Aprendizado Profundo 1

Funções de Ativação e Funções de Custo

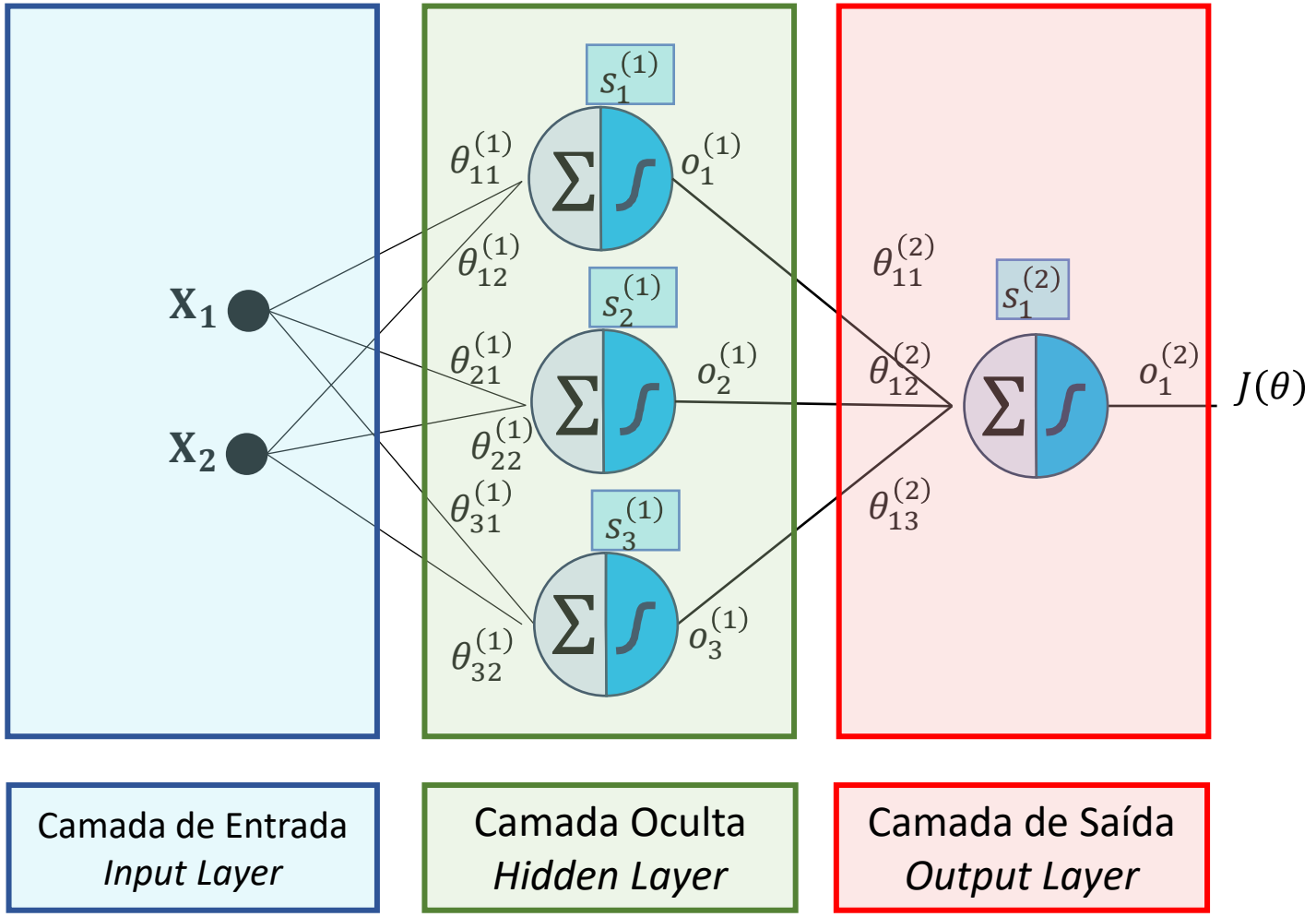
Professor: Lucas Silveira Kupssinskü

Agenda

- Funções de Ativação
  - Características Desejadas
  - Derivadas
  - Potenciais problemas
- Funções de Custo
  - Derivação a partir de máxima verossimilhança
  - Função de Custo vs Tarefa
  - Função de Custo vs Camada de Saída
- Métricas de Avaliação

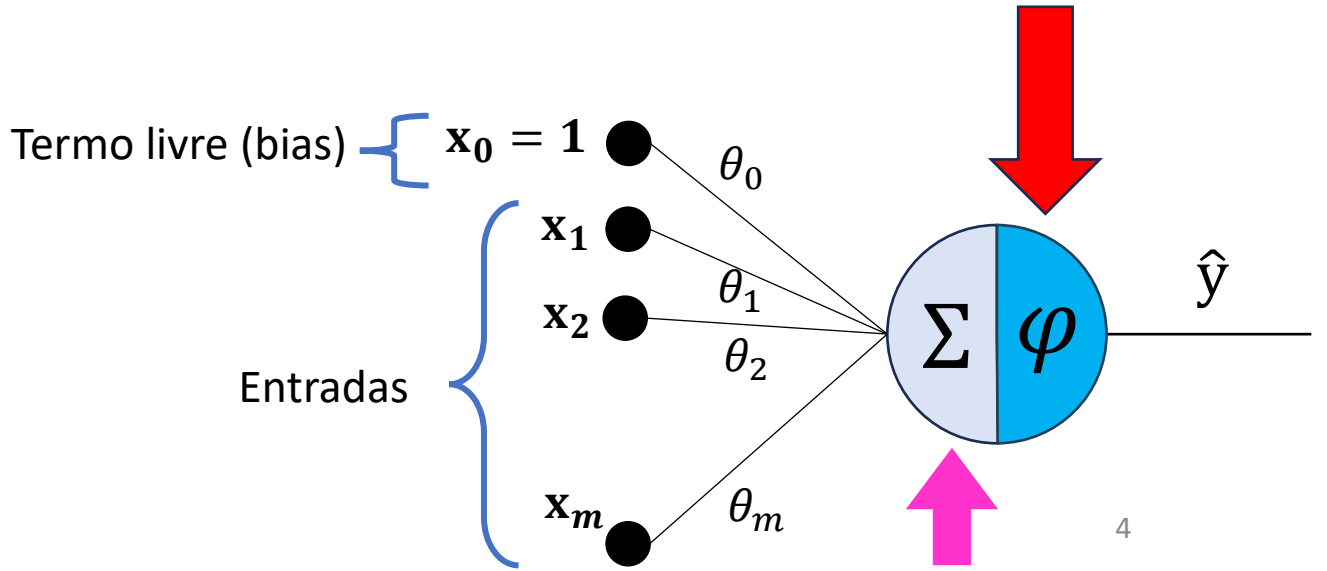
Perceptron Multicamadas

- As funções de ativação são colocadas em cada unidade e exercem função diferente dependendo da camada na qual estão posicionadas
- A função de custo é computada (usualmente) na última camada e guia o processo de otimização
- Usualmente a tarefa que vamos resolver restringe nossas escolhas de Loss e de ativação na última camada da rede
- As pré-ativações da última camada são chamadas de logits (principalmente em problemas de classificação)



Funções de Ativação

- São aplicadas na pré-ativação
- Nas camadas ocultas são responsáveis por adicionar “não linearidades”
- Na camada de saída, usualmente são escolhidas conforme a tarefa
- Idealmente devem ser:
  - contínuas
  - diferenciáveis

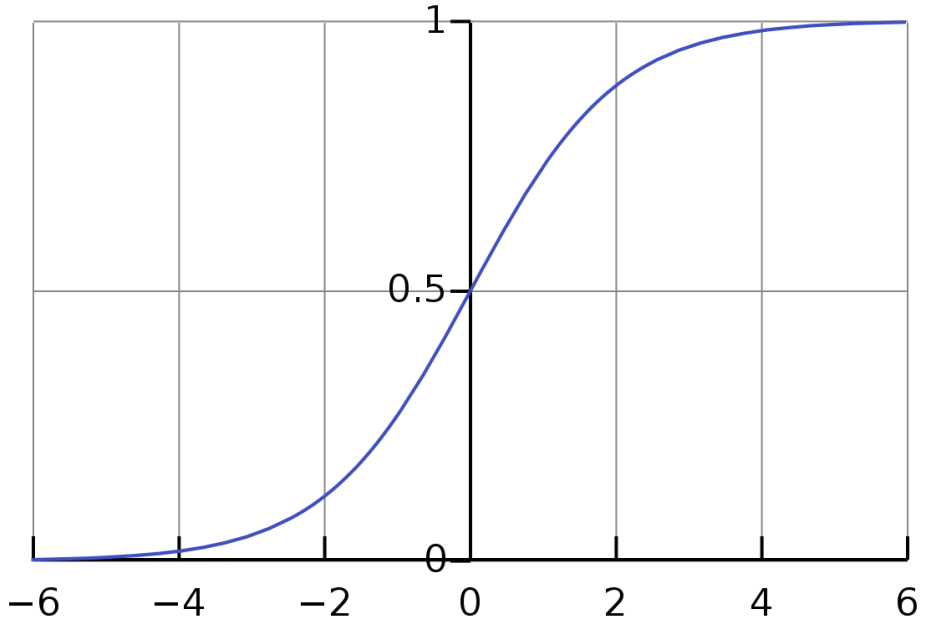


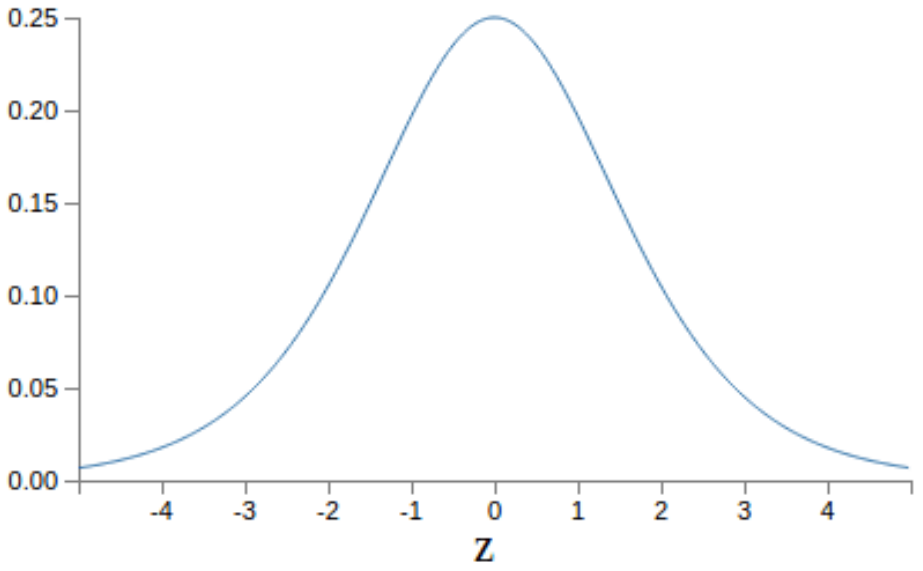
Funções de Ativação

- **Podem atuar individualmente em unidades**
- Podem atuar em todas as unidades da camada

Função *Sigmoid*

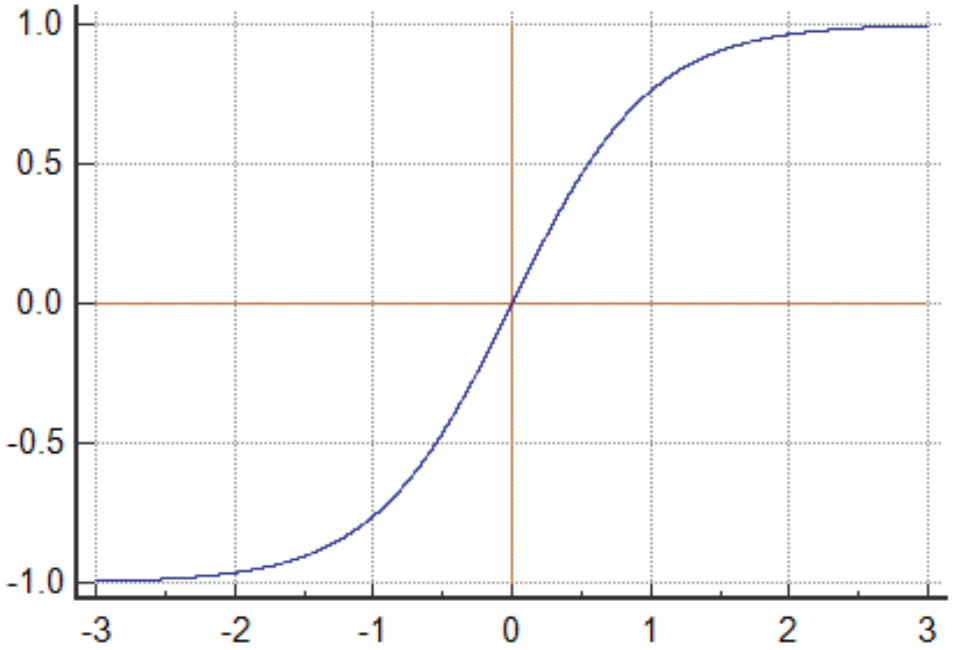
- $\varphi(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
- $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$

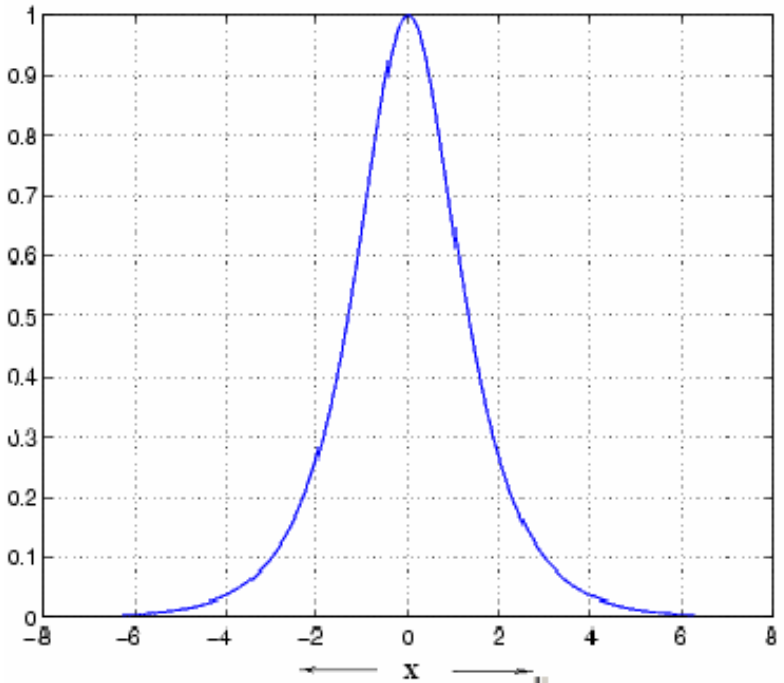




Função *tanh*

- $\varphi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$
- $\varphi'(z) = 1 - \big(\sigma(z)\big)^2$

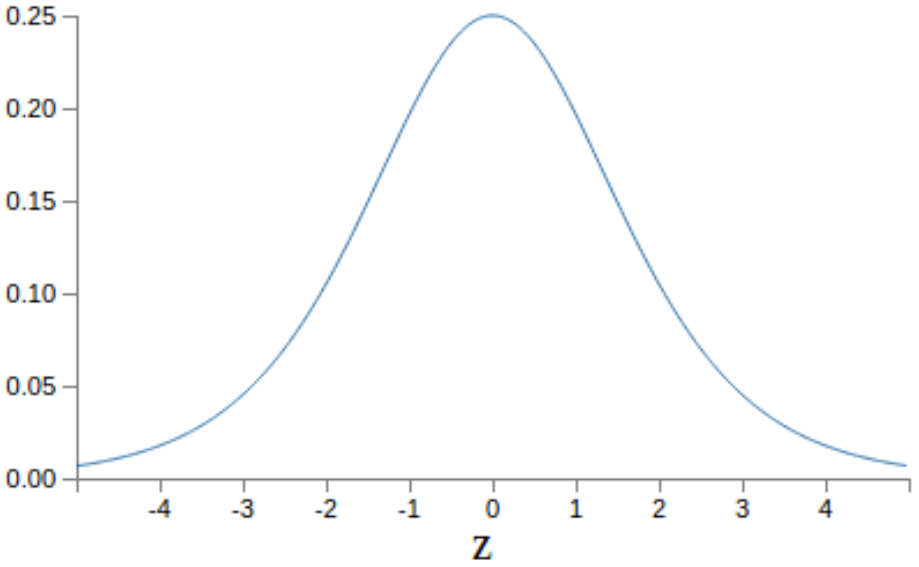




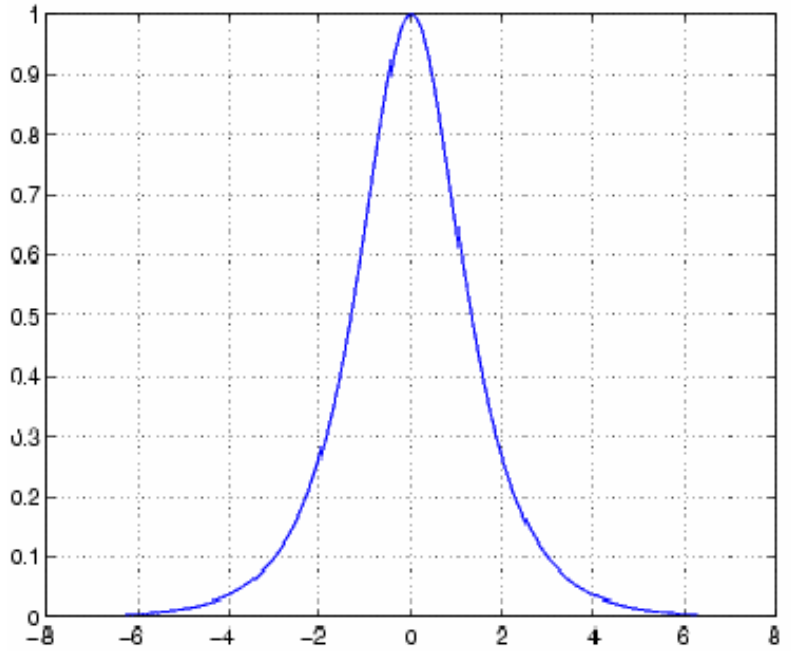
Problemas

- Você consegue pensar em um problema que essas derivadas podem trazer na descida de gradiente?

$\sigma'(z)$



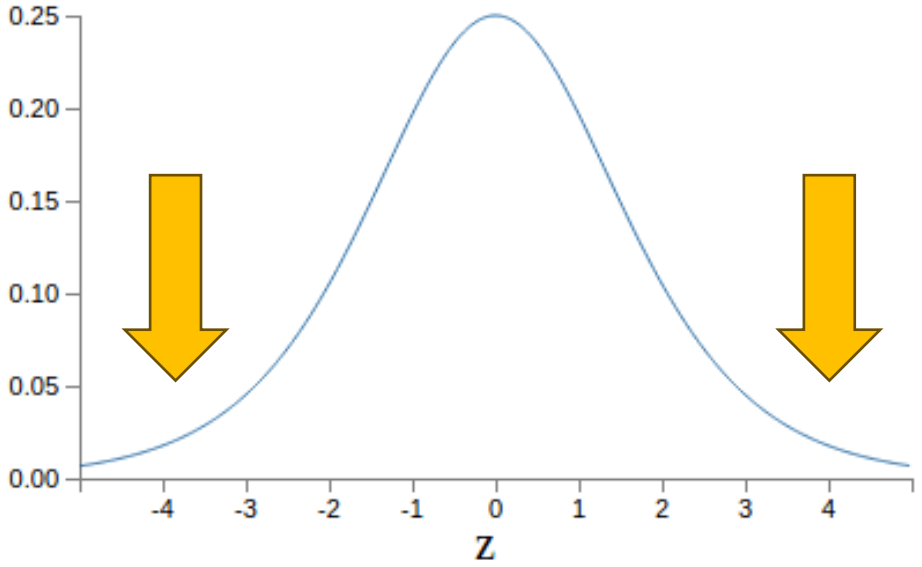
$\varphi'(z)$



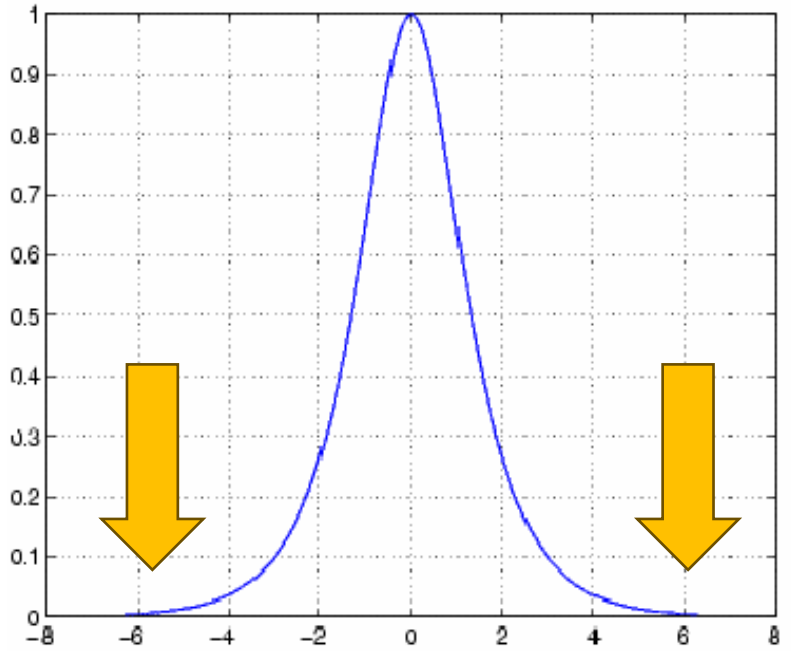
Problemas

- Os gradientes tendem a diminuir drasticamente conforme a entrada se afasta de 0.
- Dissipação de Gradiente

$\sigma'(z)$



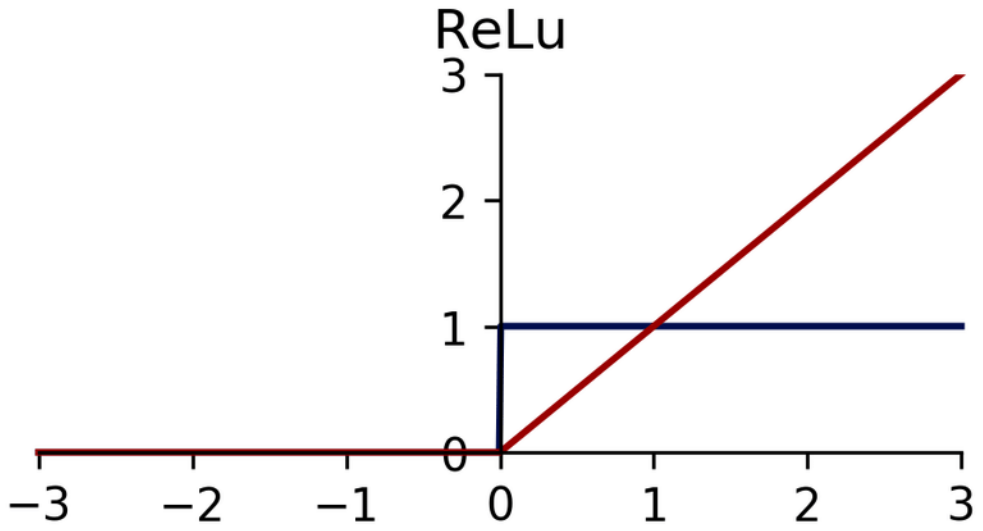
$\varphi'(z)$



ReLU

Rectified Linear Unit

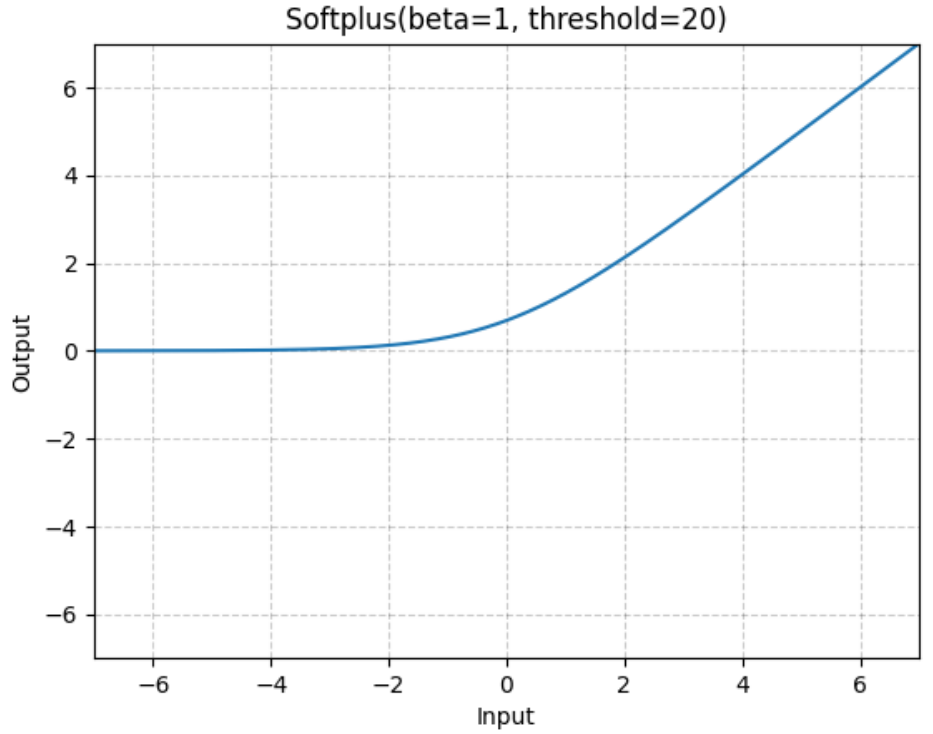
- $\varphi(x) = \max(0, x)$
- $\varphi'(x) = \begin{cases} 0, x \leq 0 \\ 1, x > 0 \end{cases}$
- Vantagens:
  - Não satura
  - Gradientes não dissipam
- Problemas:
  - Relu pode “morrer”

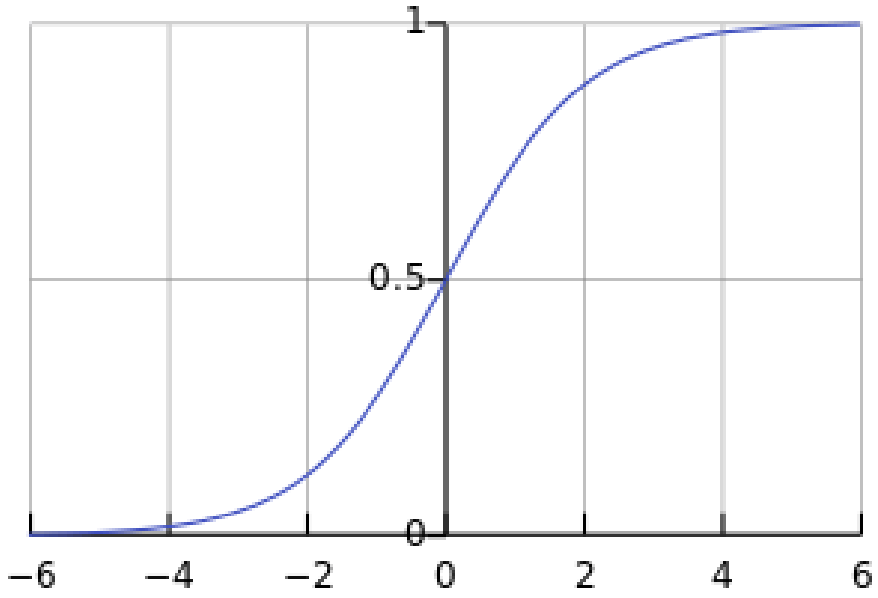


SoftPlus

- $\varphi(x) = \begin{cases} \frac{1}{\beta} \ln\left(1 + e^{\beta x}\right), x \leq k \\ x \qquad\qquad\qquad\quad , x > k \end{cases}$
- $\varphi'(x) = \sigma(x)$
  - Para $\beta = 1$
- k é um threshold, tipicamente 20

- ReLU porém mais “suave”

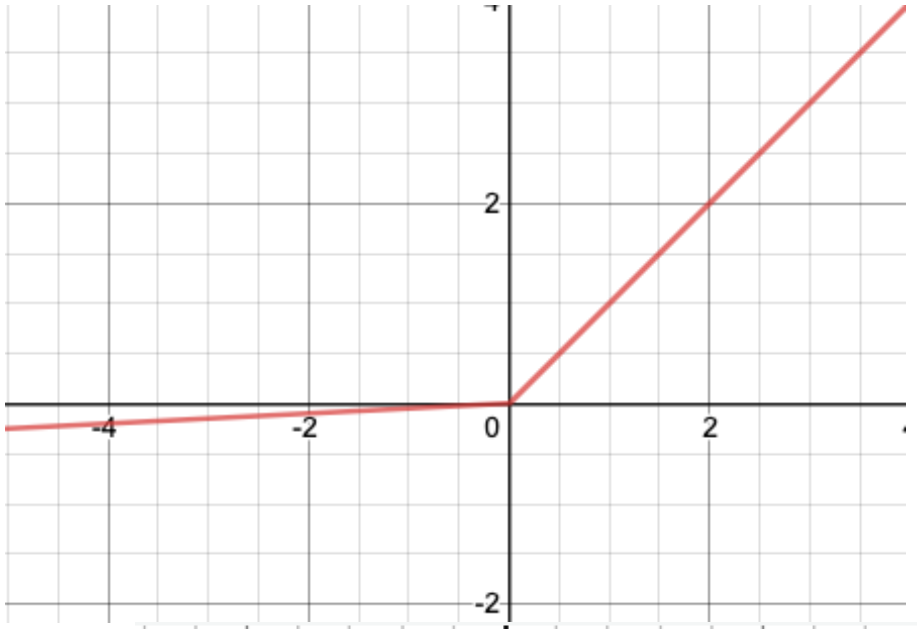


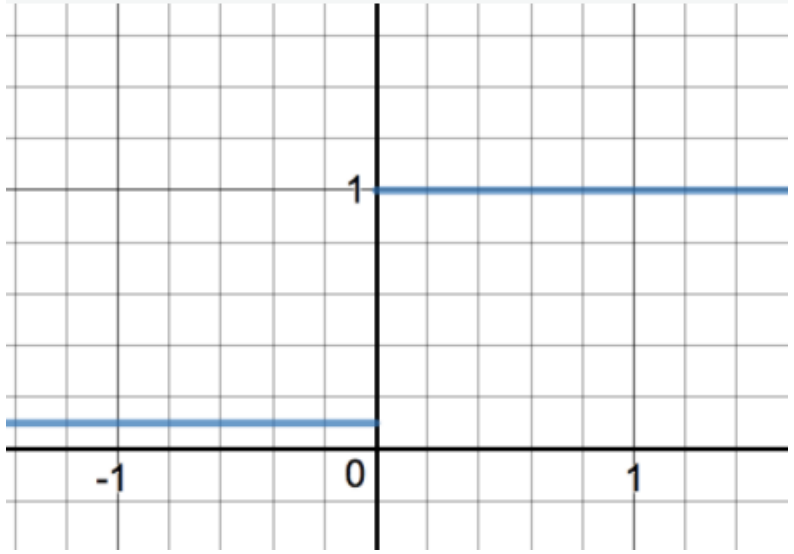


Leaky ReLU (Parametric ReLU)

Leaky Rectified Linear Unit

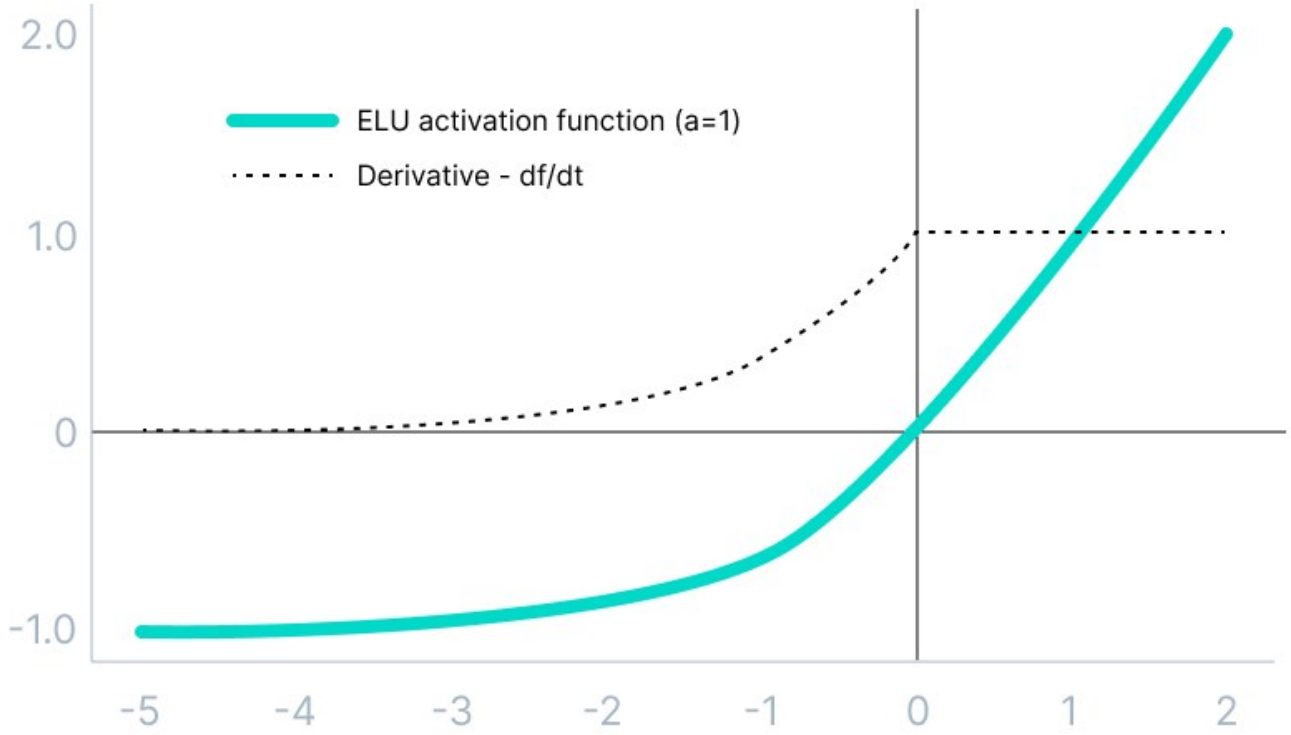
- $\varphi(x) = \max(\alpha x, x)$
- $\varphi'(x) = \begin{cases} \alpha, x \leq 0 \\ 1, x > 0 \end{cases}$
- Tipicamente $\alpha = 0.01$
- Vantagens:
  - Não satura
  - Gradientes não dissipam
  - Não “morre” como a ReLU
- Problemas:
  - O aprendizado para pré-ativações negativas tende a ser ruim
  - O gradiente pequeno α pode tornar o processo de aprendizado demorado





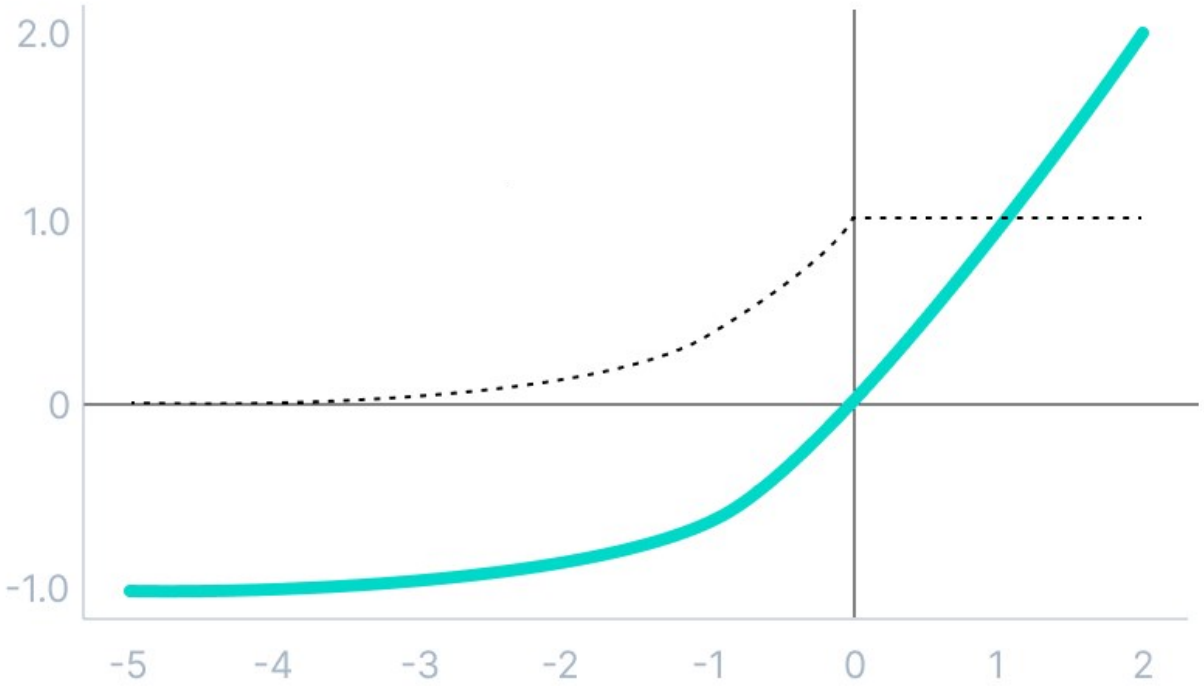
ELU

- *Exponential Linear Unit*
- $\varphi(x) = \begin{cases} \alpha(e^x - 1), x < 0 \\ x, x \geq 0 \end{cases}$
- $\varphi'(x) = \begin{cases} \alpha(e^x - 1) + \alpha, x \leq 0 \\ 1, x > 0 \end{cases}$



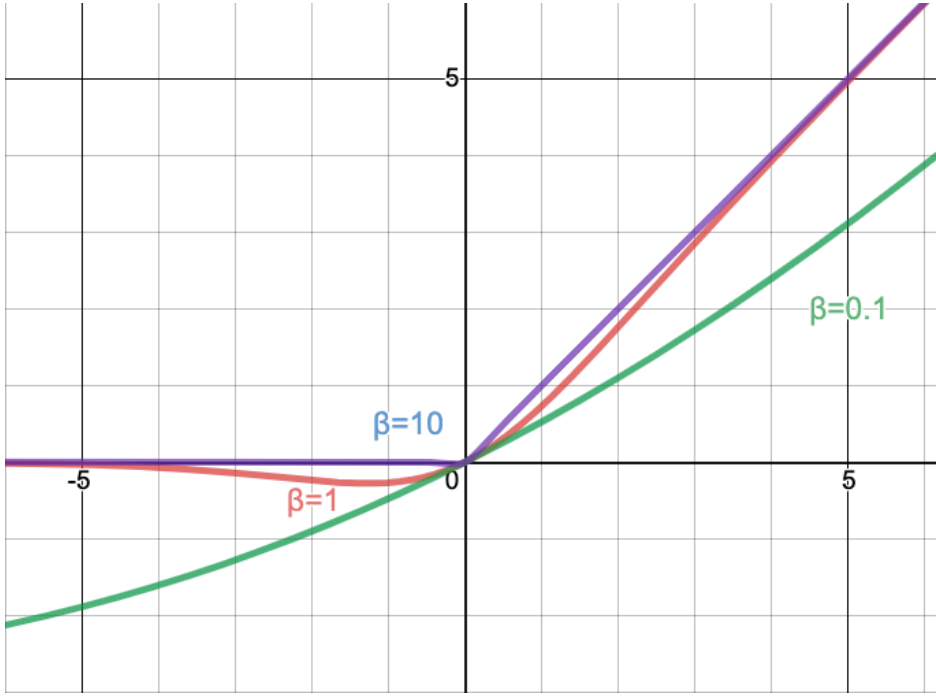
SELU

- Scaled Exponential Linear Unit
- $\varphi(x) = \begin{cases} \lambda\alpha(e^x - 1), x < 0 \\ \lambda x, x \geq 0 \end{cases}$
- $\varphi'(x) = \begin{cases} \lambda(\alpha(e^x - 1) + \alpha), x \leq 0 \\ \lambda, x > 0 \end{cases}$

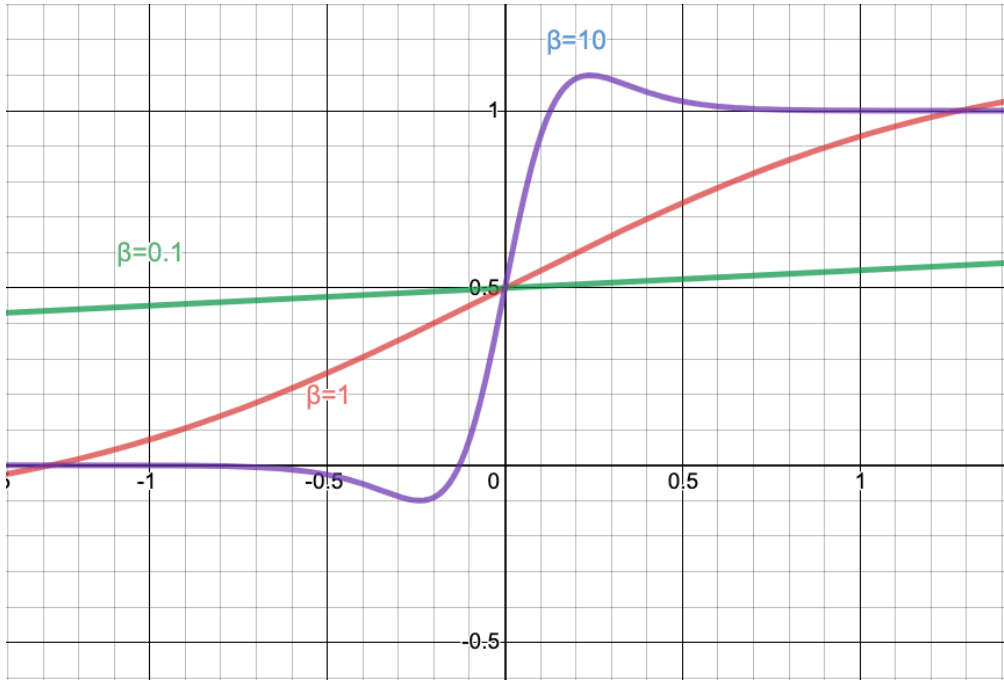


Swish

- $\varphi(x) = x\sigma(\beta x)$
- $\varphi'(x) = \beta x\sigma(\beta x) + (\sigma(\beta x)(1 - \beta\sigma(\beta x))$



- Função com limite inferior mas sem limite superior
- Permite que valores negativos pequenos sejam uteis no aprendizado
- Zera valores negativos com grande magnitude (pode gerar representação esparsa)
- Criada por pesquisadores da google, aparentemente gera resultados melhores do que as demais



RAMACHANDRAN, Prajit; ZOPH, Barret; LE, Quoc V. Searching for activation functions. **arXiv preprint arXiv:1710.05941**, 2017.

Linear ☺

- $\varphi(x) = x$
- $\varphi'(x) = 1$
- Não adiciona não-linearidades
- Não é limitada abaixo e acima

Funções de Ativação

- Podem atuar individualmente em unidades
- **Podem atuar em todas as unidades da camada**

Softmax

- $\varphi(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Essa função de ativação age em toda a camada simultaneamente, transformando as pré-ativações em números positivos cujo somatório é igual a 1 (termo normalizador no denominador)

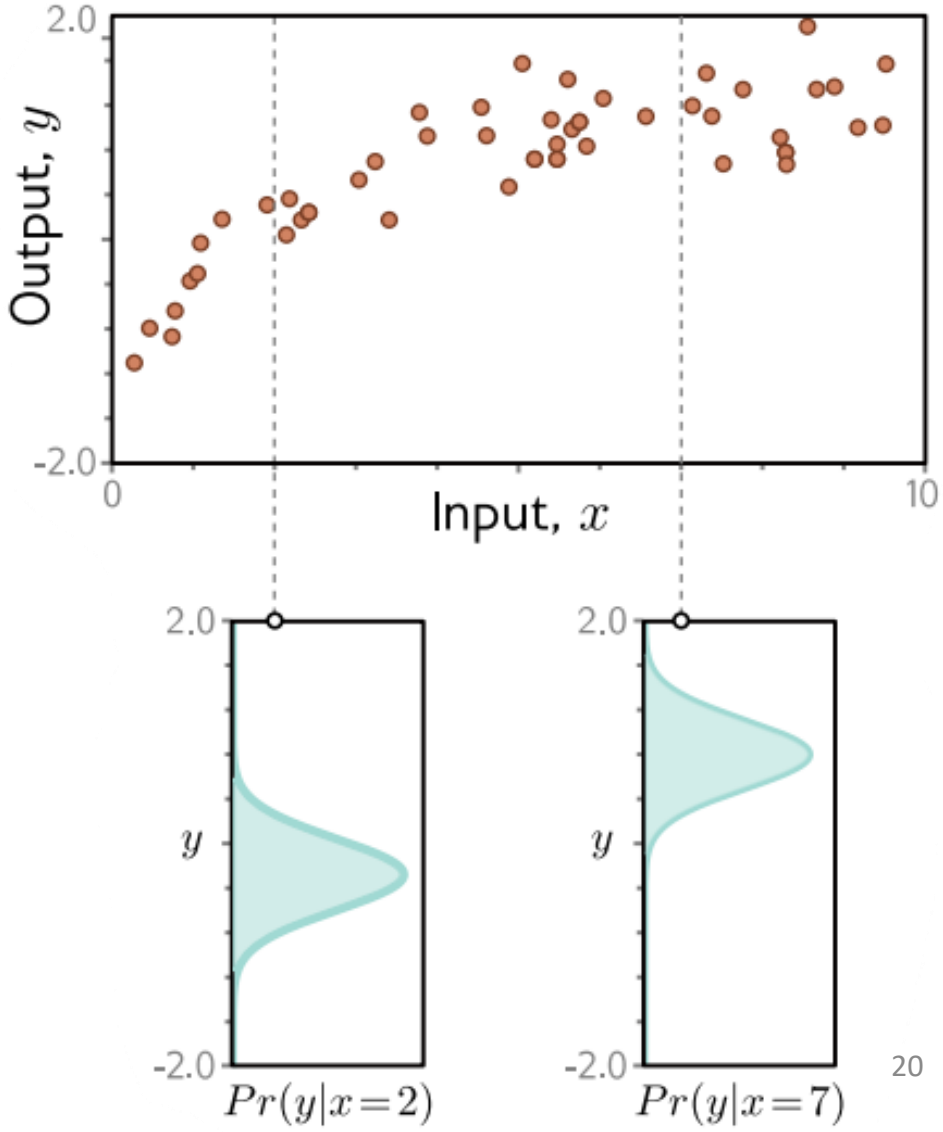
$$\varphi\left(\begin{array}{|c|}\hline \boldsymbol{s^{(1)}} \\ \hline 1.8 \\ \hline 0.7 \\ \hline -10 \\ \hline \end{array}\right) = \begin{array}{|c|}\hline \boldsymbol{o^{(1)}} \\ \hline 0.7500 \\ \hline 0.2497 \\ \hline 0.0003 \\ \hline \end{array}$$

Construindo Funções de Custo

- São usadas para avaliar o quão próximas as saídas da rede estão do *ground truth*
- Para definir funções de custo vamos mudar nossa forma de compreender a saída do nosso modelo
  - Ao invés de considerarmos um modelo $f[x, \theta]$, que computa uma saída $\hat{y}$ a partir da entrada x, vamos pensar que o modelo trabalha com uma probabilidade condicional
  - $\Pr(y|x)$ sobre todas as saídas possíveis y, condicionado pelo vetor de atributos x
- Vejamos exemplos do que essa mudança de paradigma implica

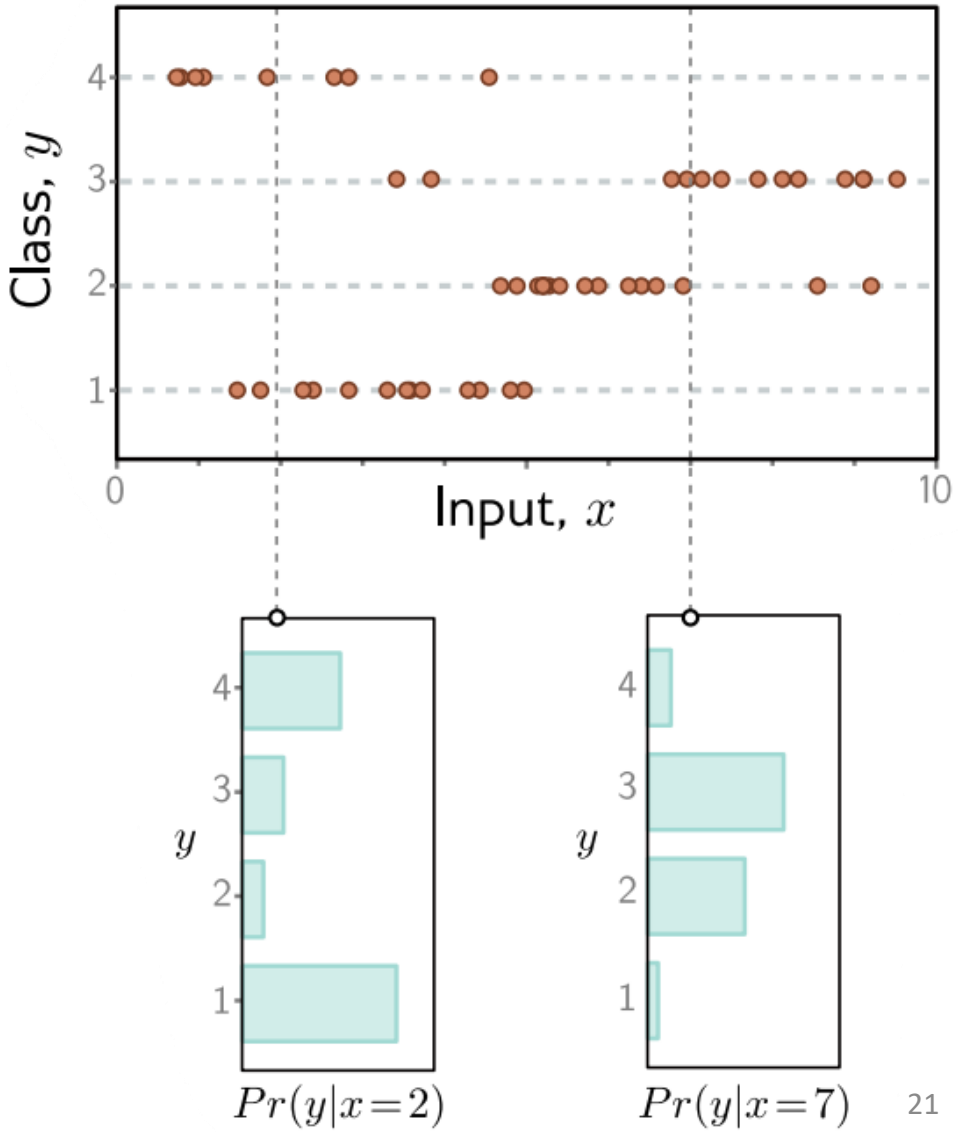
Construindo Funções de Custo

- Tarefa de regressão: $y \in \mathbb{R}$
- A Loss deve tentar maximizar a probabilidade da distribuição condicional $\Pr(y|x)$ a partir dos x correspondentes



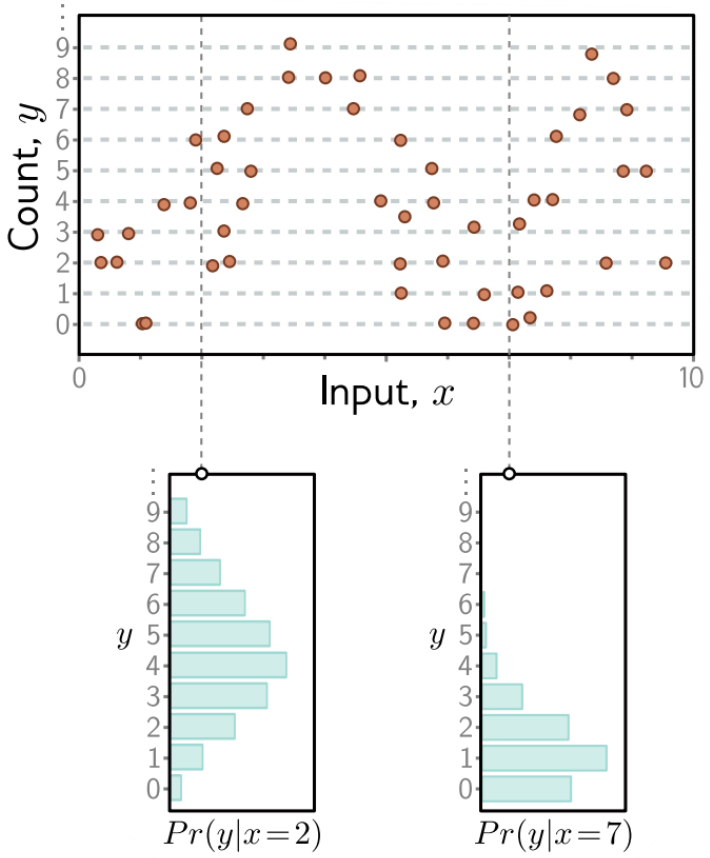
Construindo Funções de Custo

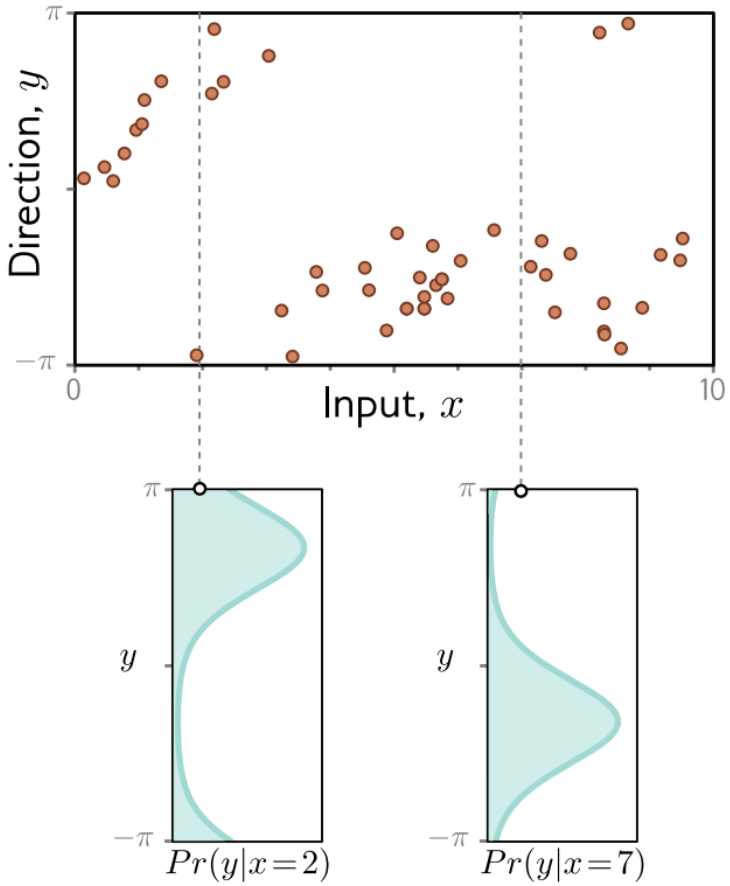
- Tarefa de classificação : $y \in \{1,2,3,4\}$
- Usamos uma distribuição discreta (Bernouli ou Multinouli) $\Pr(y|x)$ e nosso modelo tenta fazer uma predição do “histograma sobre as classes”



Construindo Funções de Custo

- Outras tarefas podem demander parametrização de outras distribuições





Construindo Funções de Custo

- Como fazer um modelo (rede neural) do tipo $f[x, \theta]$ computar uma distribuição de probabilidades?

Construindo Funções de Custo

- Como fazer um modelo (rede neural) do tipo $f[x, \theta]$ computar uma distribuição de probabilidades?
  - Escolhemos uma distribuição paramétrica que está definida no mesmo domínio de y e usamos a rede para computar os parâmetros dessa distribuição
  - Por exemplo: em um problema de regressão nós podemos definir uma gaussiana univariada, usar a rede para estimar μ e considerar a variância como um termo desconhecido

Construindo Funções de Custo

- Com nossa mudança na forma de ver o modelo, nossa saída são os parâmetros de uma distribuição de probabilidade sobre os dados de entrada
- Vamos tentar maximizar a verossimilhança

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\prod_{i=1}^{N} \Pr\left[y^{(i)}, f\left(x^{(i)}, \theta \right) \right] \right]$$

Construindo Funções de Custo

- Vamos tentar maximizar a verossimilhança

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\prod_{i=1}^{N} \Pr\left[y^{(i)}, f\left(x^{(i)}, \theta \right) \right] \right]$$

- Essa formulação traz duas assunções de forma implícita
  - Assumimos que $\left(x^{(i)}, y^{(i)} \right)$ são igualmente distribuídos
  - Assumimos que cada atributo x_j é independente
- Em outras palavras: *i.i.d.* ☺

Construindo Funções de Custo

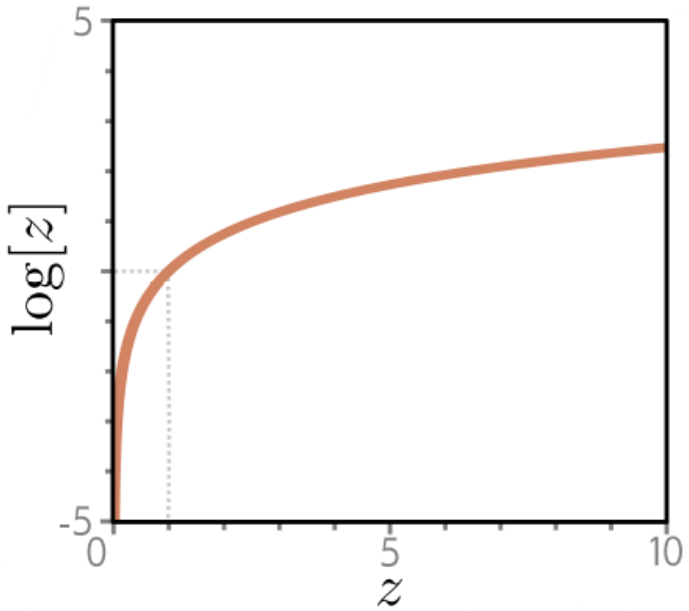
- Mas maximizar a verossimilhança é igual a maximizar a log verossimilhança

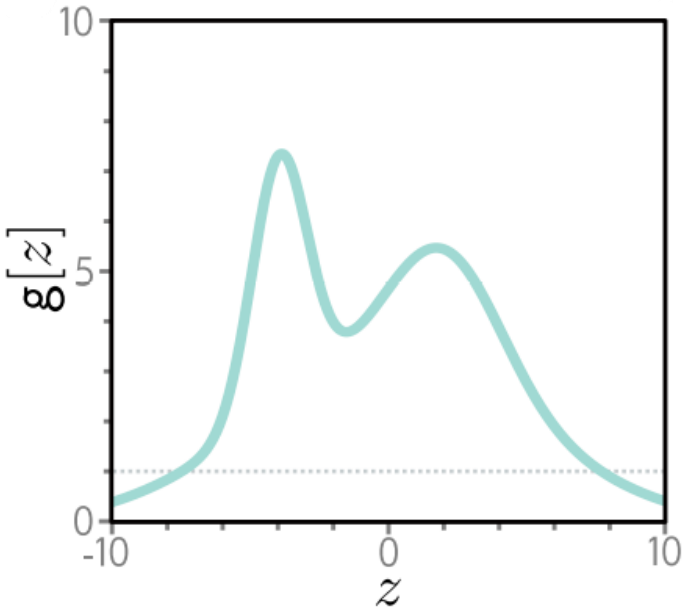
$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\prod_{i=1}^{N} \Pr\left[y^{(i)}, f\left(x^{(i)}, \theta \right) \right] \right]$$
$$= \underset{\theta}{\operatorname{argmax}} \left[\log \left[\prod_{i=1}^{N} \Pr\left[y^{(i)}, f\left(x^{(i)}, \theta \right) \right] \right] \right]$$
$$= \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{N} \log\left[\Pr\left[y^{(i)}, f\left(x^{(i)}, \theta \right) \right] \right] \right]$$

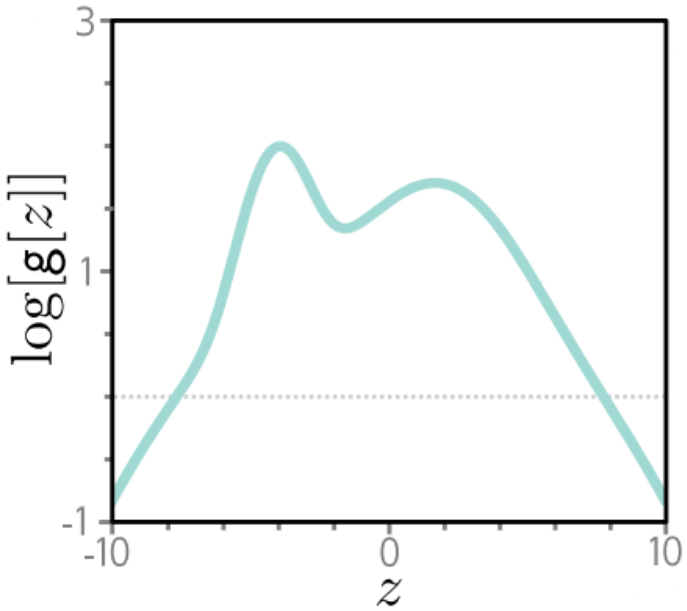
- Por que?

Construindo Funções de Custo

- Log é uma função monotônica crescente







Construindo Funções de Custo

- Por convenção, queremos minimizar uma função de custo e não maximizar

$$\hat{\theta} = \underset{\theta}{\text{argmax}}\left[\sum_{i=1}^{N} \log\left[\Pr\left[y^{(i)}, f\left(x^{(i)}, \theta\right)\right]\right]\right]$$

$$= \underset{\theta}{\text{argmin}}\left[-\sum_{i=1}^{N} \log\left[\Pr\left[y^{(i)}, f\left(x^{(i)}, \theta\right)\right]\right]\right]$$

$$= \underset{\theta}{\text{argmin}}\left[L[\theta]\right]$$

Construindo Funções de Custo

- Escolha uma distribuição de probabilidade sobre o domínio da variável alvo y que seja parametrizada por ϕ.
- Faça sua rede neural $f[x, \theta]$ estimar um ou mais desses parâmetros ϕ
  - De modo que $\phi = f[x, \theta]$ e $\Pr(y|\phi) = \Pr(y|f[x, \theta])$
- Treinar o modelo significa encontrar os parâmetros da rede $\hat{\theta}$ que minimizam a log verossimilhança negativa sobre os dados de treinamento $\{x^{(i)}, y^{(i)}\}$:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[-\sum_{i=1}^{N} \log\left[\Pr\left[y^{(i)}, f\left(x^{(i)}, \theta\right)\right]\right] \right] = \underset{\theta}{\operatorname{argmin}}\left[L[\theta]\right]$$

- Para fazer uma predição, retorne o valor máximo ou a distribuição toda para uma nova instância x

Construindo Funções de Custo

Exemplo 1 – Regressão

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 1: Escolher a distribuição

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 1: Escolher a distribuição

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

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 2: Rede neural vai estimar a média da gaussiana

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

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 3: Essa é a Loss, temos que treinar o modelo

- Mas podemos deixar ela mais bonita ☺

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[-\sum_{i=1}^{N} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[-\frac{(y - \boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^2}{2\sigma^2} \right] \right] \right]$$

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 3: Essa é a Loss, temos que treinar o modelo

- Mas podemos deixar ela mais bonita ☺
  - Aplicamos uma identidade de logaritmos

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[-\sum_{i=1}^{N} \left[\log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \right] - \frac{(y - \boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^2}{2\sigma^2} \right] \right]$$

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 3: Essa é a Loss, temos que treinar o modelo

- Mas podemos deixar ela mais bonita ☺
  - Removemos um termo que não tem relação aos parâmetros

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

Construindo Funções de Custo

Exemplo 1 – Regressão

Passo 3: Essa é a Loss, temos que treinar o modelo

- Mas podemos deixar ela mais bonita ☺
  - Multiplicação por -1 pode sair do somatório
  - A divisão pela variância pode ser considerada uma constante e removida da expressão pois ela não influencia nos pontos de mínimo e máximo

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

Construindo Funções de Custo

Exemplo 1 – Regressão

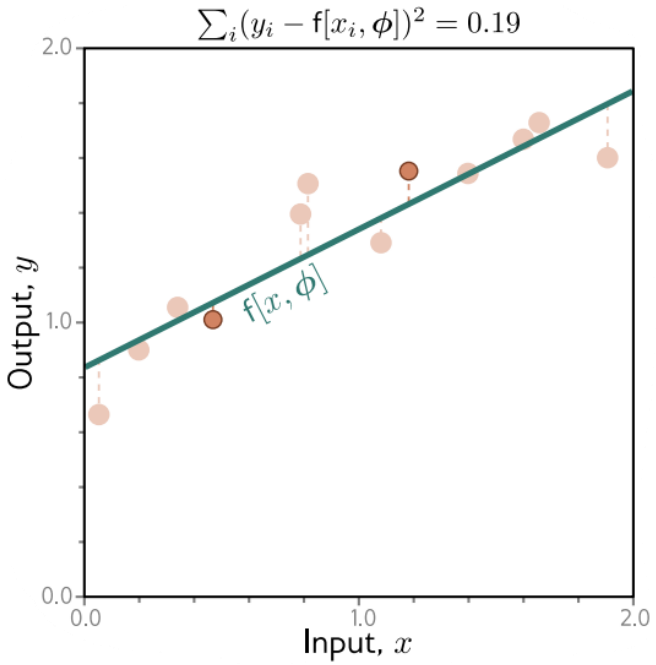
- Essa é a Loss dos mínimos quadrados (L2_loss)
  - segue naturalmente das assunções que as instâncias são independentes e amostradas de uma gaussiana univariada

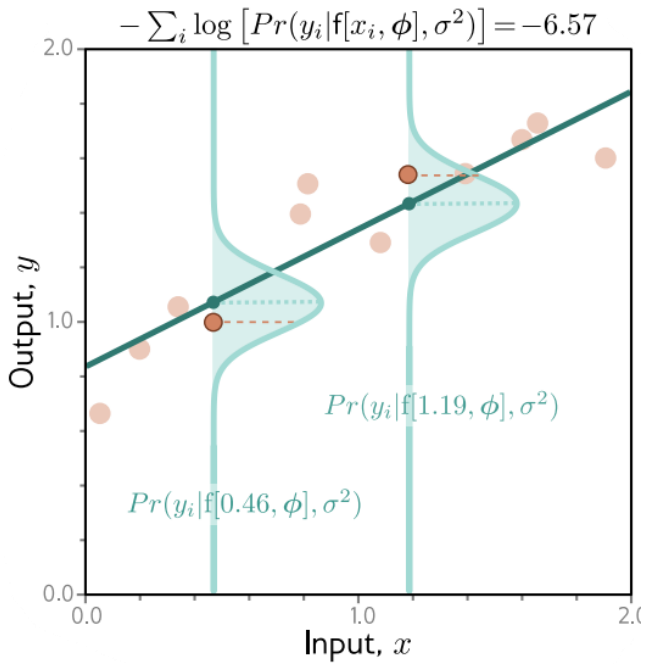
$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\sum_{i=1}^{N} (y - \boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^2 \right]$$

Construindo Funções de Custo

Exemplo 1 – Regressão

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





Construindo Funções de Custo

Exemplo 1 – Classificação Binária

Passo 1: Escolher a distribuição

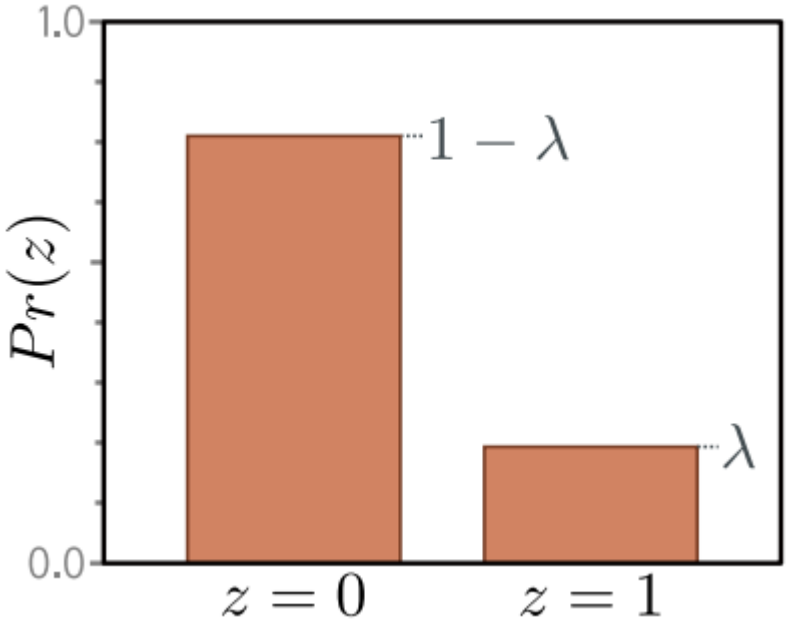
Construindo Funções de Custo

Exemplo 1 – Classificação Binária

Passo 1: Escolher a distribuição

Bernoulli

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

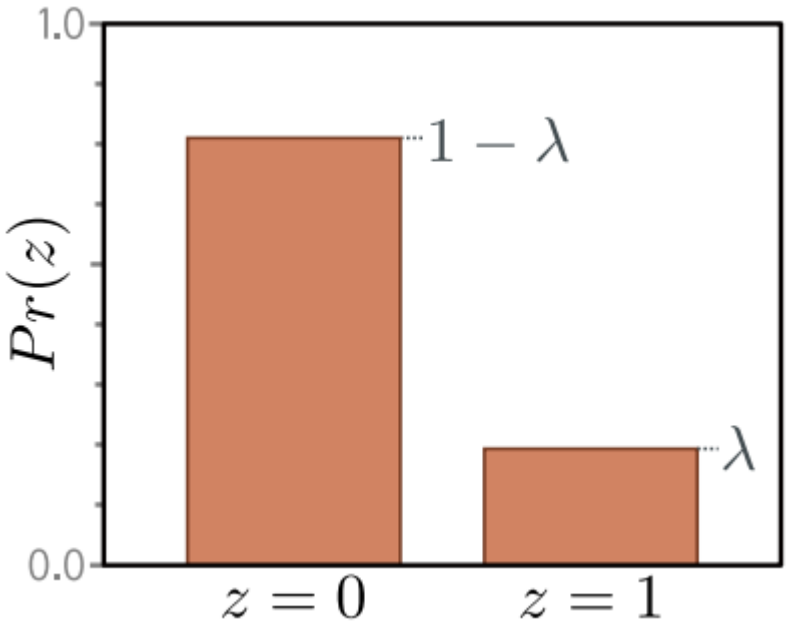


Construindo Funções de Custo

Exemplo 1 – Classificação Binária

Passo 2: A rede vai estimar o λ

$$\Pr(y|\lambda) = (1 - \boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^{1-y} \boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}]^{y}$$



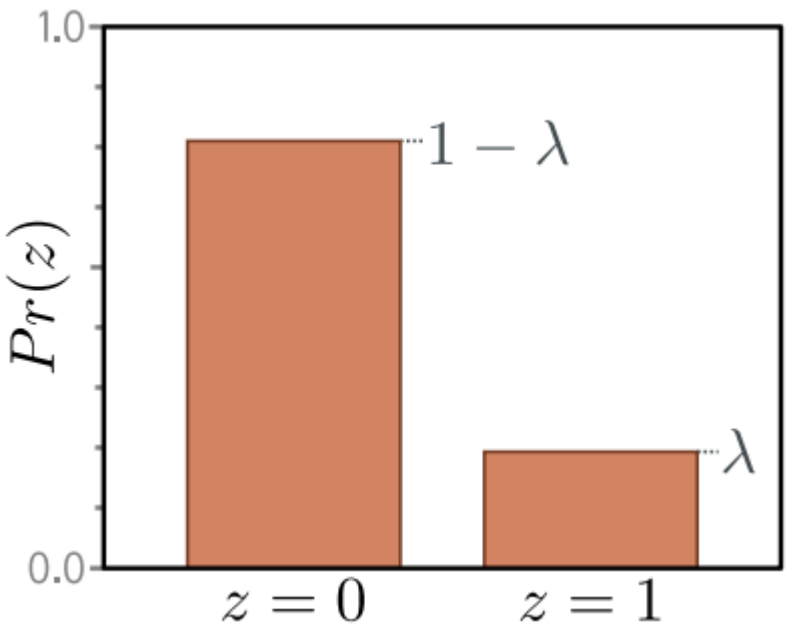
Construindo Funções de Custo

Exemplo 1 – Classificação Binária

Passo 2: A rede vai estimar o λ

- $\lambda \in [0,1]$, para garantir isso vamos passar a saída da rede em uma sigmoid

$$\Pr(y|\lambda) = \left(1 - \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])\right)^{1-y} \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^{y}$$



Construindo Funções de Custo

Exemplo 1 – Classificação Binária

Passo 3: Essa é a Loss, temos que treinar o modelo

- Mas podemos deixar ela mais bonita ☺

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\sum_{i=1}^{N} -\log \left[\left(1 - \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])\right)^{1-y^{(i)}} \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^{y^{(i)}} \right] \right]$$

Construindo Funções de Custo

Exemplo 1 – Classificação Binária

Passo 3: Essa é a Loss, temos que treinar o modelo

- Mas podemos deixar ela mais bonita ☺

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[\sum_{i=1}^{N} -\log \left[\left(1 - \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])\right)^{1-y^{(i)}} \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])^{y^{(i)}} \right] \right]$$

$$= \underset{\theta}{\operatorname{argmin}} \left[\sum_{i=1}^{N} \left[-\left(1 - y^{(i)}\right) \log\left(1 - \boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])\right) - \left(y^{(i)}\right) \log\left(\boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\theta}])\right) \right] \right]$$

Construindo Funções de Custo

Exemplo 1 – Classificação Binária

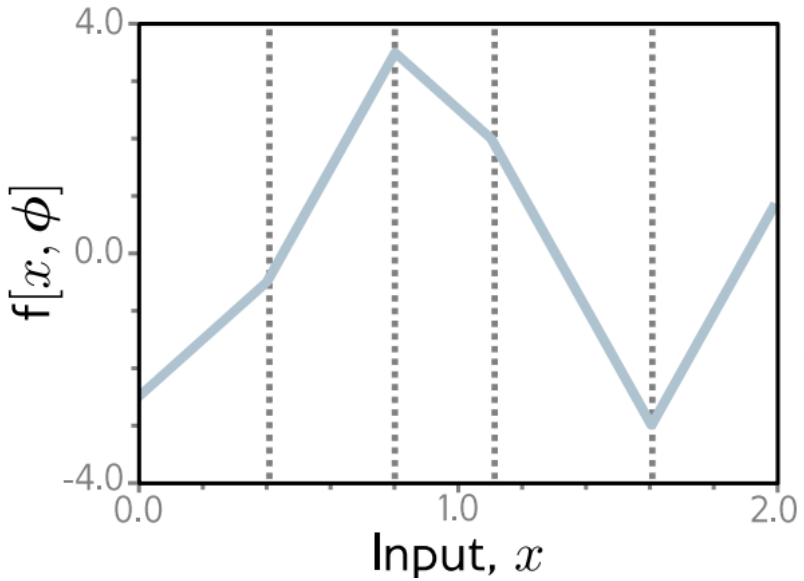
Passo 3: Essa é a **Entropia Binária Cruzada** (*BCE Loss*)

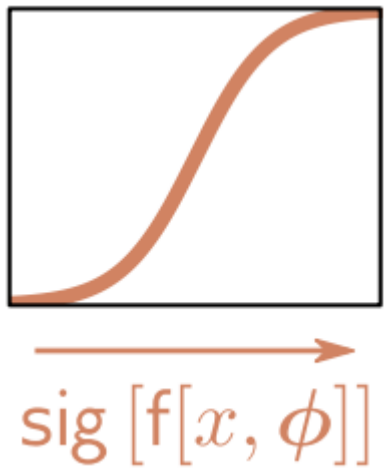
$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\left[\sum_{i=1}^{N}\left[-\left(1-y^{(i)}\right)\log\left(1-\boldsymbol{\sigma}\left(\boldsymbol{f}[\boldsymbol{x},\boldsymbol{\theta}]\right)\right)-\left(y^{(i)}\right)\log\left(\boldsymbol{\sigma}\left(\boldsymbol{f}[\boldsymbol{x},\boldsymbol{\theta}]\right)\right)\right]\right]$$

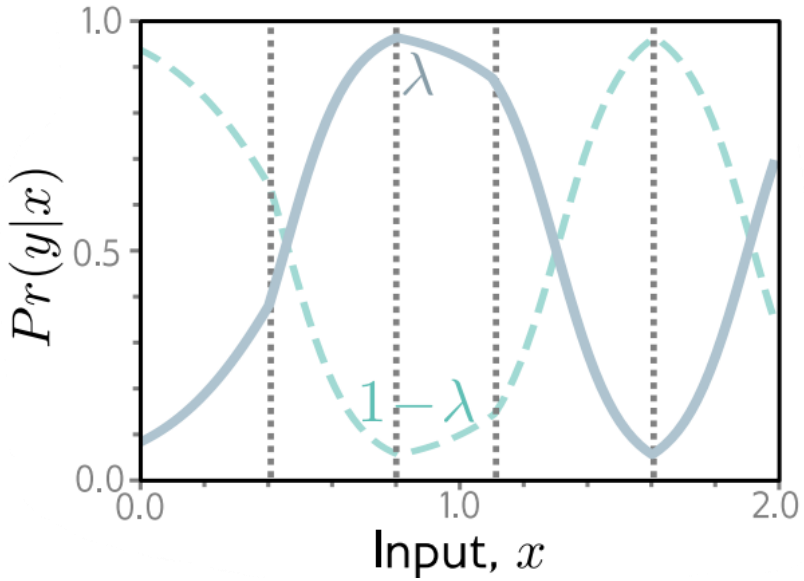
Construindo Funções de Custo

$$\hat{\theta} = \underset{\theta}{\text{argmin}}\left[\sum_{i=1}^{N}\left[-(1-y^{(i)})\log(1-\boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x},\boldsymbol{\theta}]))-(y^{(i)})\log(\boldsymbol{\sigma}(\boldsymbol{f}[\boldsymbol{x},\boldsymbol{\theta}]))\right]\right]$$

- Repare que a pré-ativação da última camada é uma função *piecewise* linear
- Depois da ativação as saídas estão transformadas para a faixa de 0 a 1







Loss vs Função de Ativação

- Ao utilizar um *framework* como *PyTorch* devemos ter atenção em como as funções de custo estão implementadas

BCEWITHLOGITSLOSS

CLASS torch.nn.BCEWithLogitsLoss(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*, *pos_weight=None*) [SOURCE]

This loss combines a *Sigmoid* layer and the *BCELoss* in one single class. This version is more numerically stable than using a plain *Sigmoid* followed by a *BCELoss* as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top, \quad l_{n,c} = -w_{n,c} \left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c})) \right]$$

```
>>> target = torch.ones([10, 64], dtype=torch.float32)  # 64 classes, batch size = 10
>>> output = torch.full([10, 64], 1.5)  # A prediction (logit)
>>> pos_weight = torch.ones([64])  # All weights are equal to 1
>>> criterion = torch.nn.BCEWithLogitsLoss(pos_weight=pos_weight)
>>> criterion(output, target)  # -log(sigmoid(1.5))
tensor(0.20...)
```

BCELOSS

CLASS torch.nn.BCELoss(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

```
>>> m = nn.Sigmoid()
>>> loss = nn.BCELoss()
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> output = loss(m(input), target)
>>> output.backward()
```

Loss vs Função de Ativação

BCEWITHLOGITSLOSS

CLASS torch.nn.BCEWithLogitsLoss(*weight=None, size_average=None, reduce=None, reduction='mean', pos_weight=None*) [SOURCE]

This loss combines a *Sigmoid* layer and the *BCELoss* in one single class. This version is more numerically stable than using a plain *Sigmoid* followed by a *BCELoss* as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top, \quad l_{n,c} = -w_{n,c} \left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c})) \right]$$

```
>>> target = torch.ones([10, 64], dtype=torch.float32)  # 64 classes, batch size = 10
>>> output = torch.full([10, 64], 1.5)  # A prediction (logit)
>>> pos_weight = torch.ones([64])  # All weights are equal to 1
>>> criterion = torch.nn.BCEWithLogitsLoss(pos_weight=pos_weight)
>>> criterion(output, target)  # -log(sigmoid(1.5))
tensor(0.20...)
```

- Trabalha nos logits não normalizados
- Possui um parâmetro para peso da classe positiva

BCELOSS

CLASS torch.nn.BCELoss(*weight=None, size_average=None, reduce=None, reduction='mean'*) [SOURCE]

Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

```
>>> m = nn.Sigmoid()
>>> loss = nn.BCELoss()
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> output = loss(m(input), target)
>>> output.backward()
```

- Trabalha na saída de uma sigmoid
- Não possui um parâmetro para peso da classe positiva

Loss vs Tarefa vs Ativação

- *Multiclass*
  - Existem mais de duas classes e elas são mutuamente exclusivas
- Multilabel
  - Existem mais de um rótulo e eles não são mutuamente exclusivos

1. Dog
2. Cat
3. Mouse
4. Human

1. Dog
2. Cat
3. Mouse
4. Human

Métricas de Avaliação

- Regressão
  - MSE
  - MAE
  - R^2
- Classificação
  - Acurácia
  - *Recall*
  - *Precision*
  - $f\beta\_score$

Métricas de Avaliação - Regressão

- $MSE = \frac{\sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right)^2}{N}$
- $MAE = \frac{\sum_{i=1}^{N} \left|\hat{y}^{(i)} - y^{(i)}\right|}{N}$
- $R^2 = \frac{\sum_{i=1}^{N} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{N} \left(y^{(i)} - \bar{y}^{(i)}\right)^2}$

Métricas de Avaliação - Classificação

- $Acc = \frac{TP+TN}{TP+TN+FP+FN}$
- $Recall = \frac{TP}{TP+FN}$
- $Precision = \frac{TP}{TP+FP}$
- $f_\beta score = (1+\beta^2) * \frac{Precision*Recall}{\beta^2*Precision+Recall}$

		Valor Predito	
		Sim	**Não**
Real	**Sim**	Verdadeiro Positivo (TP)	Falso Negativo (FN)
	Não	Falso Positivo (FP)	Verdadeiro Negativo (TN)

TLDR

Para Camada de Saída

- Regressão
  - Uma unidade Linear
- Classificação
  - Binária
    - Uma unidade Sigmoid
  - Multilabel
    - Uma unidade por label Sigmoid
  - Multiclass
    - Uma unidade por classe Softmax

Referências:

- Sugere-se **FORTEMENTE** a leitura de:
  - Capítulo 5: Understanding Deep Learning
    - https://udlbook.github.io/udlbook/