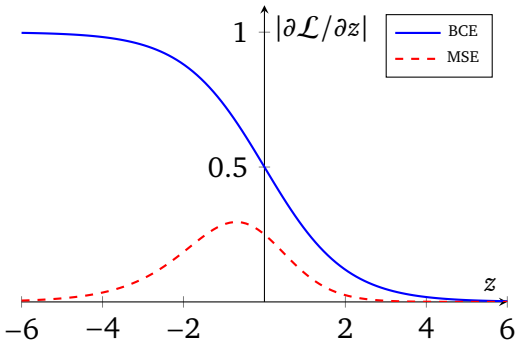
Por que não usar MSE em classificação?

Com saída $\hat{y} = \sigma(z)$:

$$\frac{\partial}{\partial z}\big(\sigma(z) - y\big)^2 = 2\big(\sigma(z) - y\big)\,\sigma'(z)$$

$$\frac{\partial \mathcal{L}_{\text{BCE}}}{\partial z} = \sigma(z) - y$$

- Na BCE, o σ' cancela: gradiente proporcional ao erro.
- No MSE, $\sigma'(z)$ sobrevive: se o modelo erra com confiança, o gradiente morre onde a correção é mais necessária.
- MSE embute a hipótese gaussiana; para $y \in \{0, 1\}$, a Bernoulli leva à BCE.



Magnitude do gradiente para um exemplo com $y = 1$.

Passo 1: distribuição categórica

Para $y \in \{1, 2, \ldots, K\}$ (exclusivo), a generalização da Bernoulli é a **categórica**:

$$\Pr(y \mid \boldsymbol{\lambda}) = \prod_{k=1}^{K} \lambda_k^{[y=k]} \quad \text{com} \quad \sum_{k=1}^{K} \lambda_k = 1,\ \lambda_k \geq 0.$$

- $[y = k]$ vale 1 se $y = k$ e 0 caso contrário (Iverson).
- Cada $\lambda_k \in [0, 1]$ e $\sum_k \lambda_k = 1$ formam um vetor de probabilidades.
- Outras notações comuns na literatura: π_k ou p_k no lugar de λ_k.

Passo 2: rede prediz λ via softmax

- Saída crua da rede: $\mathbf{z} = f(\mathbf{x}; \boldsymbol{\theta}) \in \mathbb{R}^K$ (logits).
- Aplicamos softmax para obter probabilidades:

$$\lambda_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}.$$

- Por construção, $\boldsymbol{\lambda}$ é um vetor de probabilidades válido.

Passo 3: NLL e simplificação

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= -\sum_{i=1}^{N} \log\left[\prod_{k=1}^{K} \lambda_k(\mathbf{x}^{(i)};\boldsymbol{\theta})^{[y^{(i)}=k]}\right] \\ &= -\sum_{i=1}^{N}\sum_{k=1}^{K} [y^{(i)}=k]\ \log \lambda_k(\mathbf{x}^{(i)};\boldsymbol{\theta}).\end{aligned}$$

Em forma vetorial com $\mathbf{y}^{(i)}$ *one-hot*:

$$\boxed{\mathcal{L}_{\text{CCE}}(\boldsymbol{\theta}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_k^{(i)}\ \log\big(\text{softmax}(f(\mathbf{x}^{(i)};\boldsymbol{\theta}))_k\big)}$$

Esta é a **Categorical Cross-Entropy** (CCE).

Forma equivalente: log-softmax

Como em cada amostra apenas a classe correta $k^* = y^{(i)}$ contribui (one-hot):

$$\mathcal{L}_{\text{CCE}}(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log\left[\frac{e^{z_{k^*}^{(i)}}}{\sum_{j=1}^{K} e^{z_j^{(i)}}}\right] = -\sum_{i=1}^{N}\left[z_{k^*}^{(i)} - \log\sum_{j=1}^{K} e^{z_j^{(i)}}\right].$$

- $\log\sum e^{z_j}$ é o *log-sum-exp*, calculado de forma estável em *frameworks*.

Derivada da softmax: preparação

Escrevendo $S = \sum_{j=1}^{K} e^{z_j}$, cada saída da softmax é um quociente:

$$\lambda_k = \frac{e^{z_k}}{S}.$$

- Como S soma todos os logits, λ_k depende de todos os z_j: a derivada completa é uma matriz (Jacobiano) $K \times K$ de termos $\partial \lambda_k / \partial z_j$.
- Ferramenta: regra do quociente, $\left(\frac{u}{v}\right)' = \frac{u'v - u\,v'}{v^2}$.
- Há duas situações distintas: derivar em relação ao **próprio** logit ($j = k$) ou em relação a **outro** logit ($j \neq k$).

Derivada da softmax: caso $j = k$

Com $u = e^{z_k}$ e $v = S$, temos $\dfrac{\partial u}{\partial z_k} = e^{z_k}$ e $\dfrac{\partial S}{\partial z_k} = e^{z_k}$:

$$\begin{aligned}\frac{\partial \lambda_k}{\partial z_k} &= \frac{e^{z_k} S - e^{z_k} e^{z_k}}{S^2} = \frac{e^{z_k}}{S} \cdot \frac{S - e^{z_k}}{S} \\ &= \lambda_k (1 - \lambda_k)\end{aligned}$$

- Mesma forma da derivada da sigmoid: aumentar z_k aumenta λ_k, com efeito máximo quando $\lambda_k = 1/2$.
- Satura quando $\lambda_k \to 0$ ou $\lambda_k \to 1$.

Derivada da softmax: caso $j \neq k$

Agora o numerador e^{z_k} não depende de z_j ($\partial u / \partial z_j = 0$), mas S sim ($\partial S / \partial z_j = e^{z_j}$):

$$\begin{aligned} \frac{\partial \lambda_k}{\partial z_j} &= \frac{0 \cdot S - e^{z_k}\, e^{z_j}}{S^2} = -\frac{e^{z_k}}{S} \cdot \frac{e^{z_j}}{S} \\ &= -\lambda_k\, \lambda_j \end{aligned}$$

- Sinal negativo: aumentar outro logit rouba massa de probabilidade de λ_k.
- Os dois casos se unificam com o colchete de Iverson (delta de Kronecker):

$$\boxed{\frac{\partial \lambda_k}{\partial z_j} = \lambda_k \big([k = j] - \lambda_j \big)}$$

Por que derivar a loss direto nos logits?

Caminho 1: regra da cadeia atravessando o Jacobiano da softmax, com $\frac{\partial \mathcal{L}}{\partial \lambda_k} = -\frac{y_k}{\lambda_k}$:

$$\frac{\partial \mathcal{L}}{\partial z_j} = \sum_{k=1}^{K} \left(-\frac{y_k}{\lambda_k}\right) \lambda_k \big([k=j] - \lambda_j\big) = -y_j + \lambda_j \underbrace{\sum_{k=1}^{K} y_k}_{=1} = \lambda_j - y_j.$$

Caminho 2: derivar a forma log-softmax do slide anterior, direto em z_j:

$$\frac{\partial}{\partial z_j} \left[-z_{k^*} + \log S\right] = -[j=k^*] + \frac{e^{z_j}}{S} = \lambda_j - y_j.$$

- Mesmo resultado, mas o caminho 2 dispensa o Jacobiano inteiro e evita a divisão por λ_k, que explode quando $\lambda_k \to 0$.
- O gradiente nos logits é simplesmente softmax menos one-hot: por isso *frameworks* calculam a cross-entropy direto sobre os logits.

nn.BCEWithLogitsLoss

- Recebe logits (pré-ativação).
- Aplica sigmoid + BCE em uma operação numericamente estável (*log-sum-exp trick*).
- Forma recomendada para classificação binária e multilabel.

```python
import torch
import torch.nn as nn

# Trabalha com logits (saída crua da rede, sem sigmoid)
# Aplica sigmoid + BCE em uma única operação numericamente estável.

logits = torch.tensor([2.5, -1.0, 0.3])
target = torch.tensor([1.0, 0.0, 1.0])

loss_fn = nn.BCEWithLogitsLoss()
loss = loss_fn(logits, target)

# Equivalente manual (menos estável):
# probs = torch.sigmoid(logits)
# loss = -(target * probs.log() + (1 - target) * (1 - probs).log()).mean()

print(loss.item())  # 0.5390...
```

nn.BCELoss

- Recebe probabilidades (saída já passada por sigmoid).
- Mais propenso a instabilidade numérica em logits extremos.
- Falha silenciosamente se a entrada não está em $(0, 1)$.

```python
import torch
import torch.nn as nn

# Trabalha com probabilidades (saída já passada por sigmoid).
# Numericamente menos estável; falha silenciosamente se a entrada
# não estiver em (0, 1).

probs = torch.sigmoid(torch.tensor([2.5, -1.0, 0.3]))
target = torch.tensor([1.0, 0.0, 1.0])

loss_fn = nn.BCELoss()
loss = loss_fn(probs, target)

# Para classificação multilabel, basta usar a mesma loss
# em saídas com K unidades sigmoide (uma por rótulo).

print(loss.item())  # 0.5390...
```

Multiclasse vs Multilabel

- **Multiclasse exclusiva**: o exemplo pertence a exatamente uma classe.
- **Multilabel**: o exemplo pode receber vários rótulos simultaneamente.

1. **Cachorro**
2. Gato
3. Rato
4. Humano

Multiclasse: única label correta

1. **Cachorro**
2. **Gato**
3. Rato
4. Humano

Multilabel: várias labels possíveis

Cheat sheet: tarefa → camada → loss

Tarefa	Saída esperada	Camada de saída	Função de custo
Regressão	$\hat{y} \in \mathbb{R}$	1 unidade Linear	MSE (L_2)
Classificação binária	$\hat{y} \in [0, 1]$	1 unidade Sigmoid	BCE
Multilabel (K rótulos)	$\hat{\mathbf{y}} \in [0, 1]^K$	K unidades Sigmoid	BCE somada
Multiclasse (K classes)	$\hat{\mathbf{y}} \in [0, 1]^K,\ \sum_k \hat{y}_k = 1$	Softmax (K logits)	CCE

- Em PyTorch: `BCEWithLogitsLoss` (binária e multilabel) e `CrossEntropyLoss` (multiclasse) já incorporam softmax/sigmoid.
- Para regressão com restrições (ex.: $\hat{y} \geq 0$), pode-se trocar a Linear por uma SoftPlus na saída e usar uma Poisson.

Visão Geral

1. Funções de Ativação
2. Funções de Custo
3. **Métricas de Avaliação**

Métricas ≠ função de custo

- A função de custo é otimizada durante o treinamento; precisa ser diferenciável e numericamente estável.
- A métrica é interpretável pelo usuário final; pode ser não diferenciável (acurácia, F1) ou dependente de um *threshold*.
- Quase sempre, a loss é uma aproximação suave do que de fato queremos otimizar (a métrica).

Métricas de regressão

- **MSE**: $\frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2$.
- **MAE**: $\frac{1}{N}\sum_{i=1}^{N}|\hat{y}^{(i)} - y^{(i)}|$.
- R^2: $1 - \frac{\sum_i (y^{(i)} - \hat{y}^{(i)})^2}{\sum_i (y^{(i)} - \bar{y})^2}$, onde $\bar{y}$ é a média dos rótulos.
- MSE pune grandes desvios mais fortemente; MAE é mais robusto a *outliers*; R^2 é interpretável como variância explicada.

Métricas de classificação

- **Acurácia**: $\frac{TP + TN}{TP + TN + FP + FN}$.
- **Recall**: $\frac{TP}{TP + FN}$.
- **Precision**: $\frac{TP}{TP + FP}$.
- F_β**-score**: $(1 + \beta^2)\frac{P \cdot R}{\beta^2 P + R}$.

		Predito	
		Sim	Não
Real	Sim	TP	FN
	Não	FP	TN

Métricas de classificação: exemplo numérico

Matriz de confusão

		Predito Sim	Não
Real	Sim	8	2
	Não	3	87

100 instâncias, 10 positivas.

$$\text{Acurácia} = \frac{8+87}{100} = 0.95$$

$$\text{Recall} = \frac{8}{8+2} = 0.80$$

$$\text{Precision} = \frac{8}{8+3} \approx 0.727$$

$$F_1 = 2 \cdot \frac{0.727 \cdot 0.80}{0.727+0.80} \approx 0.762$$

Acurácia alta esconde recall mais baixo: classe positiva rara é mais difícil.

Escolhendo a métrica certa

- **Datasets balanceados**: acurácia é razoável.
- **Datasets desbalanceados**: acurácia engana; prefira F1 ou recall específico.
- **Quando o custo do FN é alto** (medicina, fraude): priorize **recall**.
- **Quando o custo do FP é alto** (spam): priorize **precision**.
- **Multiclasse**: macro/micro-F1, matriz de confusão por classe.

Leituras Recomendadas

- Capítulos 5 e 6 de Simon J. D. Prince. Understanding Deep Learning. MIT Press, 2023
- Capítulo 6 de Ian Goodfellow, Yoshua Bengio e Aaron Courville. Deep Learning. MIT Press, 2016. URL: https://www.deeplearningbook.org/
- ReLU: Vinod Nair e Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". Em: ICML. 2010, pp. 807–814
- GELU: Dan Hendrycks e Kevin Gimpel. "Gaussian Error Linear Units (GELUs)". Em: arXiv preprint arXiv:1606.08415 (2016)
- Swish: Prajit Ramachandran, Barret Zoph e Quoc V. Le. "Searching for Activation Functions". Em: arXiv preprint arXiv:1710.05941 (2017)
- SwiGLU: Noam Shazeer. "GLU Variants Improve Transformer". Em: arXiv preprint arXiv:2002.05202 (2020)

Referências I

[1] Djork-Arné Clevert, Thomas Unterthiner e Sepp Hochreiter. “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”. Em: ICLR. 2016.

[2] Ian Goodfellow, Yoshua Bengio e Aaron Courville. Deep Learning. MIT Press, 2016. URL: https://www.deeplearningbook.org/.

[3] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. Em: ICCV. 2015.

[4] Dan Hendrycks e Kevin Gimpel. “Gaussian Error Linear Units (GELUs)”. Em: arXiv preprint arXiv:1606.08415 (2016).

[5] Günter Klambauer et al. “Self-Normalizing Neural Networks”. Em: NeurIPS. 2017.

[6] Andrew L. Maas, Awni Y. Hannun e Andrew Y. Ng. “Rectifier Nonlinearities Improve Neural Network Acoustic Models”. Em: ICML Workshop on Deep Learning for Audio, Speech and Language Processing. 2013.

[7] Vinod Nair e Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. Em: ICML. 2010, pp. 807–814.

[8] Simon J. D. Prince. Understanding Deep Learning. MIT Press, 2023.

[9] Prajit Ramachandran, Barret Zoph e Quoc V. Le. “Searching for Activation Functions”. Em: arXiv preprint arXiv:1710.05941 (2017).

[10] Noam Shazeer. “GLU Variants Improve Transformer”. Em: arXiv preprint arXiv:2002.05202 (2020).