Aprendizado de máquina

- Aprendizado supervisionado: "ensinar" o computador a dar respostas à dados de entrada desconhecidos a partir de dados cujas respostas são conhecidas. (dados de treinamento)
- Onde entram os algoritmos de otimização?
  ↳ no ajuste dos parâmetros do sistema que dá as respostas! (é o treinamento)

Exemplo motivacional:

Suponha que queiramos treinar o computador para decidir se um dado texto disserta sobre esportes.

- São dados n textos $(x_1, y_1), \ldots, (x_n, y_n)$ para o treinamento, em que sabemos se são sobre esporte ou não:
  - $x_i \in \mathbb{R}^m$ vetor de características do texto (palavras, por exemplo)
  - $y_i \in \{-1, 1\}$; $1 =$ esporte, $-1 =$ não.

Devemos definir uma <u>função de predição</u> h, cuja qualidade é medida contando o número de respostas erradas ($h(x_i) \neq y_i$). Neste sentido, queremos <u>minimizar</u> o <u>risco empírico</u> de uma má classificação

$$R_n(h) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(x_i) \neq y_i],$$

onde

$$\mathbb{1}(P) = \begin{cases} 1 & \text{se } P \text{ é verdadeiro} \\ 0 & \text{caso contrário.} \end{cases}$$

Problemas:

1) R_n nem ao menos é contínua...

2) mesmo que $R_n(h) = 0$ (h nunca erra nos dados de treinamento), pode ser que h seja ruim para predizer a resposta em dados desconhecidos (overfitting)

Ex.: $\tilde{h}(x) = \begin{cases} y_i & \text{se } x = x_i \text{ p/ algum } i \\ \pm 1 & \text{caso contrário (aleatório).} \end{cases}$

Veja que $R_n(\tilde{h}) = 0$, mas $\tilde{h}$ pode ser qualquer coisa sobre dados desconhecidos...

Como resolver?

- Escolher h dentro de uma classe de funções (por exemplo, funções afins)
- Uma escolha adequada é feita por testes numéricos para o problema considerado.

- de certa forma, h deve capturar as características fundamentais do problema. Assim, teremos respostas corretas a dados desconhecidos mesmo que o treinamento seja feito com número limitado de dados conhecidos (evidentemente os dados de treinamento devem ser representativos)
- é desejável h diferenciável e fácil de computar (geralmente o treinamento é feito com muitos (x_i, y_i) $(n \gg 1)$).

Os conjuntos de dados coletados por experimentação (resposta manual, humana) são divididos em:

- dados de treinamento: usados para treinar o computador ($\min R_n(h)$).
- dados de validação: usados para escolha da melhor função de predição h (já otimizada)
- dados de teste: usados para medir a eficácia da h escolhida.

Escolha comum:

$$h(x) = h(x; w, b) = w^t x + b,$$

w: pesos;
b: viés
> devem ser otimizados para buscar $h(x_i; w^*, b^*) = y_i$.

Funciona bem em vários problemas.

$$R_n(w, b) = \frac{1}{n} \sum_{i=1}^{n} \text{sinal}(h(x_i; w, b), y_i)$$

$\text{sinal}(h(x_i; w, b), y_i) \in \{0, 1\}$ é descontínua.

↳ intratável para $n \gg 1$ por métodos de PNL.

Ao invés de utilizar "sinal", utilizamos funções contínuas aproximadoras de ℓ (funções de perda), escolhidas por experimentação numérica.

Exemplos

1) $\ell(h,y) = \log(1+e^{-hy})$ (para $h \in \mathbb{R}-\{0\}$)

Veja que $\min \ell(h,y)$ é atingido quando $hy = 1$, isto é, h e y têm mesmo sinal.

2) $\ell(h,y) = (h-y)^2$.

Função a ser minimizada:

$$R_n(\bullet) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(x_i; \bullet), y_i).$$

---

h como sucessivas aplicações — Redes neurais

Dado um vetor entrada $x \in \mathbb{R}^d$, escrevemos

$$h(x) = f_L(f_{L-1}(\cdots f_1(x) \cdots))$$

(aplicação sucessiva de $f_1, f_2, \ldots, f_L$).

- Trabalhar com h deste modo é efetivo em aplicações.
- Para <u>avaliar $h(x)$</u>, avaliamos em sequência

* $f_1 = f_1(x)$

* $f_2 = f_2(f_1)$

* $f_3 = f_3(f_2)$

⋮

* $h(x) = f_L(f_{L-1})$

} "feedforward" ($f_1 \longrightarrow f_L$).

- Para <u>avaliar $\nabla h(x)$</u>, usamos a regra da cadeia. Para simplificar, vamos supor que $f_i : \mathbb{R} \to \mathbb{R}$, $\forall i$. Assim,

$$h'(x) = f_L'(f_{L-1}) \cdot f_{L-1}'(f_{L-2}) \cdots f_1'(x).$$

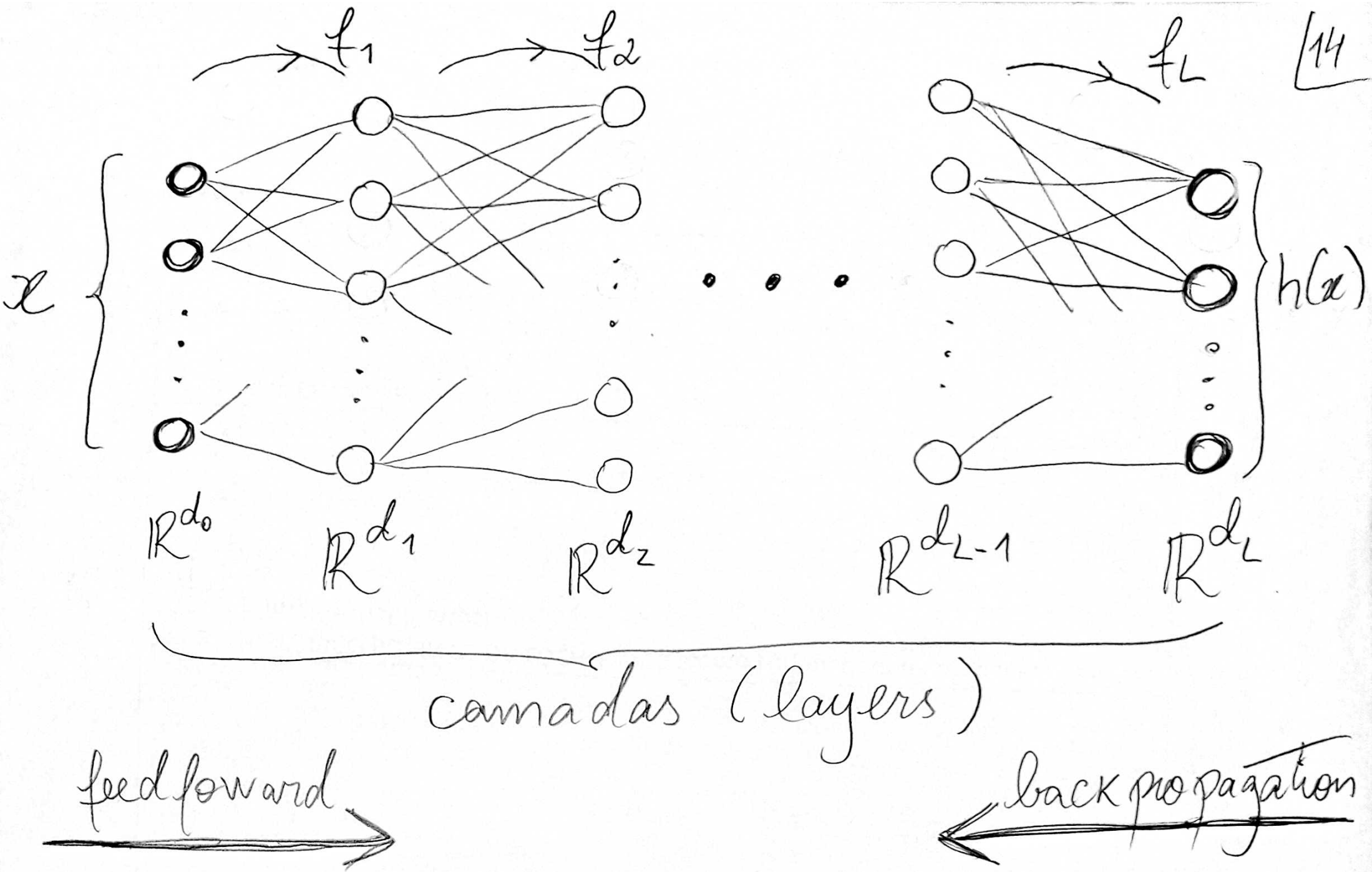
Então, calculados $f_1, \ldots, f_{L-1}$, fazemos:

* $f_L'(f_{L-1})$
* $f_{L-1}'(f_{L-2})$
⋮
* $f_1'(x)$

} "backpropagation" ($f_L \longrightarrow f_1$).

É claro que, em geral, $f_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$, e temos que usar a regra da cadeia para várias variáveis...

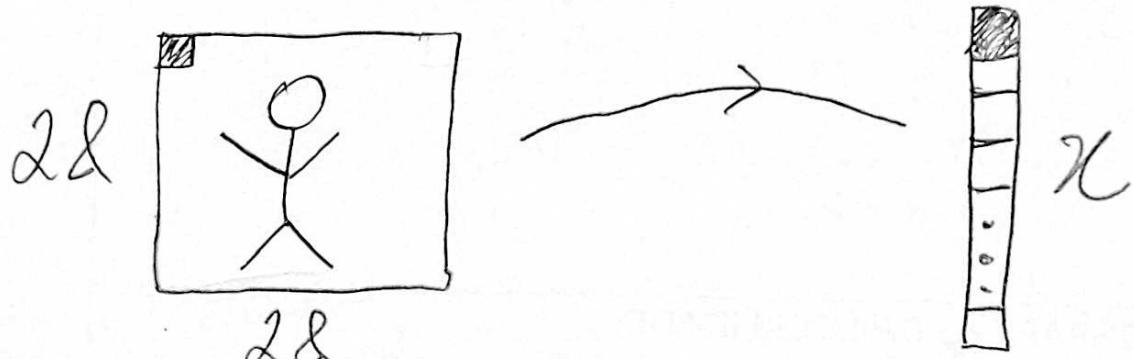
A concatenação das f_i's pode ser vista por meio de um grafo. Por sua inspiração em modelos para neurônios, tal grafo é chamado rede neural. Os nós junto com "seu mecanismo de controle do fluxo" é chamado neurônio.


f_1
f_2
f_L
x
$h(x)$
$\mathbb{R}^{d_0}$
$\mathbb{R}^{d_1}$
$\mathbb{R}^{d_2}$
$\mathbb{R}^{d_{L-1}}$
$\mathbb{R}^{d_L}$
camadas (layers)
feedforward
backpropagation

- Camada mais à esquerda: <u>camada de entrada</u>

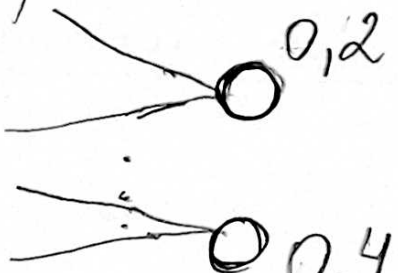
  ↳ é onde o dado x é codificado.

  Ex.: $x \in [0,256]^{784}$ representando a escala de cinza de uma imagem em tons de cinza de tamanho $2\ell \times 2\ell$.



- Camada mais à direita: <u>camada de saída</u>
  ↳ é o valor de $h(x)$

<u>Ex 1)</u> resposta -1 ou 1.



$$\text{resposta} = \begin{cases} 1 \text{ se } h_1(x) \geq h_2(x) \\ -1 \text{ c.c.} \end{cases}$$

<u>Ex: 2)</u> x é a imagem de um dígito entre 0 e 9 escrito à mão.

$h(x) \in [0,1]^{10}$ é um vetor de ... 17

10 coordenadas. A resposta à pergunta "qual o dígito da imagem x?" é dada por

$$\underset{0 \leq i \leq 9}{\operatorname{argmax}} \; h_{i+1}(x).$$

É comum que cada f_j seja a composta de uma aplicação afim com uma função não linear a (<u>função de ativação</u>):

$$f_j : \mathbb{R}^{d_{j-1}} \longrightarrow \mathbb{R}^{d_j} \quad , j = 1, \dots, L$$

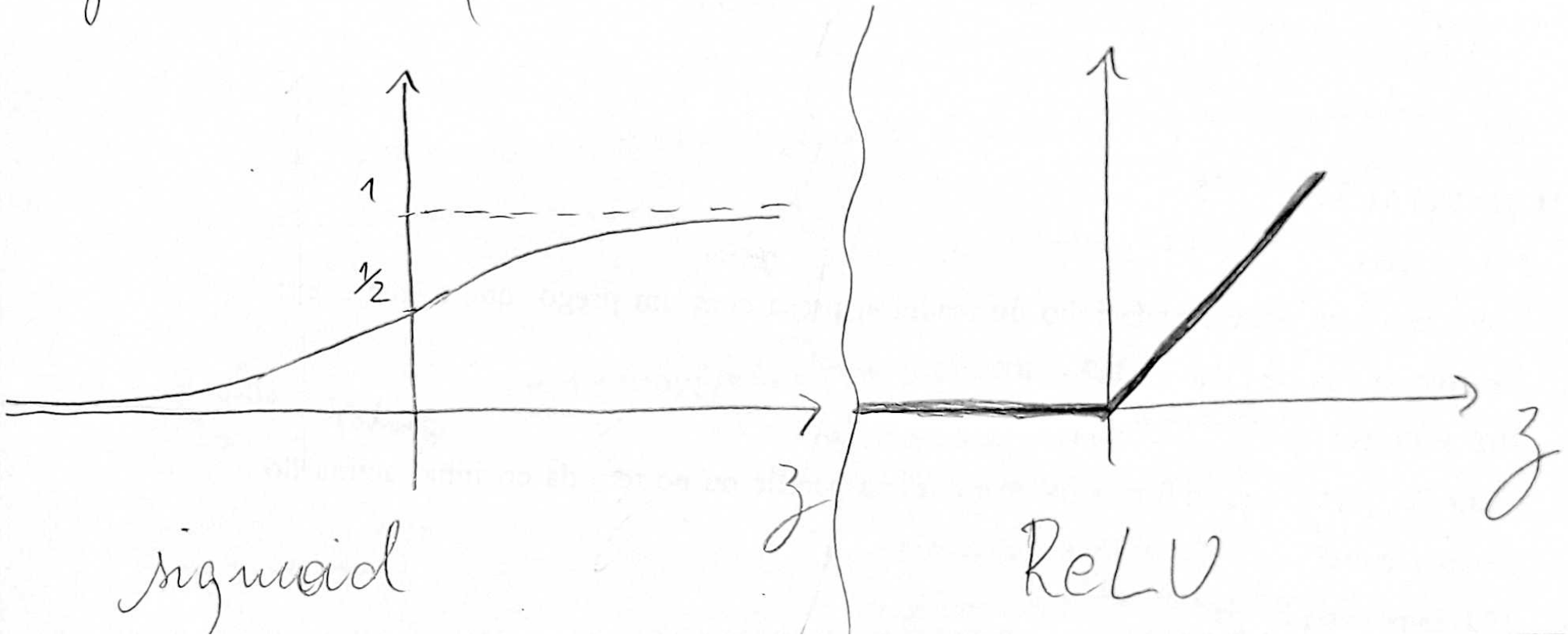
$$\boxed{x_i^{(j)} = a\left(W_j x_i^{(j-1)} + b_j\right) \in \mathbb{R}^{d_j} \quad , \forall i}$$

Algumas escolhas:

- $a(z) = \dfrac{1}{1+e^{-z}}$ (sigmoid)

- $a(z) = \max\{0, z\}$ (rectified linear unit – ReLU).

As funções de ativação tem o intuito de ajustar o fluxo na rede neural.



Objetivo neste caso é minimizar

$$R_n(w,b) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(x_i; w, b); y_i).$$

Essa função é não linear e não convexa. Felizmente isso não é grande problema nas aplicações → métodos tipo gradiente dão boas soluções (isto é, bons (w^*, b^*), que são a rede neural treinada).

O processo de otimização de (w, b) é o treinamento da rede neural.

Obs: 1) a soma em $R_n(w, b)$ envolve muitos termos ($n \gg 1$) pois geralmente há muitos dados no treinamento. Portanto calcular ∇R_n é caro (iteração do método do gradiente)

↳ Solução: trocar ∇R_n pela soma de alguns poucos gradiente da soma a cada iteração do método → gradiente incremental.

2) Ao obter os parâmetros (w^*, b^*) da rede neural (pesos e viéses de cada neurônio) podemos entrar com um $\tilde{x}$ desconhecido e tomar a resposta na camada de saída. (rede treinada)

---

3) Podemos ver os dados de treinamento como amostras de uma variável aleatória. Nesse sentido, R_n é uma aproximação da função objetivo verdadeira: o risco esperado, ou

a esperança de má classificação:

$$R(h) = \mathbb{P}[h(x) \neq y] = \mathbb{E}[\mathbb{1}[h(x) \neq y]].$$

Aqui, $\mathbb{P}[h(x) \neq y]$ é a probabilidade de h errar a resposta para x. Note que x é qualquer, não mais somente um dado da amostra de treinamento.

Não conhecemos $\mathbb{P}$, apenas uma aproximação pelos dados de treinamento. Porém,

queremos dizer algo em relação à $R(h)$ (convergência em probabilidade do método de gradiente incremental).

O método "resultante" da ideia de gradiente incremental com a probabilidade é o método do gradiente estocástico. Ele (e variantes) são os mais utilizados no treinamento de redes (lembre-se que gradientes são calculados por backpropagation).