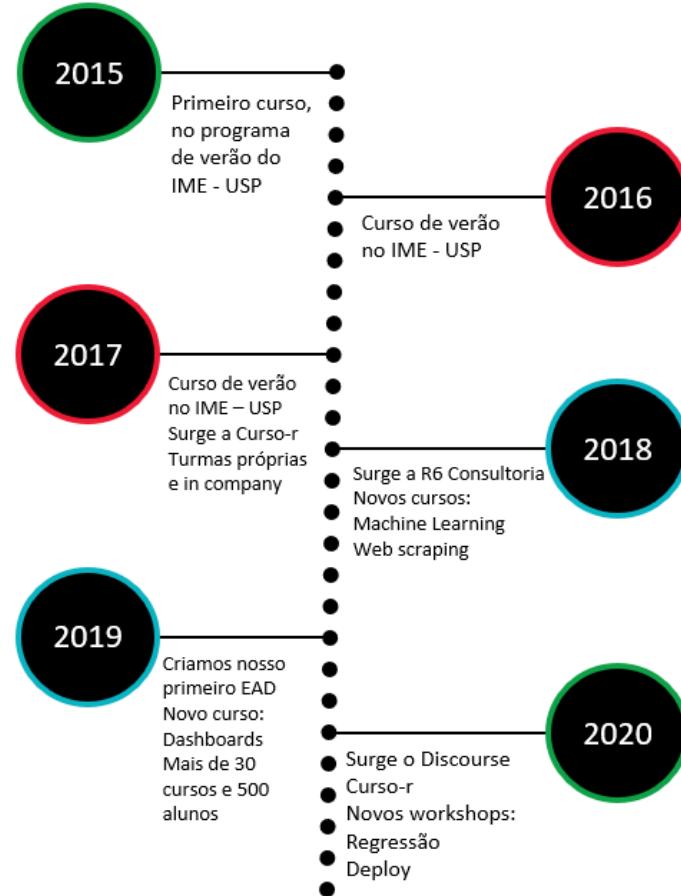


# Introdução ao Deep Learning com R

## Curso-R

# **Curso-R**

# Linha do tempo



# Nossos cursos

## Programação em R

---

R para Ciência de dados I

R para Ciência de dados II

## Modelagem

---

Regressão Linear

Machine Learning

XGBoost

Deep Learning

## Extração de dados

---

Web scraping I

Web scraping II

## Comunicação e automação

---

Dashboards com R

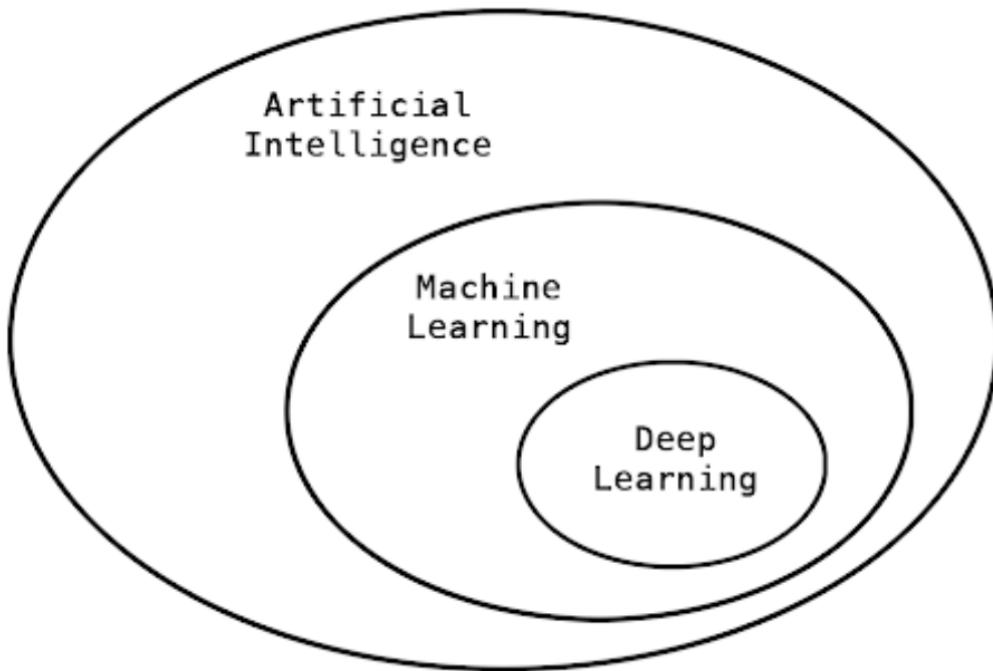
Deploy

# Nesse curso vamos falar de

- Regressão linear e logística
- Multi-layer perceptrons
- Redes neurais convolucionais
- Modelos pré-treinados para imagens
- Redes neurais para textos
- Redes neurais recorrentes
- Modelos pré-treinados para textos

# O que é Deep learning?

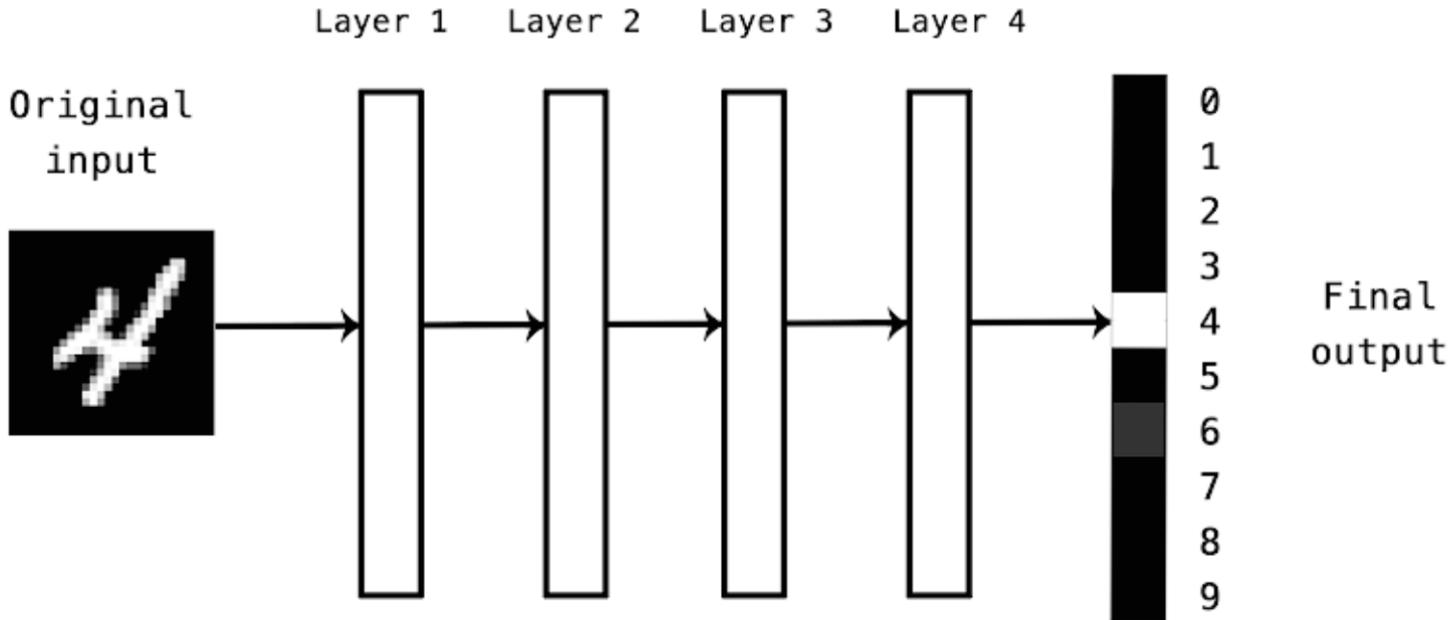
- Subconjunto de técnicas de Machine Learning



Fonte: Deep Learning with R, Chollet, F et al

# O que é Deep learning?

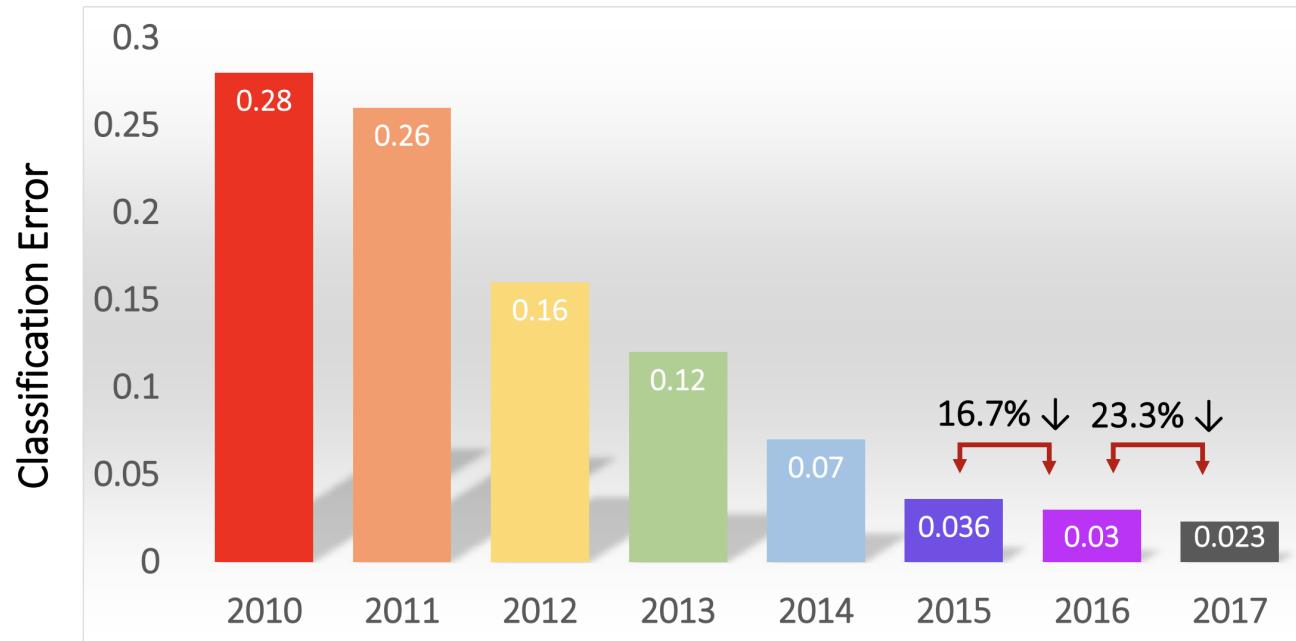
- Aprender representações dos dados em camadas. Aprender representações dos dados 'hierarquicamente'.



Fonte: Deep Learning with R, Chollet, F et al

# Por que Deep Learning

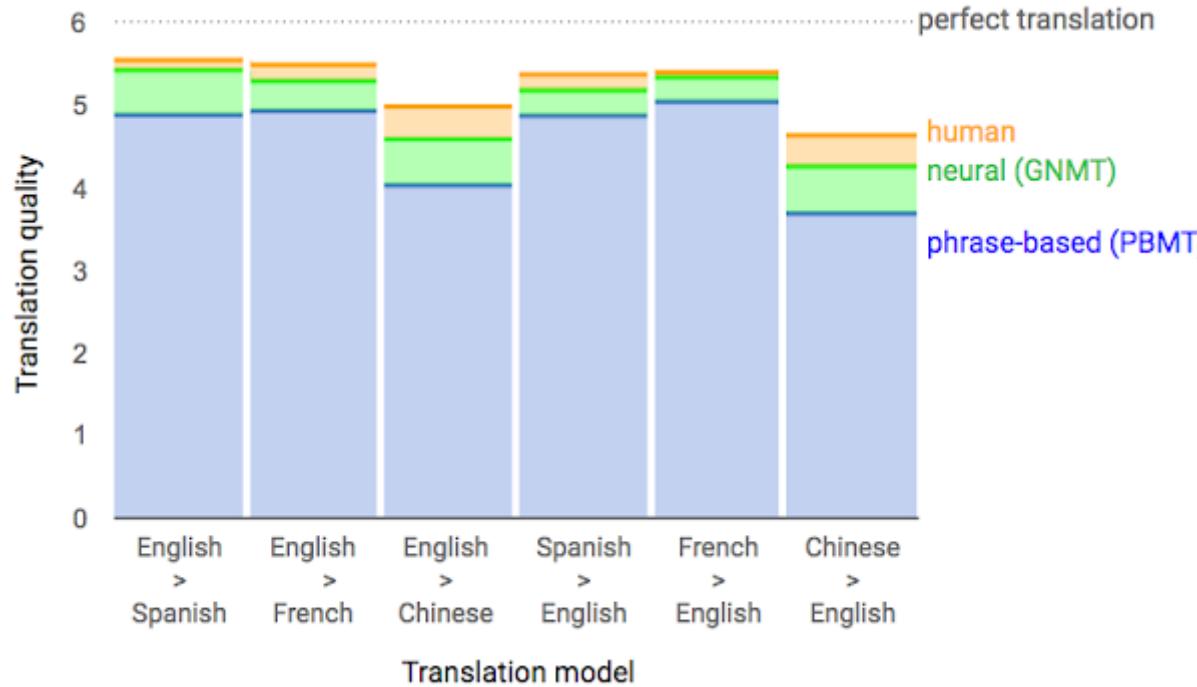
- Estado da arte em diversos problemas muito importantes.



Fonte: Apr. resultados ILSVRC 2017

# Por que Deep learning?

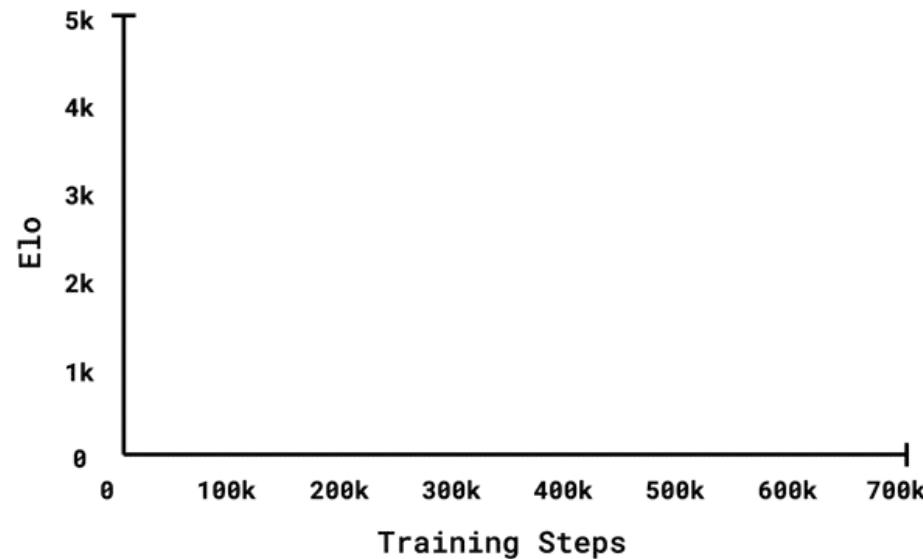
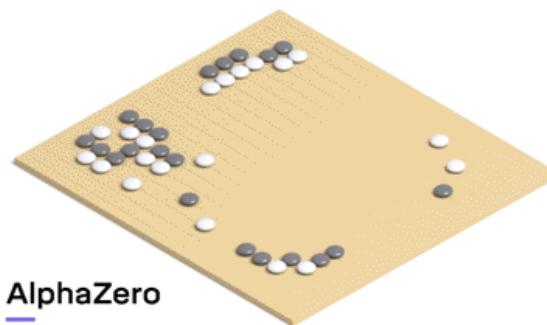
- Resultados impressionantes em tradução.



Fonte: A Neural Network for Machine Translation, at Production Scale

# Por que Deep Learning?

- Resultados em Reinforcement Learning



Fonte: AlphaZero: Shedding new light on chess, shogi, and Go

# Por que Deep Learning?

- Reconhecimento de fala
- Reconhecimento em vídeos



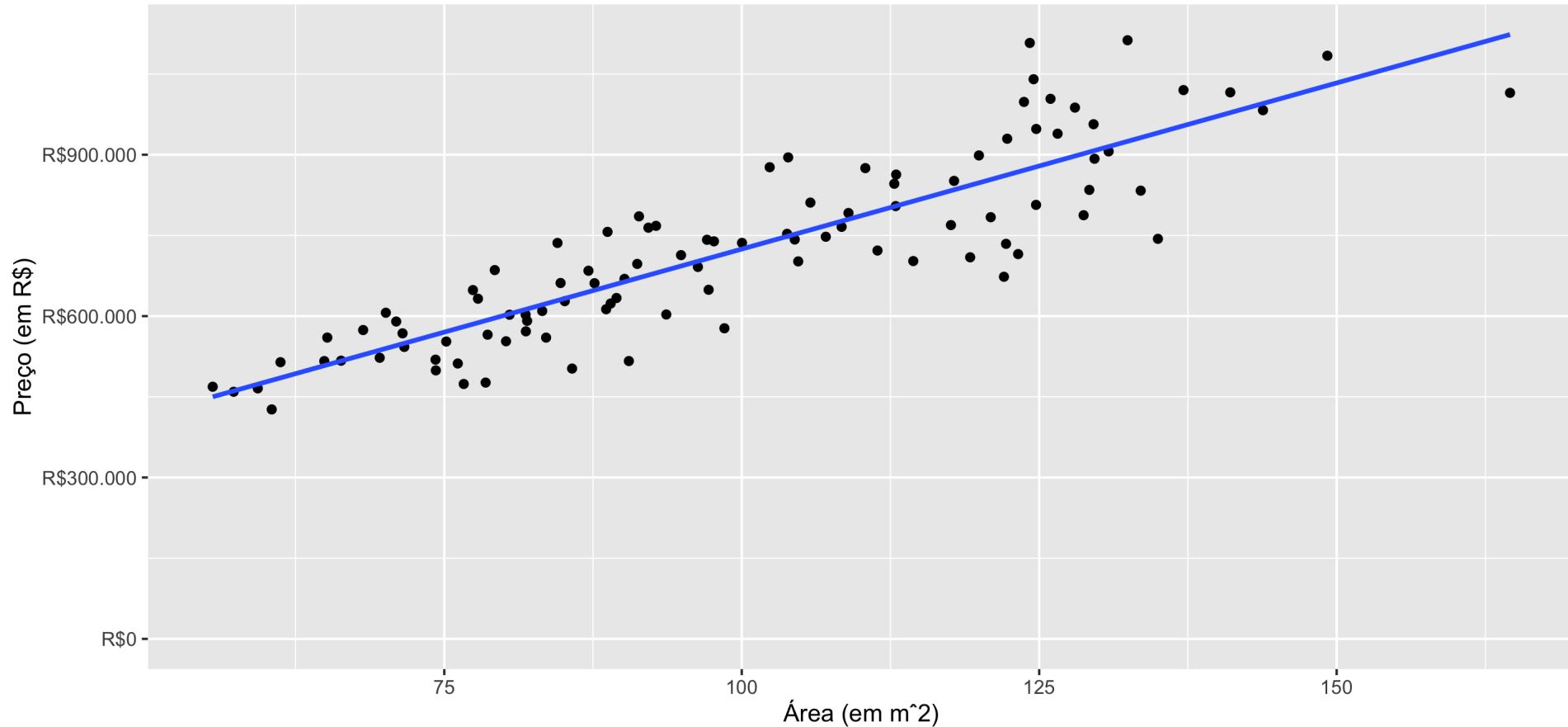
Fonte: [Tesla's Deep Learning at Scale: Using Billions of Miles to Train Neural Networks](#)

Tudo começa com . . .

Régressão Linear

# Régressão Linear

Prevendo o preço de uma casa a partir do tamanho em metros quadrados.



# Definição do modelo

Definimos o modelo de regressão linear da seguinte forma:

$$\hat{y}_i = w \times x_i + b$$

- $y_i$ : preço do imóvel  $i$
- $x_i$ : área do imóvel  $i$

Poderíamos escrever

$$\hat{y}_i = f(x_i)$$

em que:

- $f(x) = w \times x + b$

Chamamos  $f$  de 'layer' (camada) na linguagem do Deep learning.

# Layers (Camadas)

- Uma 'layer' é uma transformação dos dados que é parametrizada por **pesos**.
- 'Aprender', então, significa encontrar os melhores **pesos** para cada camada.

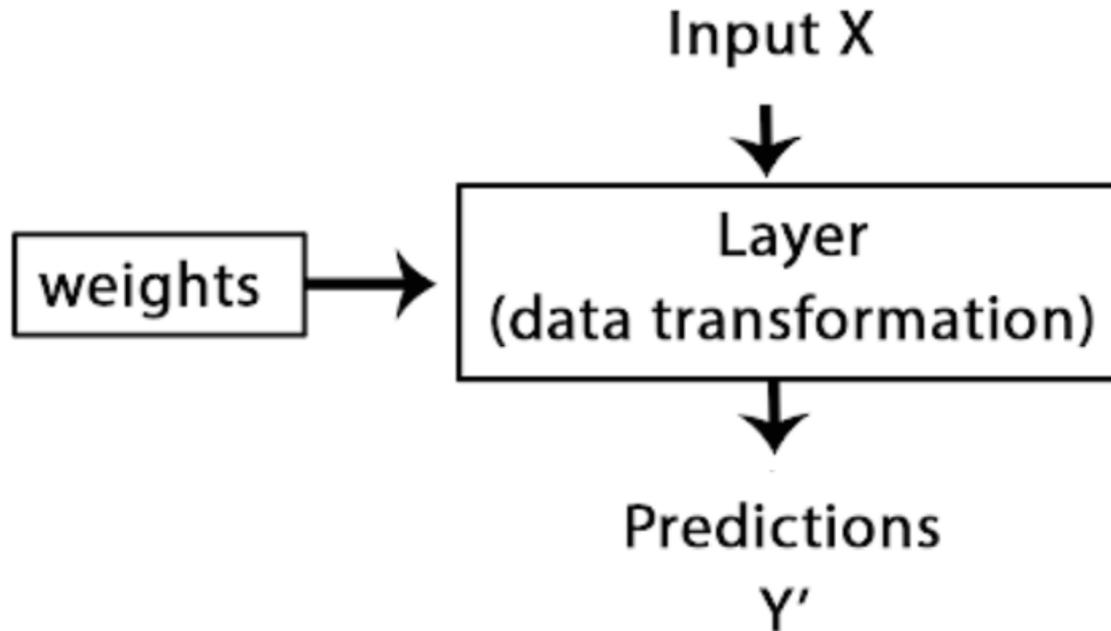
No exemplo:

$$f(x) = w \times x + b$$

Os pesos são  $w$  e  $b$ .

- Essas camadas são os 'tijolos' do Deep Learning e existem diversas 'camadas'.
- A camada do exemplo é chamada de '**Densa**' ou '**Linear**'.
- Um modelo pode possuir uma ou mais dessas camadas.

# Regressão Linear



Fonte: Figura adaptada do Deep Learning with R, Chollet, F. et al.

Objetivo: encontrar os melhores 'pesos' para essa Layer.

# Função de perda

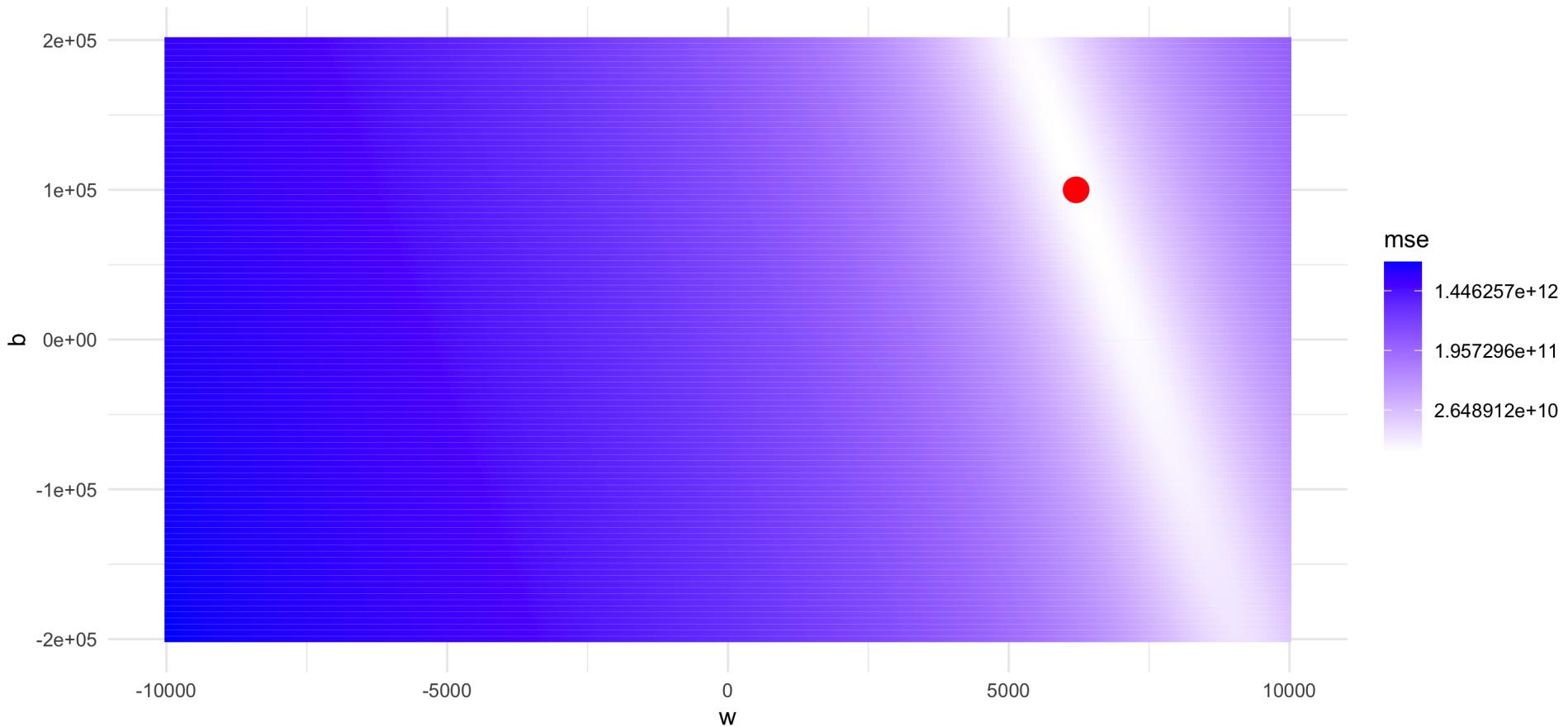
- Mede quanto o modelo está perto do que queremos que ele fique.
- No nosso caso, mede o quanto a previsão dada por  $w \times x + b$  está perto de  $y$ , o verdadeiro valor daquele imóvel.
- Uma função de perda bastante usada é o **MSE** - Erro quadrático médio.
- O MSE é dado por:

$$L(\hat{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

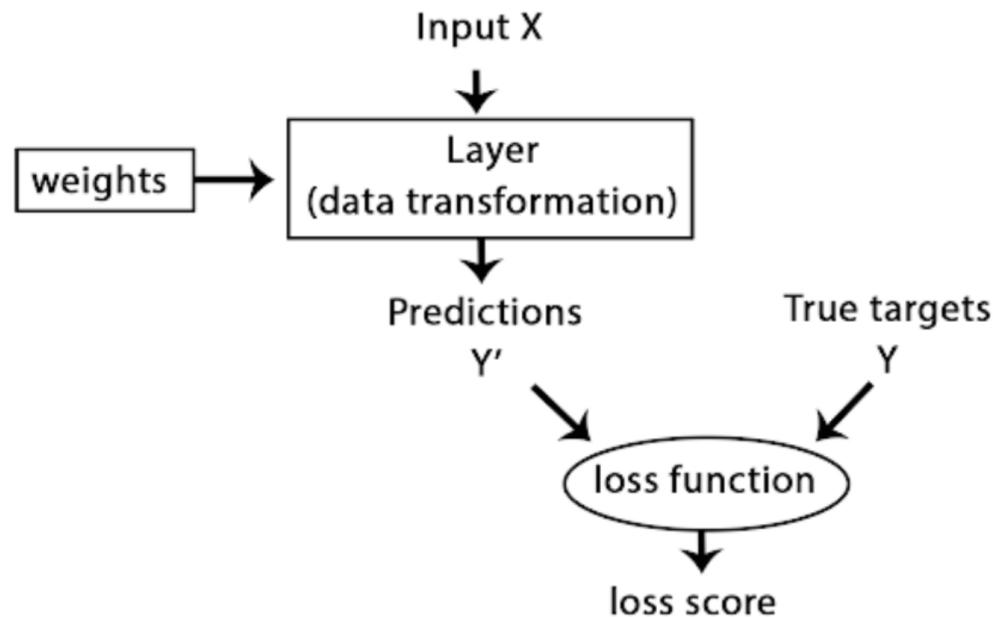
- Podemos reescrever em função dos pesos:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n (y_i - w \times x_i - b)^2$$

# Função de perda em função do valor dos parâmetros



# Função de perda



Fonte: Figura adaptada do Deep Learning with R, Chollet, F. et al.

- A função de perda nos diz, dado um valor para cada peso, o quão próximo estamos do valor esperado.

# Encontrando o mínimo da função de perda

Até agora:

- Vimos que nosso objetivo é minimizar a função de perda.
- Para isso precisamos encontrar o valor dos pesos que minimiza faz a função de perda ter o valor mínimo possível.

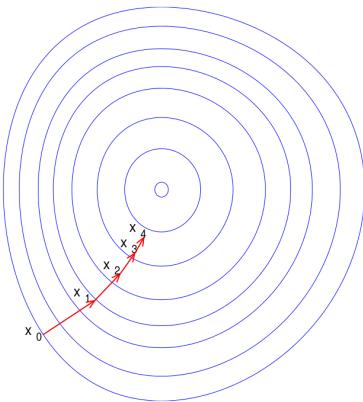
Agora:

- O processo de encontrar o mínimo de uma função é chamado de **otimização**.
- Existem diversos algoritmos de otimização. Em geral eles são adequados ou não dependendo da função que você está otimizando.
- Em Deep Learning usamos algoritmos que são variações do **Gradient Descent** - método de descida do gradiente.

# Gradient Descent

O **gradient descent** diz que se uma função  $L(x)$  é diferenciável na vizinhança de um ponto  $w$  então  $L(x)$  decresce mais rapidamente se você andar de  $w$  para uma direção contrária ao gradient de  $L$  no ponto  $w$ .

Em outras palavras, fazer  $w - \nabla L(w)$  é a forma de caminhar o mais rápido possível para o mínimo de  $L(x)$ .



Fonte: [Gradient Descent](#)

# Gradient descent no exemplo

No nosso exemplo temos:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n (y_i - w \times x_i - b)^2$$

Então para andar mais rápido para o mínimo de  $L$  cada passo de  $w$  e  $b$  tem que ser calculado da seguinte forma:

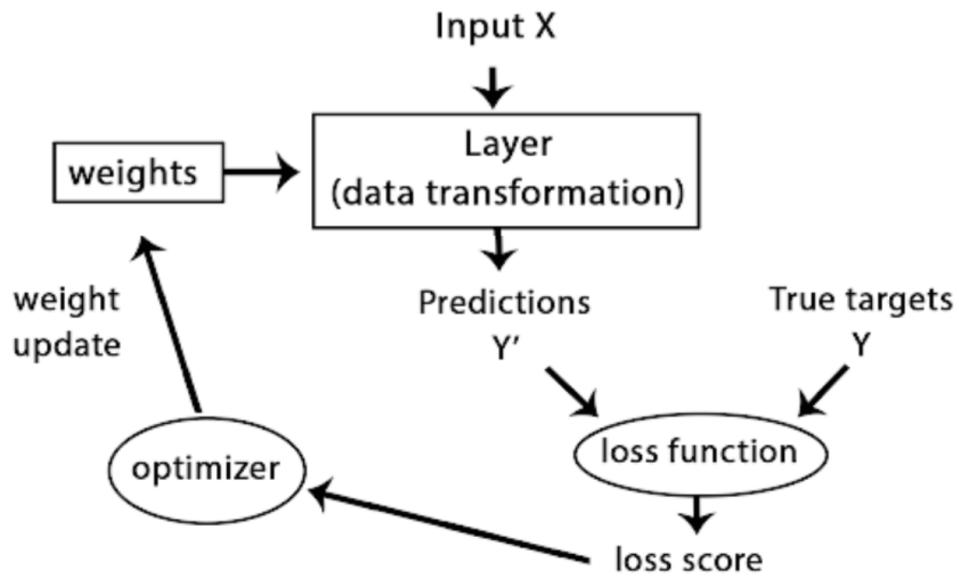
$$w_{(k+1)} = w_{(k)} - \alpha \frac{\partial L}{\partial w} = w_{(k)} - \alpha \frac{1}{n} \sum_{i=1}^n (-2 \times x_i)(y_i - w \times x_i - b)$$

$$b_{(k+1)} = b_{(k)} - \alpha \frac{\partial L}{\partial b} = b_{(k)} - \alpha \frac{1}{n} \sum_{i=1}^n (-2)(y_i - w \times x_i - b)$$

Calma! vamos dissecar essas fórmulas. Para a formulação mais formal ver capítulo 6.5 do [Deep Learning Book](#).

# Otimizando

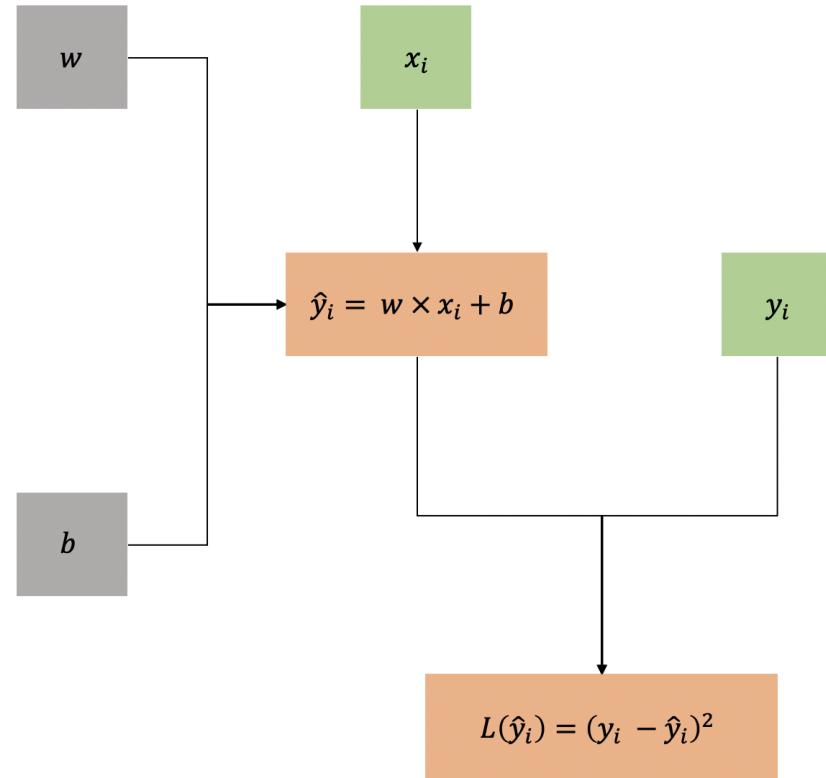
Esse é o diagrama geral que vale para os modelos que vamos implementar neste curso.



Fonte: Figura adaptada do Deep Learning with R, Chollet, F. et al.

# Grafo de computação

- É útil representar modelo em um grafo de computação.
- Cinza são os pesos.
- Verde são os dados.
- Laranja são variáveis derivadas de dados & pesos.



# Calculando as derivadas

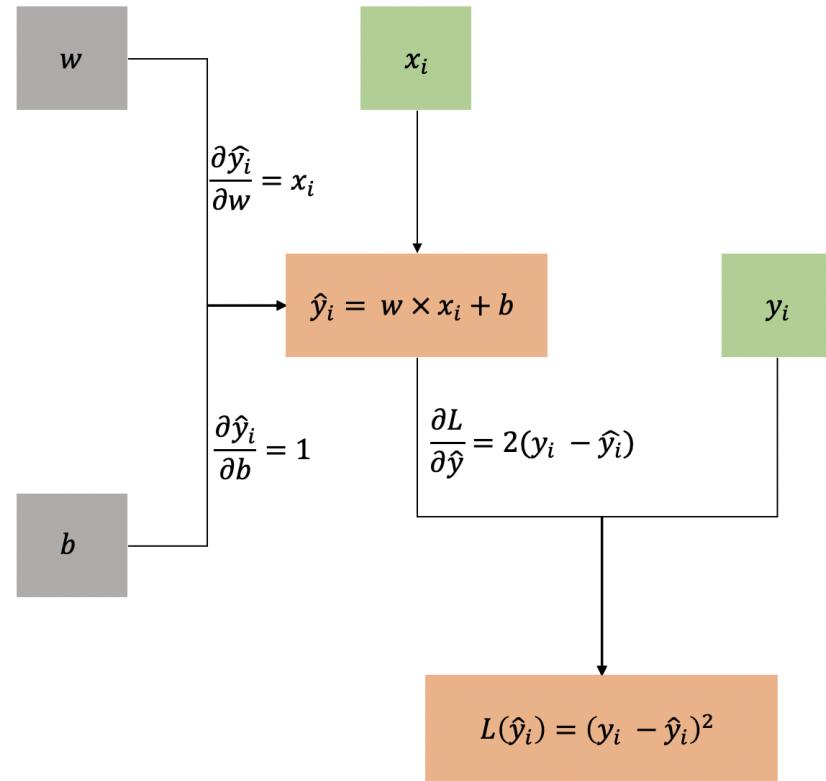
1. Calculamos as derivadas parciais para cada transformação.
2. Usamos a regra da cadeia para calcular as derivadas  $\frac{\partial L}{\partial w}$  e  $\frac{\partial L}{\partial b}$ .

Pela regra da cadeia temos:

$$\frac{\partial L}{\partial w} = 2(y_i - \hat{y}) \times x_i$$

$$\frac{\partial L}{\partial b} = 2(y_i - \hat{y}) \times 1$$

**Nota:** tiramos as médias p/ simplificar a notação.



# Gradient descent

No fim temos que a regra de atualização dos pesos é:

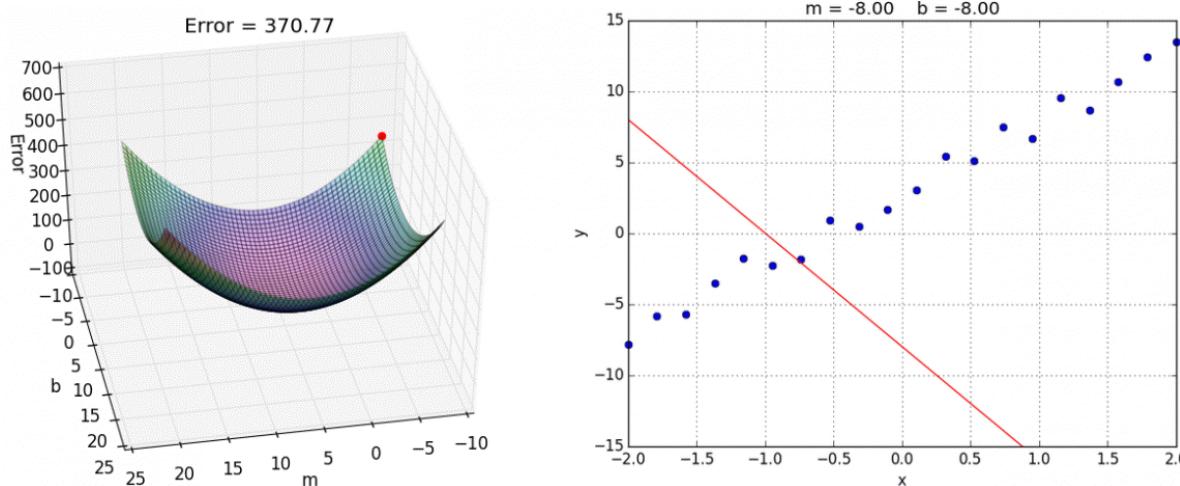
$$w_{(k+1)} = w_{(k)} - \alpha \times \frac{1}{n} \sum_{i=1}^n (2(y_i - \hat{y}) \times x_i)$$

$$b_{(k+1)} = b_{(k)} - \alpha \times \frac{1}{n} \sum_{i=1}^n (2(y_i - \hat{y}) \times 1)$$

- $\alpha$  é um hiper parâmetro que chamamos de '**learning rate**'. Ele controla com qual intensidade vamos andar na direção do gradiente.
- Na fórmula vemos que podemos obter  $w_{(i+1)}$  em função do  $w_{(i)}$ , mas e o  $w_{(0)}$ ? Geralmente inicializamos ele com algum número aleatório. A mesma coisa para o  $b_{(0)}$ .

# Gradient descent

- Esquerda eixos  $b$  e  $m$  representam  $b$  e  $w$  no nosso exemplo. O eixo 'Error' representa o valor da função de perda.
- Conseguimos visualizar a descida até o mínimo da função de perda pelo método do gradiente.
- Na **direita** a reta ajustada para os dados.



Fonte: [https://alykhantejani.github.io/images/gradient\\_descent\\_line\\_graph.gif](https://alykhantejani.github.io/images/gradient_descent_line_graph.gif)

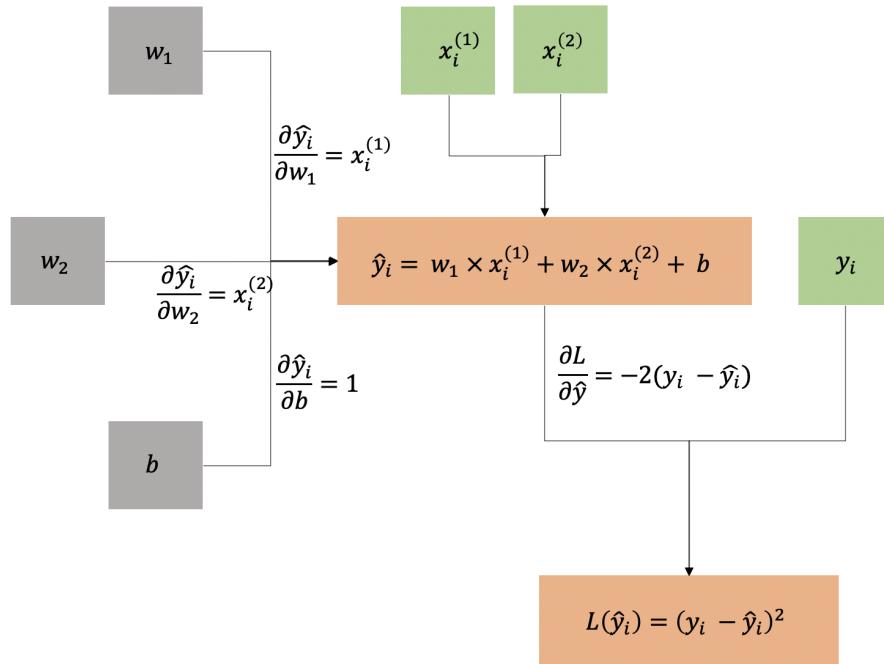
$x_1.$   
 $r_{11} = x^1$   
 $r_{12} \vdots y$

exemplo 01: exemplos/01-linear-regression.R

$$x + 2 = 10 \quad \underline{\underline{=}} \quad 40 \times 10 = 100$$

# Exercício 1

Arquivo: exercicios/01-linear-regression.R



# SGD (Stochastic gradient descent)

- Em vez de calcular a média da derivada em todos os exemplos da base de dados, calculamos em apenas 1 e já andamos. Isto é, trocamos:

$$w_{(k+1)} = w_{(k)} - \alpha \times \frac{1}{n} \sum_{i=1}^n (2(y_i - \hat{y}) \times x_i)$$

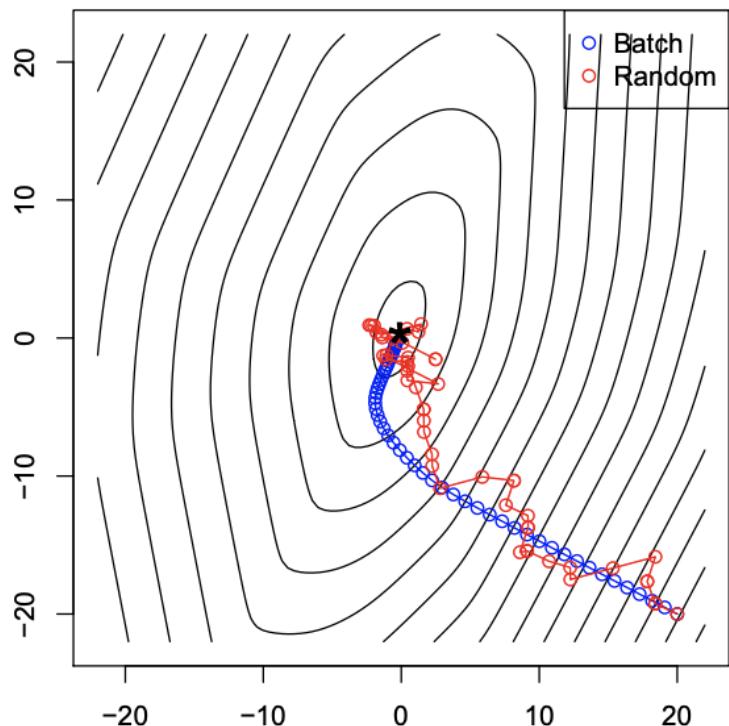
por

$$w_{(k+1)} = w_{(k)} - \alpha \times (2(y_i - \hat{y}) \times x_i)$$

- Cada vez que atravessamos a base inteira dessa forma chamamos de 'epoch'.
- Agora é possível atualizar os pesos sem precisar fazer contas na base inteira. Mais **rápido**.
- Como estimamos o passo com apenas uma observação, os passos (principalmente quando já estão perto do mínimo) podem ser meio ruins.

# SGD

- Na prática, parece que o fato dos passos serem ruins perto do mínimo é bom - isso faz um certo tipo de regularização. Não se sabe explicar esse comportamento muito bem ainda.



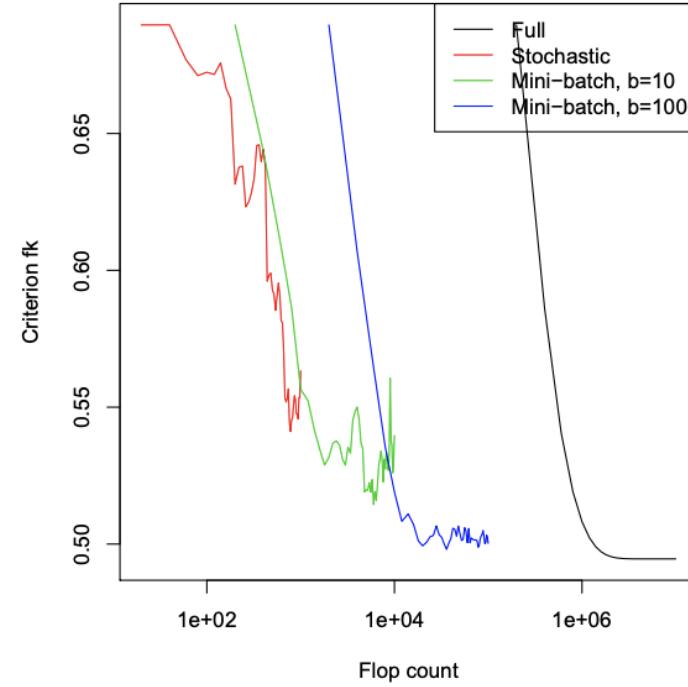
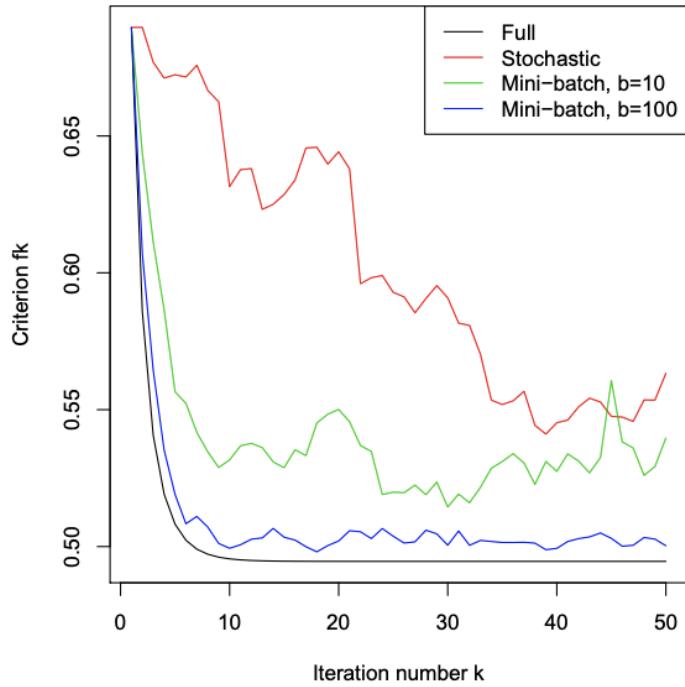
# Mini-batch SGD

- Em cada iteração, selecionamos uma amostra de tamanho  $b$  da base de tamanho  $n$ . Em geral  $b \ll n$ .
- Calculamos o passo do GD com essa amostra. No final temos:

$$w_{(k+1)} = w_{(k)} - \alpha \times \frac{1}{b} \sum_{i=1}^b (2(y_i - \hat{y}) \times x_i)$$

- Na prática é o que funciona melhor. Compensa o passo do SGD ser meio ruim por se basear em apenas uma observação e o fato do GD ser muito pesado computacionalmente para bases muito grandes.
- Em geral usa-se  $b$  (**batch size**) múltiplos de 2.

# Mini-batch SGD



Fonte: <https://www.stat.cmu.edu/~ryantibs/convexopt/lectures/stochastic-gd.pdf>

# Variações

- Existem outras variações do SGD, cada tentando resolver um problema diferente.
- Esse artigo é um ótimo resumo de todas as versões que existem.
- Não existe um que é extremamente melhor do que os outros.
- Além do SGD, os mais usados são SGD com momentum, adam e rmsprop.

exemplo 02: *exemplos/02-sgd.R*

exercicio 02: *exercicios/02-mini-batch-sgd.R*

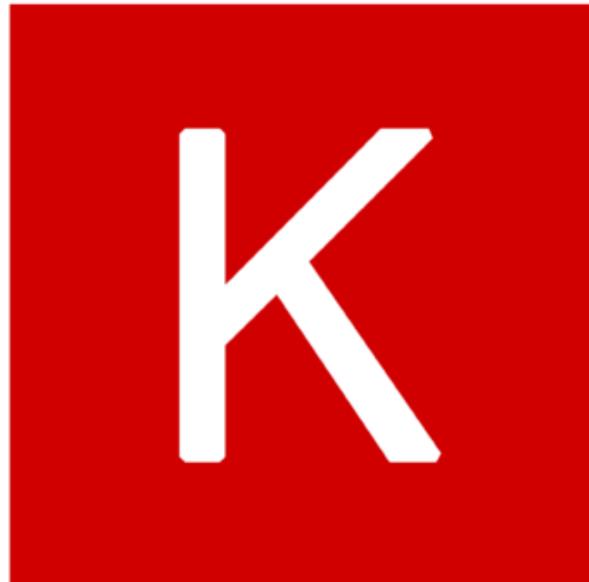
# TensorFlow

- Biblioteca open-source para cálculos numéricos.
- Desenvolvida inicialmente pela Google.
- Foco em Machine Learning e principalmente Deep Learning.
- Muito rápido - implementado para diversos hardwares como GPU's e até TPU's.
- Feature: **Automatic Differentiation!**
- Um grande ecossistema de add-ons e extensões.
- Biblioteca mais utilizada para fazer Deep Learning atualmente.



# Keras

- É uma biblioteca open-source criada para especificar modelos de Deep Learning
- Foi criada antes do TensorFlow existir
- Funciona com múltiplos 'backends' - exemplo Theano, CNTK e PlaidML
- Foi **incorporada pelo TensorFlow** e à partir do 2.0 é a forma recomendada de especificar modelos no TensorFlow



Keras

# TensorFlow e Keras no R

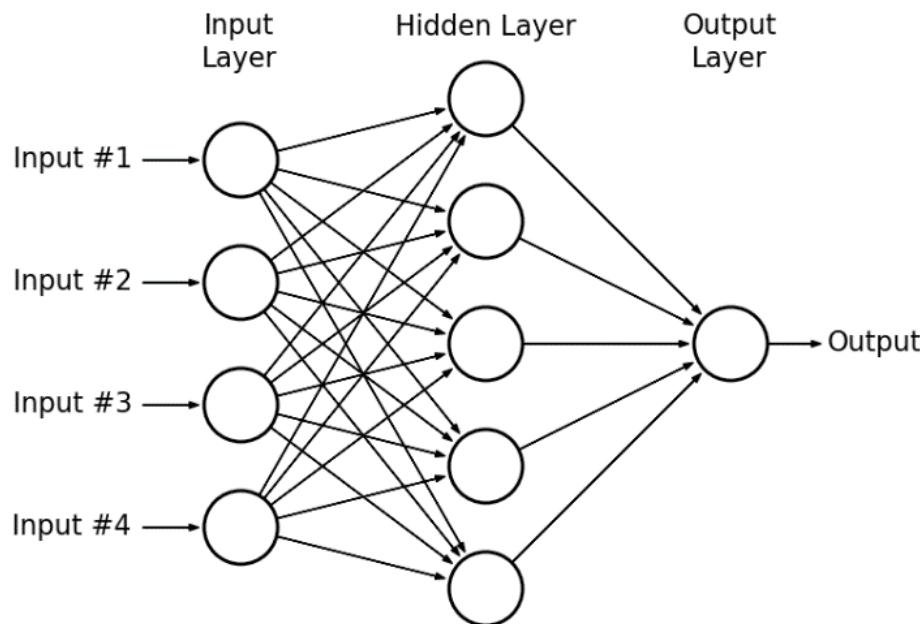
- TensorFlow e Keras são **implementados no R** usando o 'reticulate', isso significa que chamamos funções do Python.
- Keras para o R fornece uma API muito mais *user friendly* para quem já programa em R e quer aprender Deep Learning.
- Muitos guias e tutoriais [aqui](#).
- A performance é comparável com a do Python - as contas pesadas acontecem no C++.
- A comunidade é menor → tem seus prós e contras.

exemplo 03: exemplos/03-keras.R

exercício 03: exercícios/03-keras-linear-regression.R

# Multi-layer perceptron (MLP)

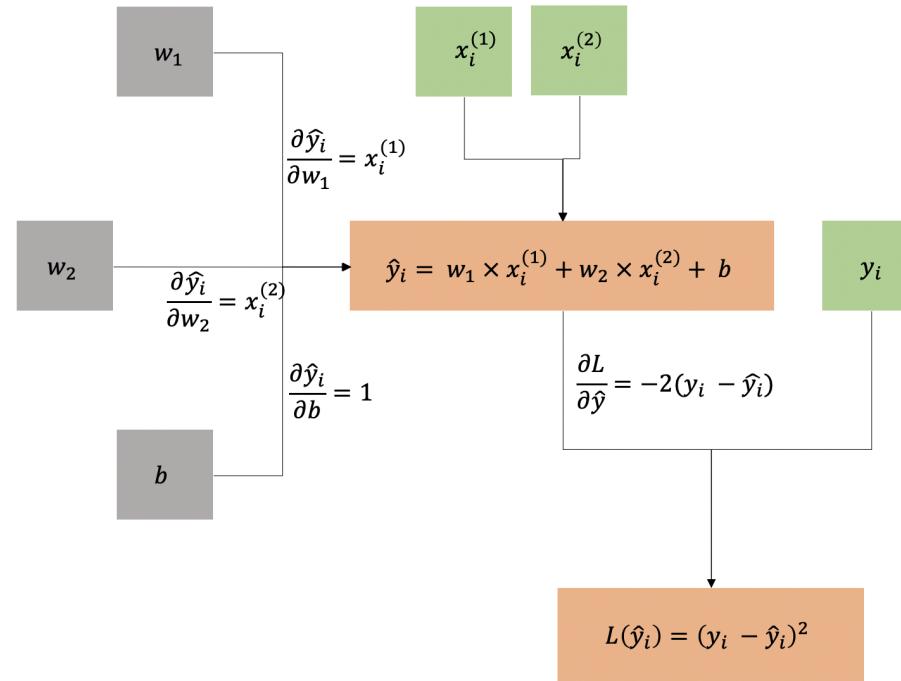
- Essa imagem é uma das primeiras a passar pela cabeça quando falam sobre redes neurais.



Fonte: [https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network\\_fig4\\_303875065](https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065)

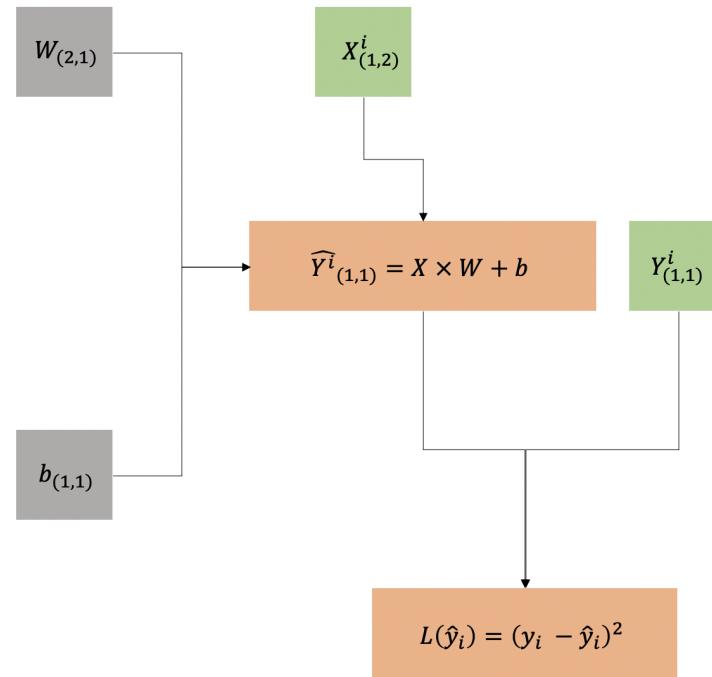
# Notação matricial

Escrever na forma escalar é mais fácil para exemplos mais simples mas, para exemplos um pouco mais avançados começa a ficar muito confusa.



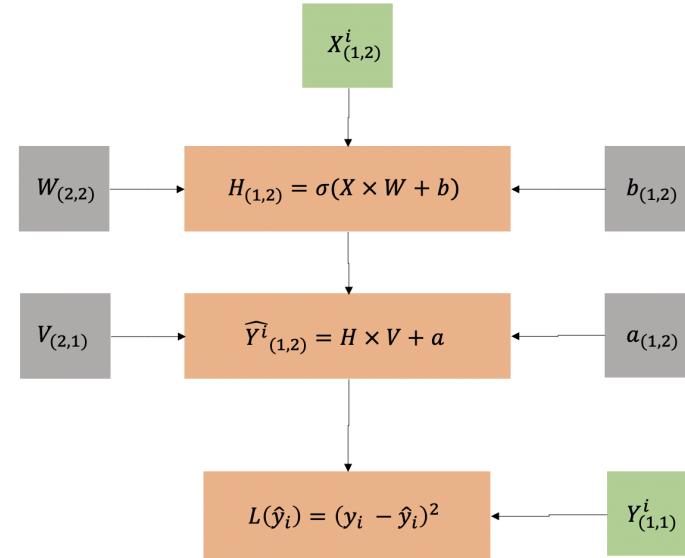
# Notação matricial

Re-escrevemos o modelo anterior usando a notação matricial. Veja que temos um pouco menos de caixinhas. Fica mais fácil de ler.



# MLP

- Ao lado temos a representação do grafo computacional de um MLP com uma camada escondida ('hidden layer') com 2 neurônios.
- $\sigma$  é uma função de ativação, reposável por adicionar não linearidades ao modelo.
- O número de colunas da matriz de pesos  $W$  é o número de 'hidden units'.
- A matrix  $H$  é o que chamamos de 'hidden units'.



# MLP

Vamos escrever a fórmula para calcular a previsão para uma observação  $i$ ,  $\hat{y}_i$ . Considere:

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

$$b = [ b_{11} \quad b_{12} ]$$

$$V = [ v_{11} \quad v_{12} ]$$

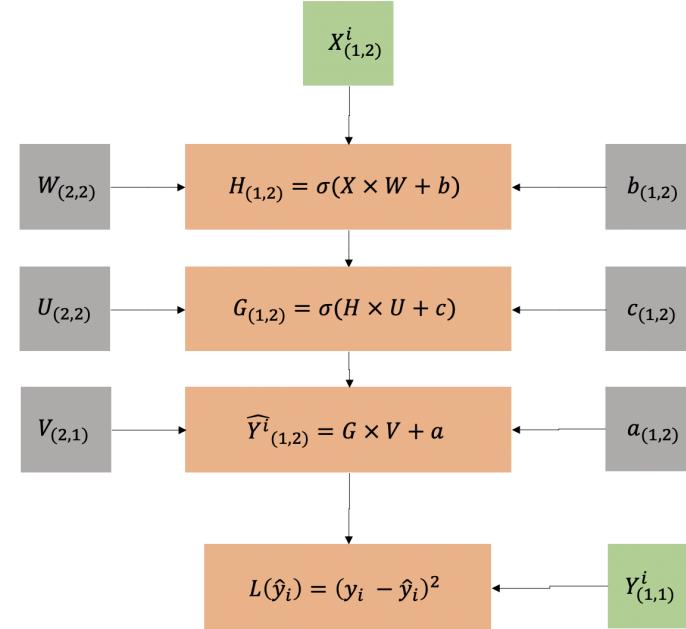
$$a = [ a_{11} ]$$

Então temos que

$$\hat{y}_i = v_{11}\sigma(w_{11} * x_{i1} + w_{21} * x_{i2} + b_{11}) + v_{12}\sigma(w_{12} * x_{i1} + w_{22} * x_{i2} + b_{12}) + a_{11}$$

# MLP

- Agora acrescentamos mais uma camada no nosso modelo.
- Cada camada pode ter uma função  $\sigma$  diferente.
- Cada camada pode ter um número de unidades diferente também.



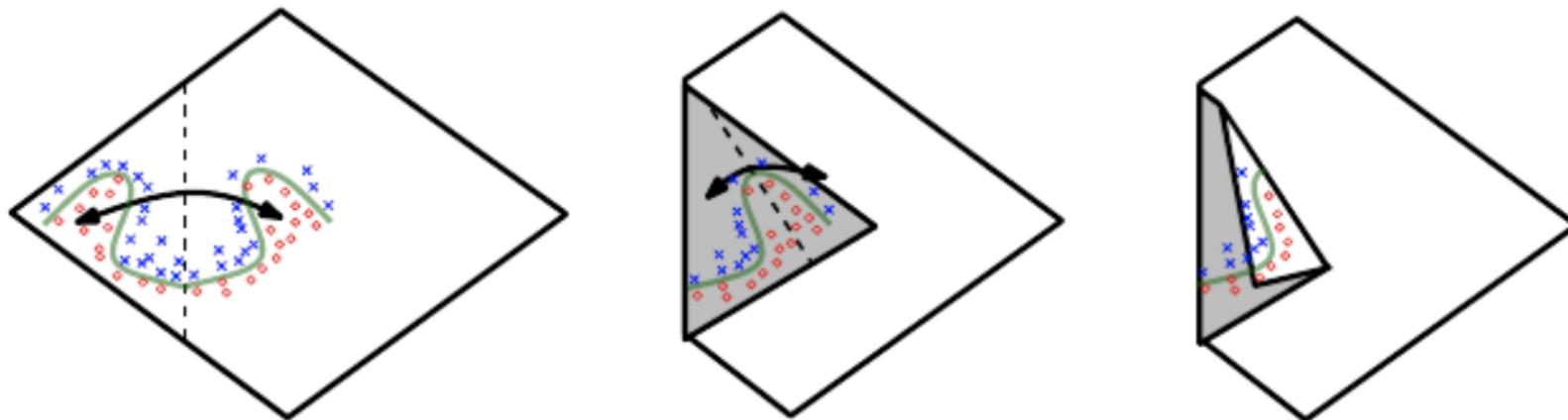
# MLP

A ideia de usar camadas surge da seguinte forma:

- Um modelo linear só pode, por definição, representar funções lineares.
- A seguir veio a ideia de criar modelos específicos para cada tipo de não linearidade que é estudada. Modelos exponenciais, modelos logísticos, etc.
- Teorema da aproximação universal (Hornik et al.) fala que um MLP com uma camada com uma função de ativação que achata as entradas, pode aproximar qualquer função desde que ela tenha um número suficiente de '**hidden units**'.
- Na prática isso não funciona muito bem porque o número de '**hidden units**' pode ter que ser muito grande - então esbarramos na quantidade de dados disponível.

# MLP

- A ideia a seguir foi usar várias camadas, ao invés de uma.
- Várias funções podem ser aproximadas com uma quantidade menor de *hidden units* se a **arquitetura** tiver mais camadas.



Fonte: Deep Learning Book - capítulo 6

# Hidden units

Existem duas escolhas com relação às 'hidden units':

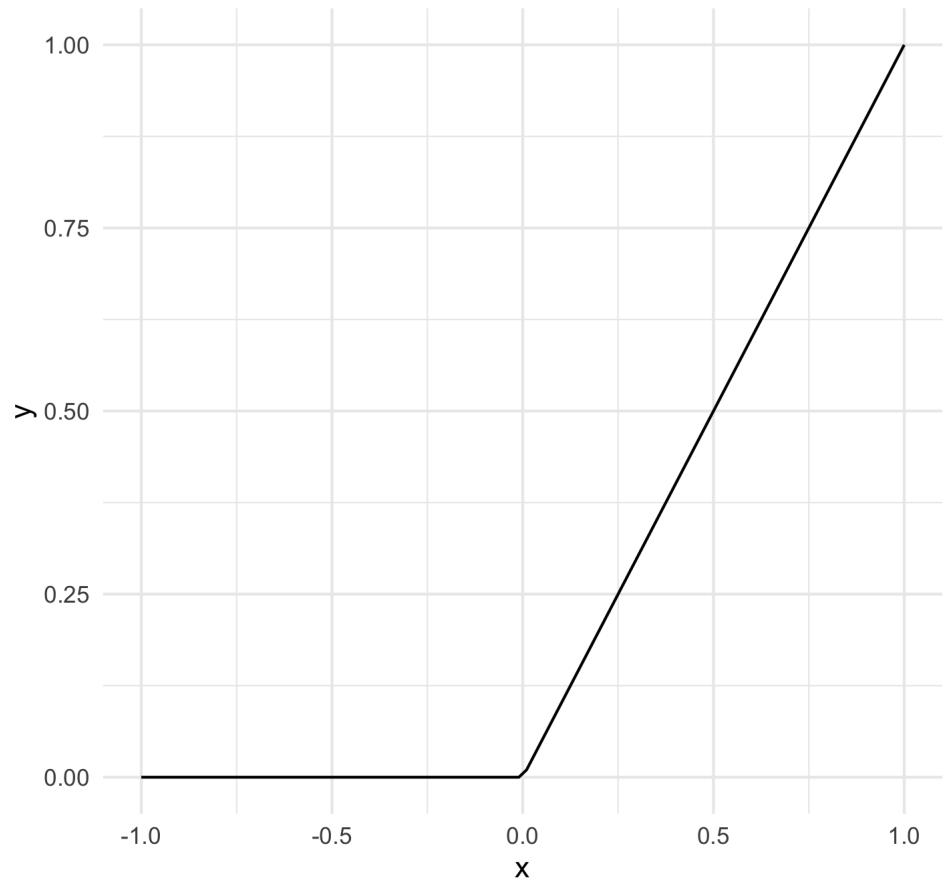
1. Qual é o número de hidden units? Em geral isso é uma função do tamanho da matrix  $X$ . Não existe uma regra de ouro para escolher isso, é preciso testar diversos tamanhos e ver qual fica melhor.
2. A **função de ativação** é a parte mais importante. Em geral **rectified linear units** são a primeira coisa para testar.

A seguir vamos conhecer algumas das funções de ativação mais utilizadas. Antes disso, vale a pena lembrar alguns pontos.

- As funções de ativação entram na conta do gradiente e portanto precisam ser deriváveis (pelo menos na maioria dos pontos).
- Descobrir qual função de ativação vai funcionar é também uma área de pesquisa ativa, e em geral é necessário testar várias versões para descobrir qual fica melhor.

# Rectified Linear Units (ReLU)

- Usamos a função de ativação  $\sigma(z) = \max\{0, z\}$ .
- A ideia do ReLU é ser fácil de otimizar, porque é muito parecido com simplesmente não ter ativação (ativação linear).
- As derivadas são altas quando a unidade é importante, isto é, quando o output é alto.
- A segunda derivada não tem muita influencia pois ou é zero ou um.
- Ver capítulo 6.3.1 do Deep Learning Book para outras extensões.

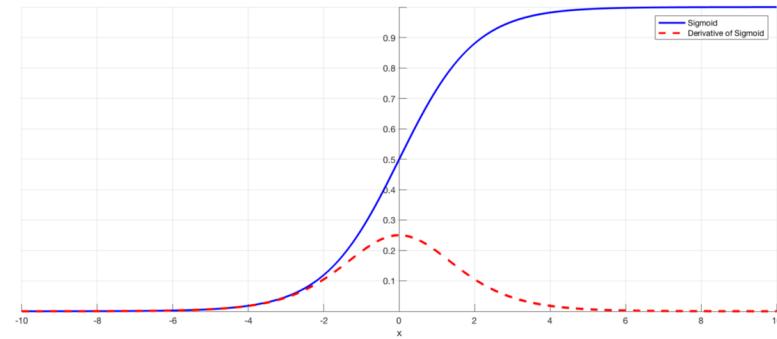


# Sigmoid

- A função sigmoid é dada por:

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

- Antes da introdução do ReLu, a maioria das redes neurais usava esse tipo de ativação.
- Diferentemente das unidades lineares, a derivada da sigmoid fica saturada quando o input é muito negativo ou muito positivo.
- Hoje em dia não se recomenda usá-la como função de ativação em camadas escondidas ('hidden layers').



Fonte:

<https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e>

exemplo 04: exemplos/mlp.R

exercicio 04: *exercicios/04-boston-housing.R*

# Régressão logística

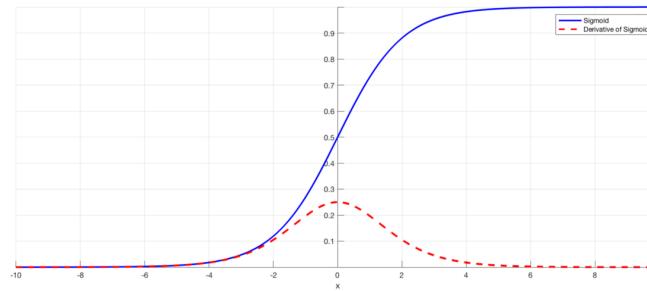
- Até agora só falamos de redes neurais que podem retornar números em qualquer intervalo. Isto é, quando fazemos  $\hat{y} = w * x + b$ ,  $w$  e  $b$  podem assumir quaisquer valores e portanto podem fazer com que  $\hat{y}$  seja qualquer número de  $-\infty$  até  $+\infty$ .
- Em problemas de classificação, por exemplo, não queremos que  $\hat{y}$  seja qualquer valor, e sim a probabilidade de  $y$  ser de uma determinada classe. Temos então 2 pontos:
  1. Queremos que  $\hat{y}$  seja um número entre 0 e 1.
  2. Como queremos que o output  $\hat{y}$  seja uma probabilidade, não queremos minimizar o MSE e sim uma outra função de perda que leve à essa interpretação. Além disso, o sigmoid satura muito rápido, então precisamos de uma função de perda que arrume isso.

Vamos ver como resolver esses dois problemas.

# $\hat{y}$ precisa ser um número entre 0 e 1

- Para resolver esse problema basta usar uma função de ativação na última camada que transforme o intervalo  $]-\infty, \infty[$  no intervalo  $[0, 1]$ .
- Uma função famosa por isso, e que já conhecemos é a função sigmoid.

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



Fonte: <https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e>

# A função de perda

Em geral usamos a função de perda que chamamos de '**cross entropy**'.

Essa função é dada por:

$$L(\hat{y}) = \sum_{i=1}^n [y_i \times \log \hat{y}_i + (1 - y_i) \times \log(1 - \hat{y}_i)]$$

Isso fica mais claro quando lemos o seguinte código:

```
cross_entropy ← function(y_hat) {  
  if (y = 1) -log(y_hat)  
  else -log(-y_hat)  
}
```

Isso equivale a estimativa de máxima verossimilhança quando assumimos que  $y$  tem distribuição  $Bernoulli(\hat{y})$ .

exemplo 05: exemplos/05-regressao-logística.R

exercício 05: exercícios/05-boston-housing-logistic.R

# Outras funções de perda

- Existem muitas outras funções de perda que podem ser utilizadas.
- Dependendo do problema você pode preferir uma ou outra.

Alguns exemplos:

- MAE - mean absolute error - para resposta numérica quando você não quer dar tanto peso para os outliers.
- Categorical cross-entropy: é uma generalização da binary cross-entropy quando você tem mais do que duas classes.
- Huber Loss: variação do mse um pouco menos sensível aos outliers.

# Mais de duas categorias

- Quando temos um problema de predição de muitas categorias, a nossa resposta é uma matriz, por exemplo:

```
## # A tibble: 3 x 3
##   banana  maçã laranja
##   <dbl> <dbl>    <dbl>
## 1     0     1        0
## 2     1     0        1
## 3     0     0        1
```

- O número de colunas igual ao número de categorias possíveis.
- O número de linhas é o número de observações da base.
- Os valores são 0 quando a observação não é daquela categoria e 1 quando é da categoria.

# Mais de duas categorias

Queremos que o nosso modelo retorne uma matriz de probabilidades, por exemplo:

```
## # A tibble: 3 x 3
##   banana  maçã laranja
##   <dbl> <dbl>    <dbl>
## 1     0.2     0.7     0.1
## 2     0.6     0.1     0.3
## 3     0.1     0.3     0.6
```

- Veja que as linhas somam 1.
- Para isso usamos a função de ativação 'softmax'.

Seja  $x = (x_1, x_2, \dots, x_k)$  então:

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{i=1}^n e^{x_i}}$$

# Mais de duas categorias

Também precisamos modificar a função de perda. Ao invés de usarmos a função `binary crossentropy`, usamos a função `categorical crossentropy`.

A `binary crossentropy` era dada assim:

```
cross_entropy ← function(y_hat) {  
  if (y = 1) -log(y_hat)  
  else -log(-y_hat)  
}
```

A `categorical cross entropy` é dada por (considere que `y_hat` agora é uma matriz em que o numero de colunas é o número de classes).

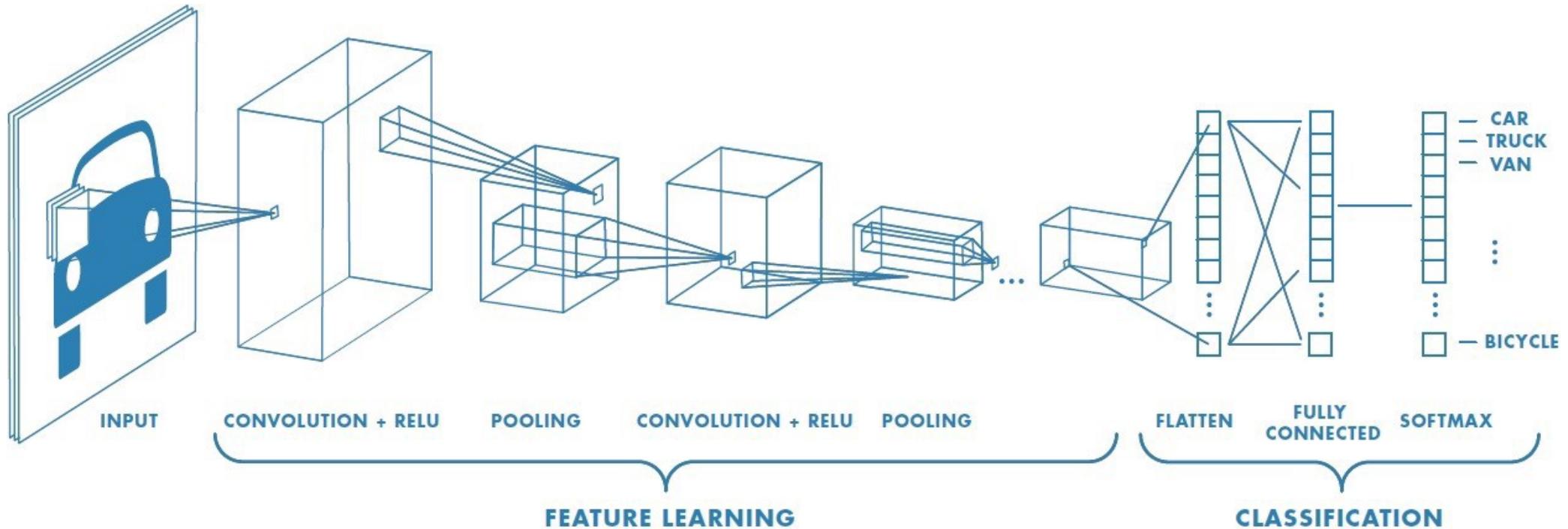
```
cross_entropy ← function(y_hat) {  
  if (y = 1) -log(y_hat[,1])  
  else if (y = 2) - log(y_hat[,2])  
  ...  
  else if (y = k) - log(y_hat[,k])  
}
```

exercicio 06: *exercicios/06-mlp-mnist.R*

# Convolutional Neural Networks (CNN)

# CNN's

- É uma arquitetura de redes neurais que é útil principalmente para classificação de imagens.



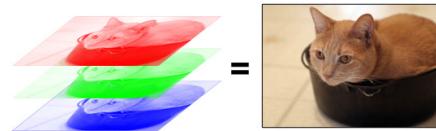
Fonte: Introduction to CNN's

# Imagens como dados

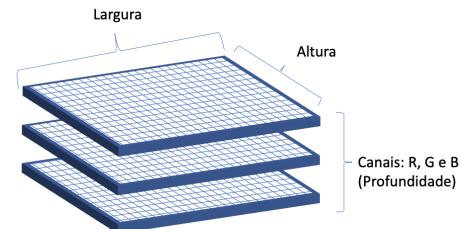
- Uma imagem em preto e branco (sem cinza) é uma matriz de 0's e 1's. Algo dessa forma:

# Imagens como dados

- Imagens coloridas são representadas como um array de 3 dimensões.
- É como se fosse um 'empilhado' de 3 matrizes.
- Cada elemento é a intensidade de cada cor daquele píxel.

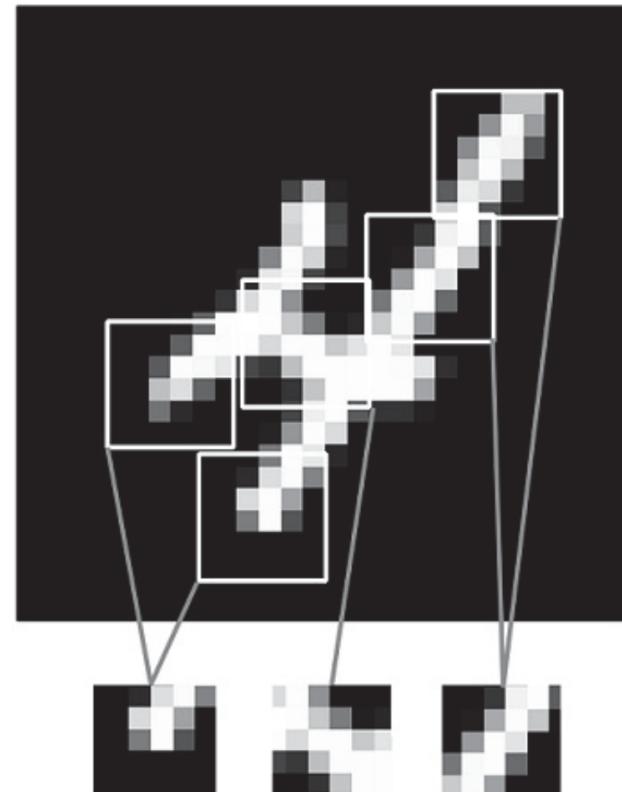


Fonte: [Create a Retro CRT Distortion Effect Using RGB Shifting](#)



# CNN's

A principal diferença com relação à MLP é que as camadas 'densas' aprendem padrões globais dos inputs, enquanto convoluções aprendem padrões *locais* dos inputs.

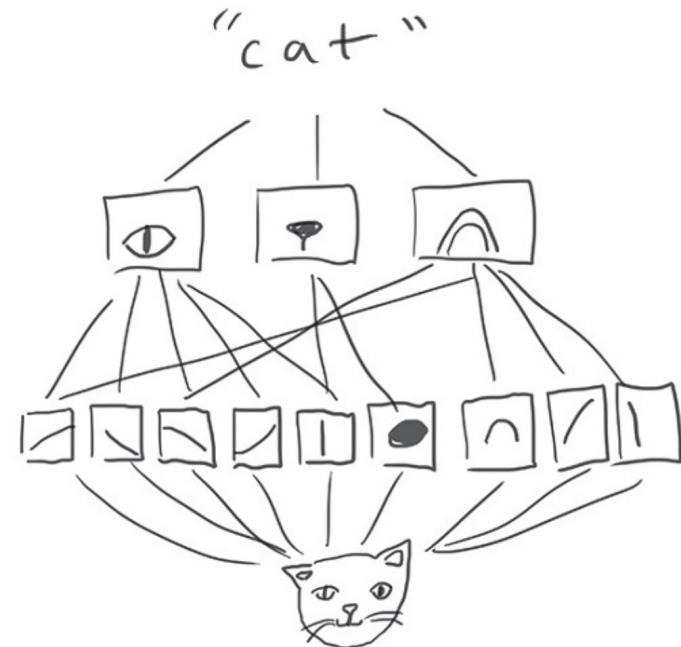


Fonte: Deep Learning with R

# CNN's

Essa característica, de aprender padrões locais, permitem duas importantes propriedades:

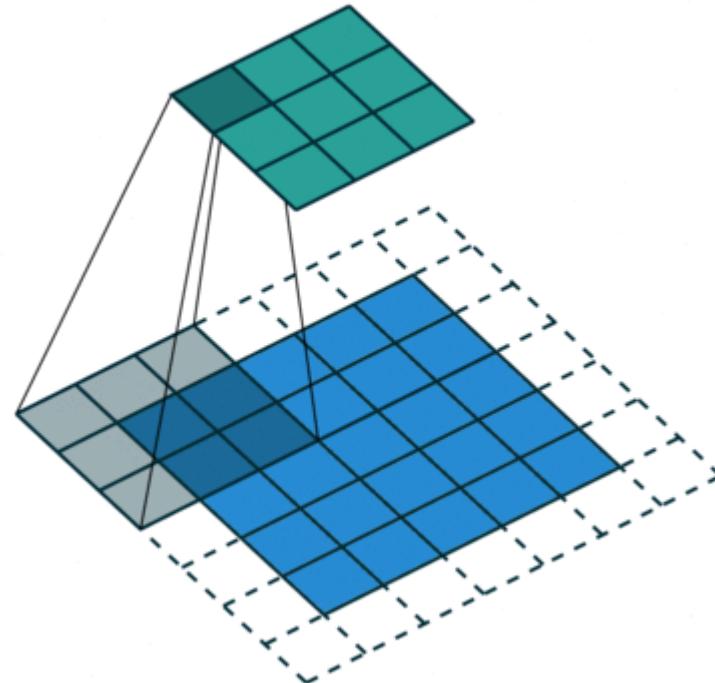
- **translation invariant**: um padrão pode ser encontrado mesmo que ele apareça no canto em uma imagem e no centro em outra.
- **hierarquia espacial de padrões**: em um primeiro momento a CNN aprende a identificar cantos, linhas. Mais para frente aprende padrões um pouco maiores e mais complexos.



Fonte: Deep Learning with R

# Uma convolução

- Definimos uma matriz de pesos (em cinza na representação ao lado)
- Andamos com essa matriz de pesos para cada parte da imagem (em azul ao lado).
- Esses pesos são multiplicados e depois somados para gerar uma nova 'imagem' (em verde).

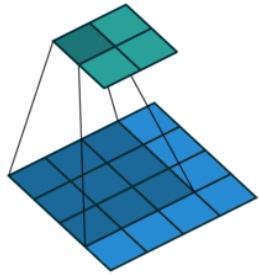


Fonte: [Conv arithmetic](#)

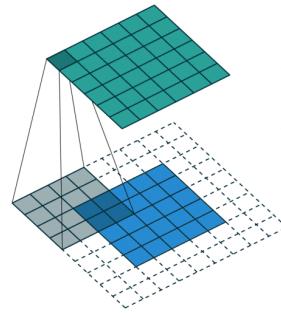
exemplo 06: convolution.R

# Outros tipos de padding/strides

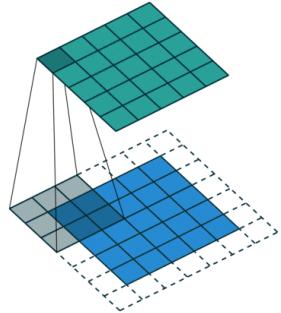
---



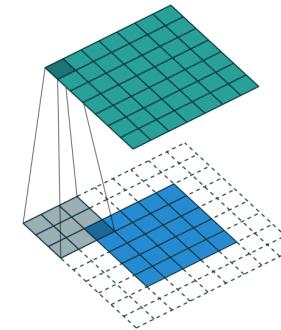
No padding, no  
strides



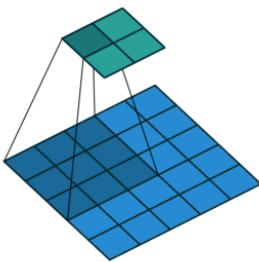
Arbitrary padding,  
no strides



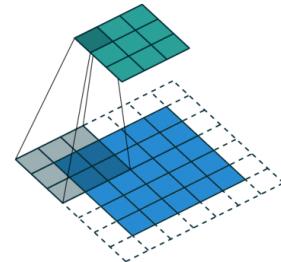
Half padding, no  
strides



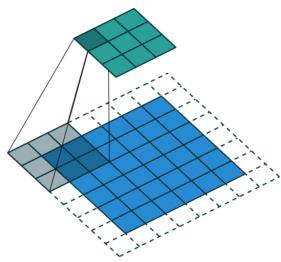
Full padding, no  
strides



No padding,  
strides



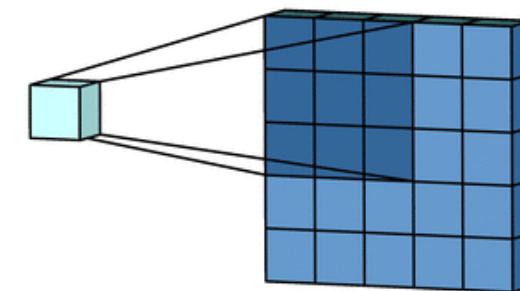
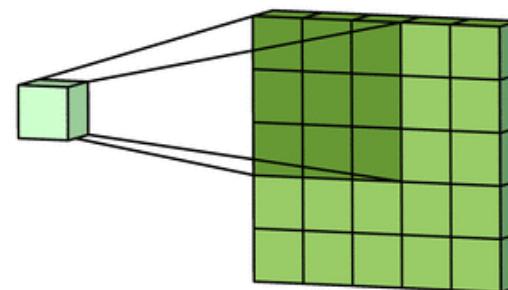
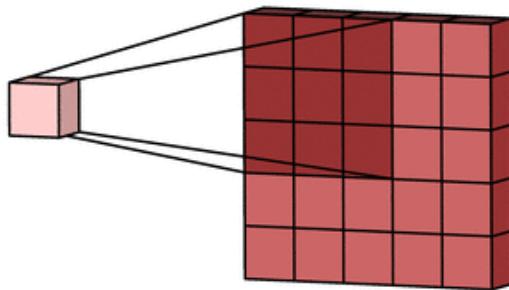
Padding, strides



Padding, strides  
(odd)

# CNN's

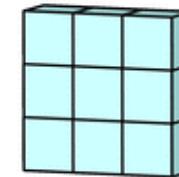
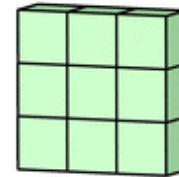
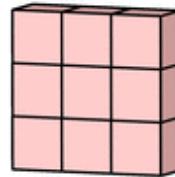
Primeiro temos um 'kernel' (matriz de parâmetros p/ cada canal):



Fonte: *Intuitively Understanding Convolutions for Deep Learning*

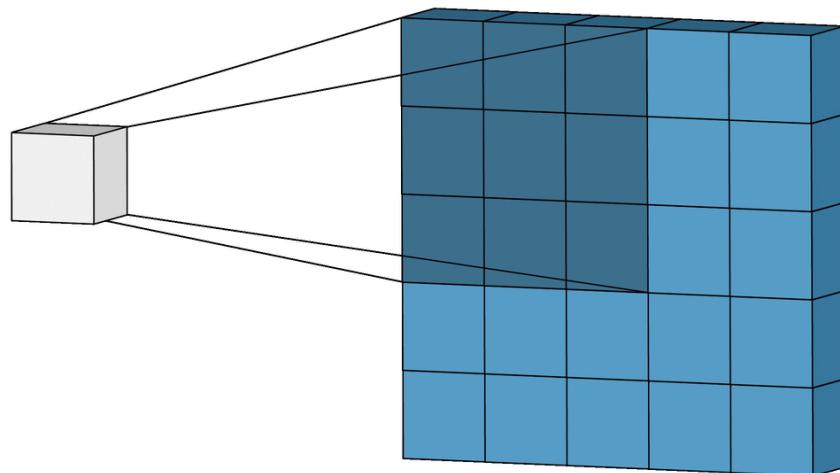
# CNN's

Em seguida somamos os outputs de cada canal:



# Max Pool

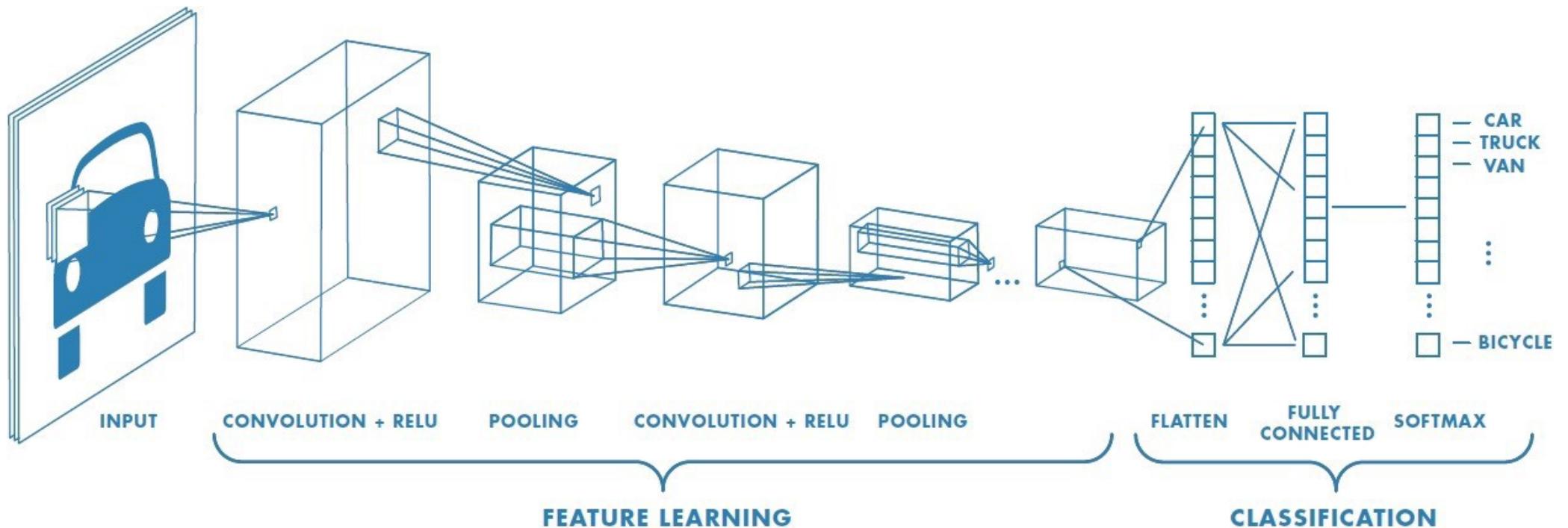
Parece com o que fazemos na convolução, mas em vez disso calculamos o máximo dos valores de cada janela.



Fonte: [Intuitively Understanding Convolutions for Deep Learning](#)

# Resumo:

- Mesclamos algumas camadas de convolução e max pooling, diminuindo a altura e largura das imagens e aumentando a profundidade.
- Depois transformamos em uma matriz e fazemos um modelo de classificação logístico usual.

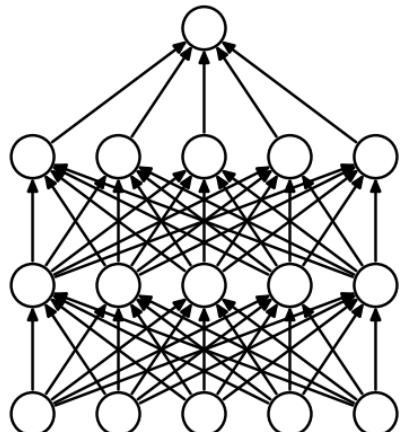


exemplo 07 - exemplos/07-conv-mnist.R

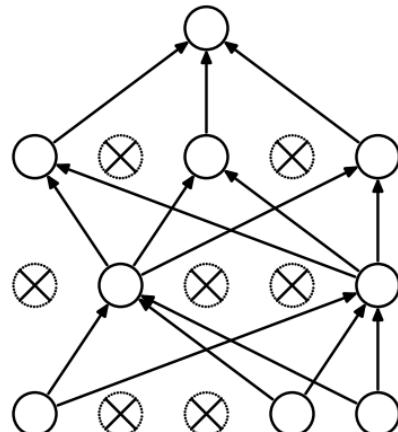
exercício 07: exercícios/07-cifar-conv.R

# Dropout

- Técnica de regularização bastante utilizada em deep learning.
- Consiste em aleatoriamente zerar alguns outputs.
- É parametrizado por uma probabilidade  $p$  de zerar alguns parâmetros.



(a) Standard Neural Net



(b) After applying dropout.

Fonte: Dropout: A Simple Way to Prevent Neural Networks from Overfitting

# Batch normalization

- Outra técnica que ajuda bastante a ajustar modelos em deep learning.
- Consiste em normalizar os valores das 'hidden units'.
- Em geral usamos Batch Norm antes da ativação.

Ver: [https://www.youtube.com/watch?v=tNIPeZLv\\_eg](https://www.youtube.com/watch?v=tNIPeZLv_eg)

<https://www.youtube.com/watch?v=nUUqwaxLnWs>

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

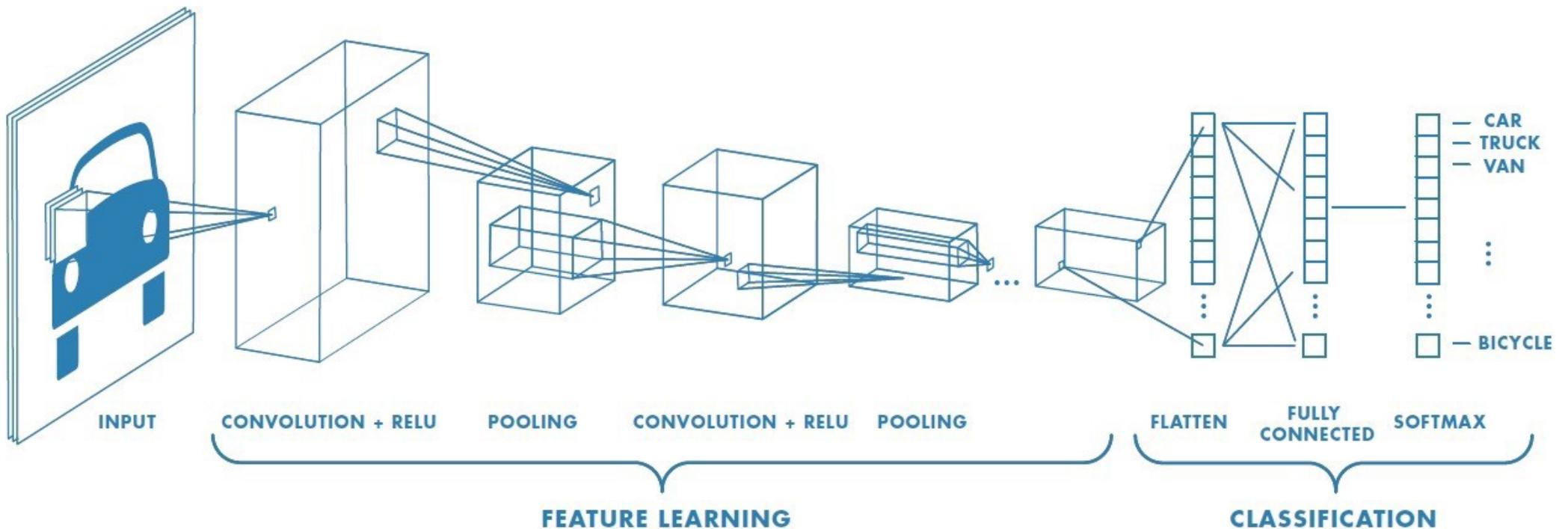
**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

**Fonte:** Batch Normalization:  
Accelerating Deep Network Training  
by Reducing Internal Covariate Shift

exercício 08: exercícios/08-dropout-batch\_norm.R

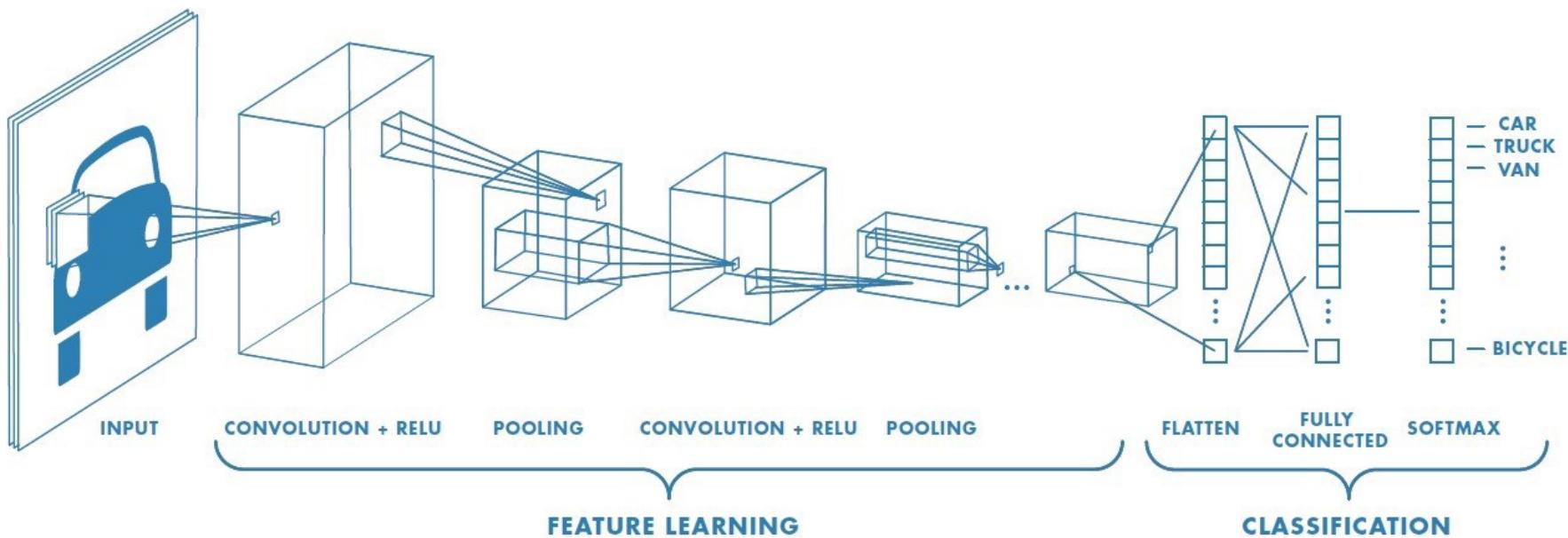
# Modelos pré-treinados

- Reutilizar a parte 'feature learning' do diagrama abaixo em outros bancos de dados.
- O conceito não vale apenas p/ imagens mas para qualquer modelo em deep learning.



# Modelos pré-treinados

- Um modelo inteiro é treinado em um banco de dados X.
- Salvamos apenas a parte 'feature learning'.
- Em um outro banco de dados usamos as 'features' que foram aprendidas no banco de dados X e apenas ajustamos a parte de classificar para um outro banco de dados.



# Modelos pré-treinados

As vantagens são:

- Reduz muito o tempo para treinar um modelo.
- Faz com que seja possível treinar modelos em bases menores.

Desvantagens:

- Tem que tomar cuidado com a base em que eles foram treinados inicialmente.

# tfhub

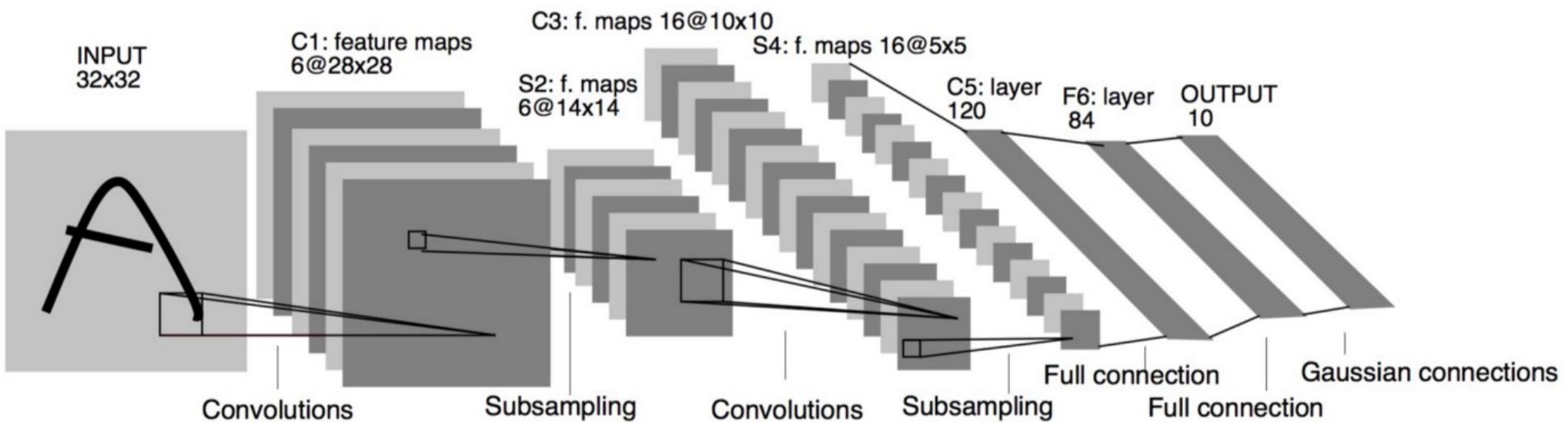
- Um pacote de R para fazer transfer learning. Possibilita usar modelos pré-treinados disponíveis no [tfhub.dev](https://tfhub.dev).
- É fácil de usar! Só mais uma 'camada' do Keras

```
url ← "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/2"  
layer_hub(handle = url)
```

exemplo 08: exemplos/tfhub.R

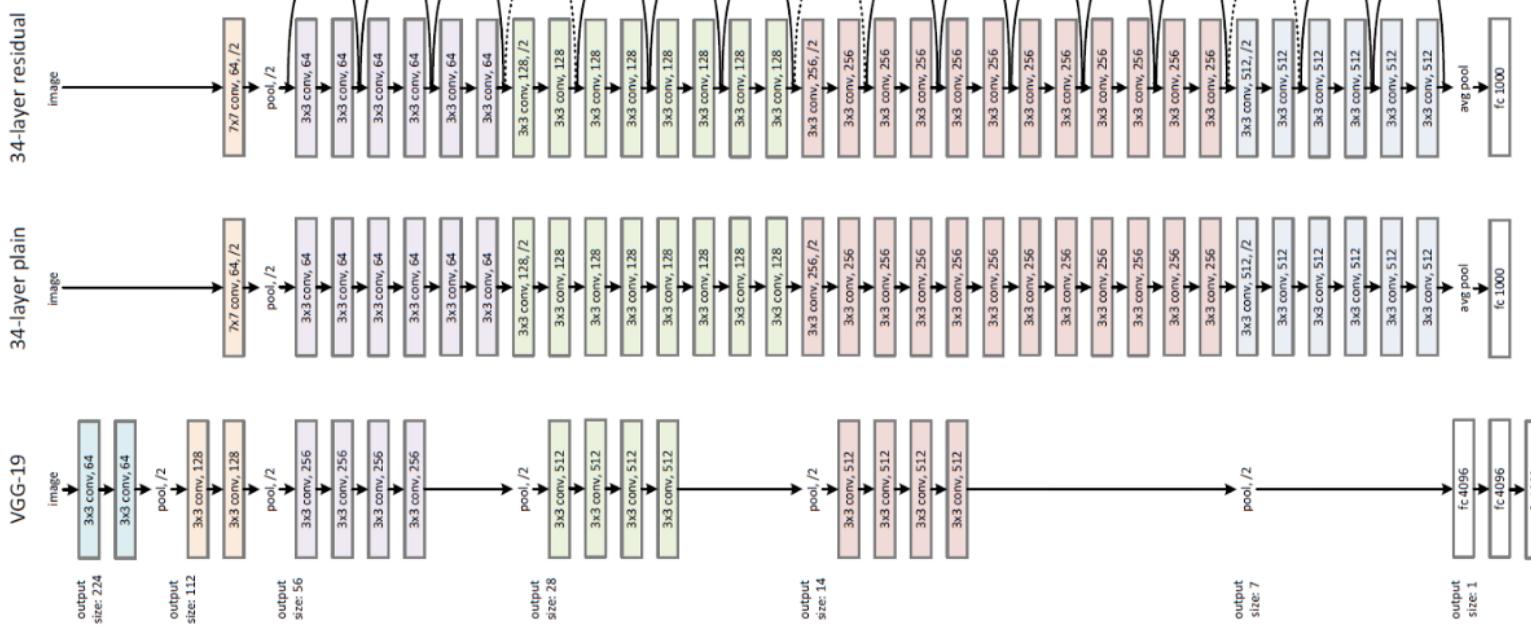
# LeNet5

- 1994 !



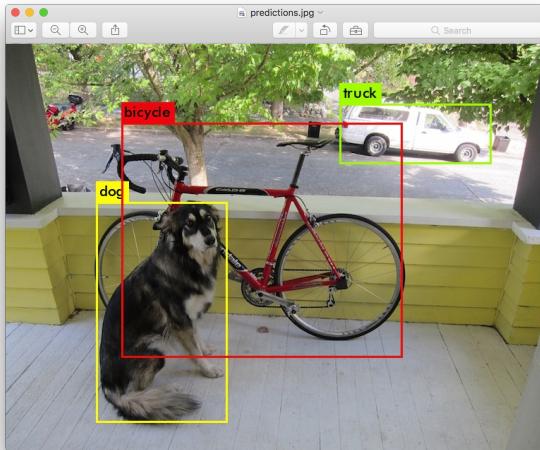
# ResNets

- 2015
  - passar os inputs para as camadas da frente.



# Outros tópicos

- Image Segmentation: segmentar a imagem em diversos objetos. **U-Net** é um dos principais representantes.
- Object detection: encontrar objetos nas imagens e marcá-los. **YOLO**



- Face recognition: uso de **triplet loss function**.

# Redes neurais em textos

## Textos como dados

Eu gosto de gatos

01

02

03

04

Eu gosto de cachorros

01

02

03

05

# Cada texto, um vetor

Eu gosto de gatos

01 02 03 04

00 00 01 02 03 04

Eu gosto de cachorros

01 02 03 05

00 00 01 02 03 05

Eu gosto de gatos e cachorros

01 02 03 04 06 05

01 02 03 04 06 05

exemplo 09: examples/09-text-vectorization.R

# Embedding's (ou Encoding)

- Até agora, cada palavra virou um número inteiro.
- O problema disso é que esses números inteiros representam categorias e não têm as principais de serem números.
- Não faz sentido fazer operações matemáticas como multiplicação e adição com esses números.
- Não existe uma noção de proximidade entre as palavras.

# One-hot encoding

- Cada texto é representado por uma matriz

Eu gosto de gatos

01    02    03    04

pad	Eu	Gosto	De	Gatos	Cachorros	E
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0

Eu gosto de gatos e cachorros

01    02    03    04    06    05

pad	Eu	Gosto	De	Gatos	Cachorros	E
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	1
0	0	0	0	0	1	0

# One-hot encoding

- O problema de fazer 'one-hot encoding' é principalmente computacional.
- Imagine um vocabulário de 20k palavras, faz com que a matriz tenha 20k colunas, a maioria dos valores sendo 0, sem trazer muita informação.
- A distância entre todas as palavras é igual.

# Embedding

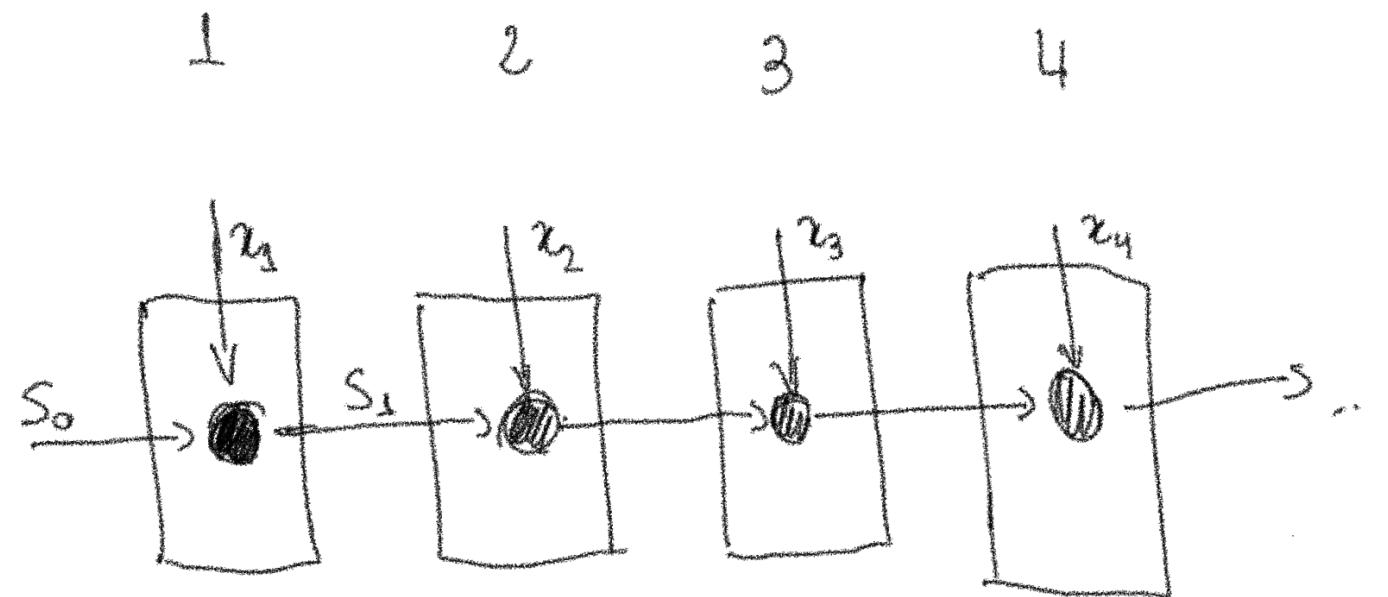
- Assume-se cada palavra endo representada por um vetor de pesos de dimensão  $d$ .

exemplo 10: examples/10-embedding.R

exemplo 11: examples/11-avg-pooling.R

exercício 10: exercícios/10-sarcasm.R

# Simple Recurrent Neural Networks



# Simple Recurrent Neural Network

