Capítulo 3 : Neurônios

Em março de 2016, o software AlphaGo venceu um mestre de Go. O feito é impressionante por se tratar de um jogo difícil de computar.
Inventado há mais de 2,500 anos, motivou avanços em matemática. Existem $2,08 * 10^{170}$ maneiras válidas de dispor as peças no tabuleiro. O polímata chinês Shen Kuo (1031–1095) chegou a um resultado próximo 10^{172} séculos atrás. Vale lembrar que o número de átomos no universo observável é de módicos 10^{80}.

No capítulo anterior, aprendemos uma formulação básica de modelo preditivo, a regressão linear simples. A seguir, estenderemos nosso leque de ferramentas para novas classes de relações, também incluindo mais informações na entrada de nossos modelos.

Mais do que isso, conheceremos a primeira máquina inteligente da história.

O perceptron de Rosenblatt

Frank Rosenblatt (1928 - 1971) nasceu e morreu em 11 de julho, mas esse não é o fato mais curioso da biografia deste psicólogo. Foi o responsável pelo desenvolvimento do primeiro neurônio artificial. Em suas palavras, o primeiro objeto não biológico a recriar uma organização do ambiente externo com significado.

It can tell the difference between a cat and a dog, although it wouldn't be able to tell whether the dog was to the left or right of the cat. Right now it is of no practical use, Dr. Rosenblatt conceded, but he said that one day it might be useful to send one into outer space to take in impressions for us. - New Yorker, December, 1958[1]

O aparato reproduzia o entendimento da época sobre o funcionamento de um neurônio. O corpo recebe sinais de dendritos e, após processamentos ocultos, produz um output na forma de sinal elétrico pelo axônio. A primeira matematização viria do modelo de McCulloch & Pitts ("A Logical Calculus of the Ideas Immanent in Nervous Activity", 1943).

Em 1949, Donald Hebb descreveu em seu clássico *The Organization of Behavior* um mecanismo plausível para a aprendizagem. Comumente expressa na máxima "Cells that fire together wire together" (células que disparam juntas, conectam-se entre si).

Com o objetivo de criar uma máquina que pudesse processar inputs diretamente do ambiente físco (e.g. luz e som), Rosenblatt concebeu extensão elegante do modelo em 1957 ("The Perceptron[*do latim, percipio, compreender*"]–a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory"). Composto de três partes: o sistema S (sensório); o sistema A (associação) e o sistema R (resposta).

[1] Ele consegue diferenciar um gato de um cachorro, ainda que não seja capaz de dizer se o cachorro estava à esquerda ou à direita do gato. No momento, não tem uso prático, Dr. Rosenblatt admitiu, porém disse que um dia pode ser útil para enviar um [aparato] ao espaço para capturar impressões para nós.

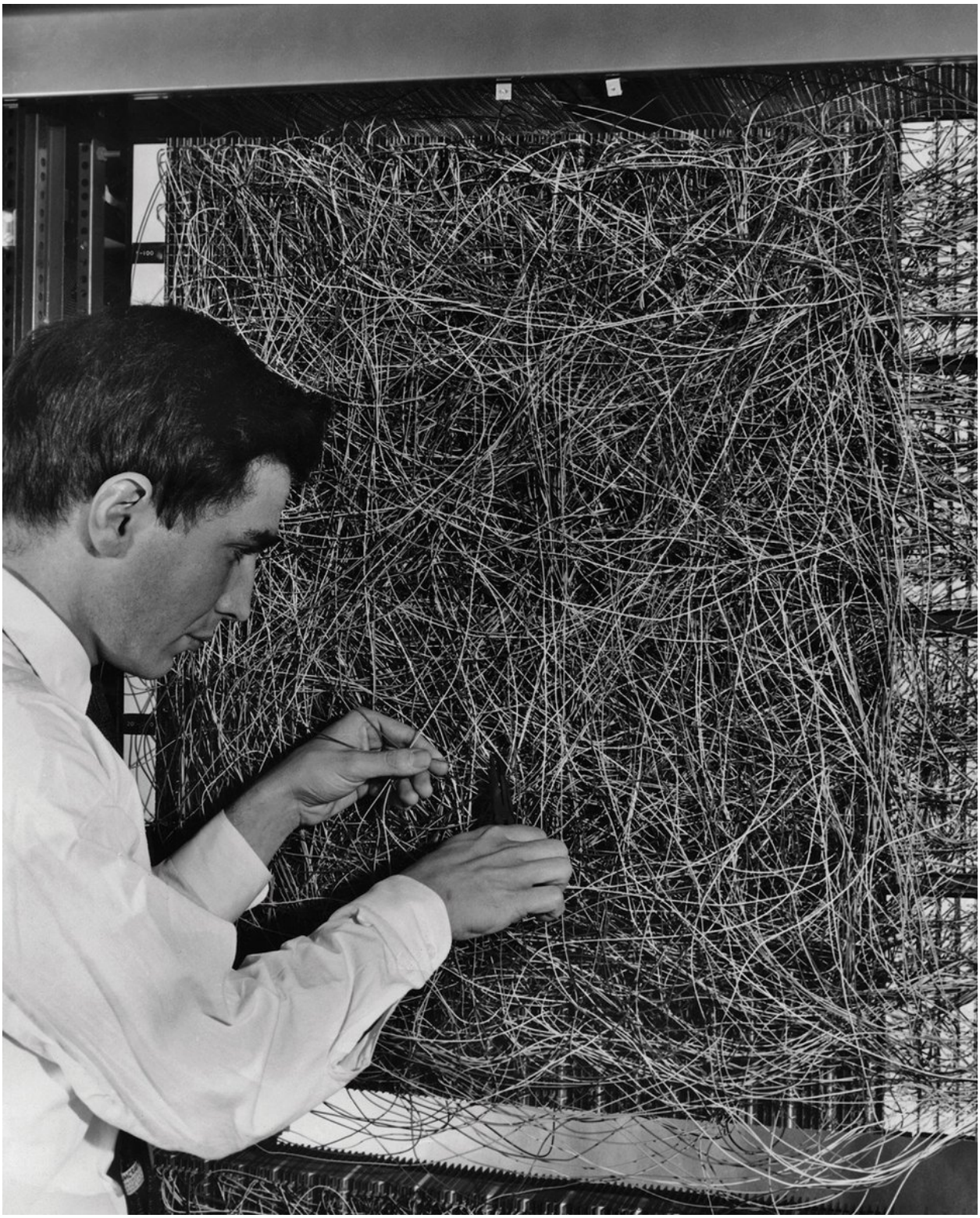


Figure 1: Frank Rosenblatt e Mark I.

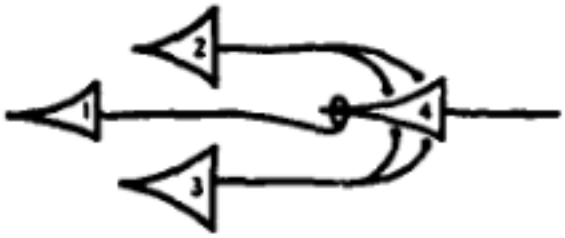


Figure 2: Diagrama de células lógicas em McCulloch & Pitts

O neurônio "lógico" cru de McChulloch & Pitts foi modificado de maneira a processar inputs através de pesos antes da saída. A aprendizagem se dá pela modificação desses pesos.

Inicialmente, o perceptron foi simulado em um IBM 704 (também berço das linguagens FORTRAN e LISP). Em seguida, implementado como um dispositivo físico, batizado de Mark I Perceptron.[2] Um estudo mais profundo foi publicado por ele em 1962 (Principles of neurodynamics)

[2]Mark I é um título comumente utilizado para a primeira versão de uma máquina.

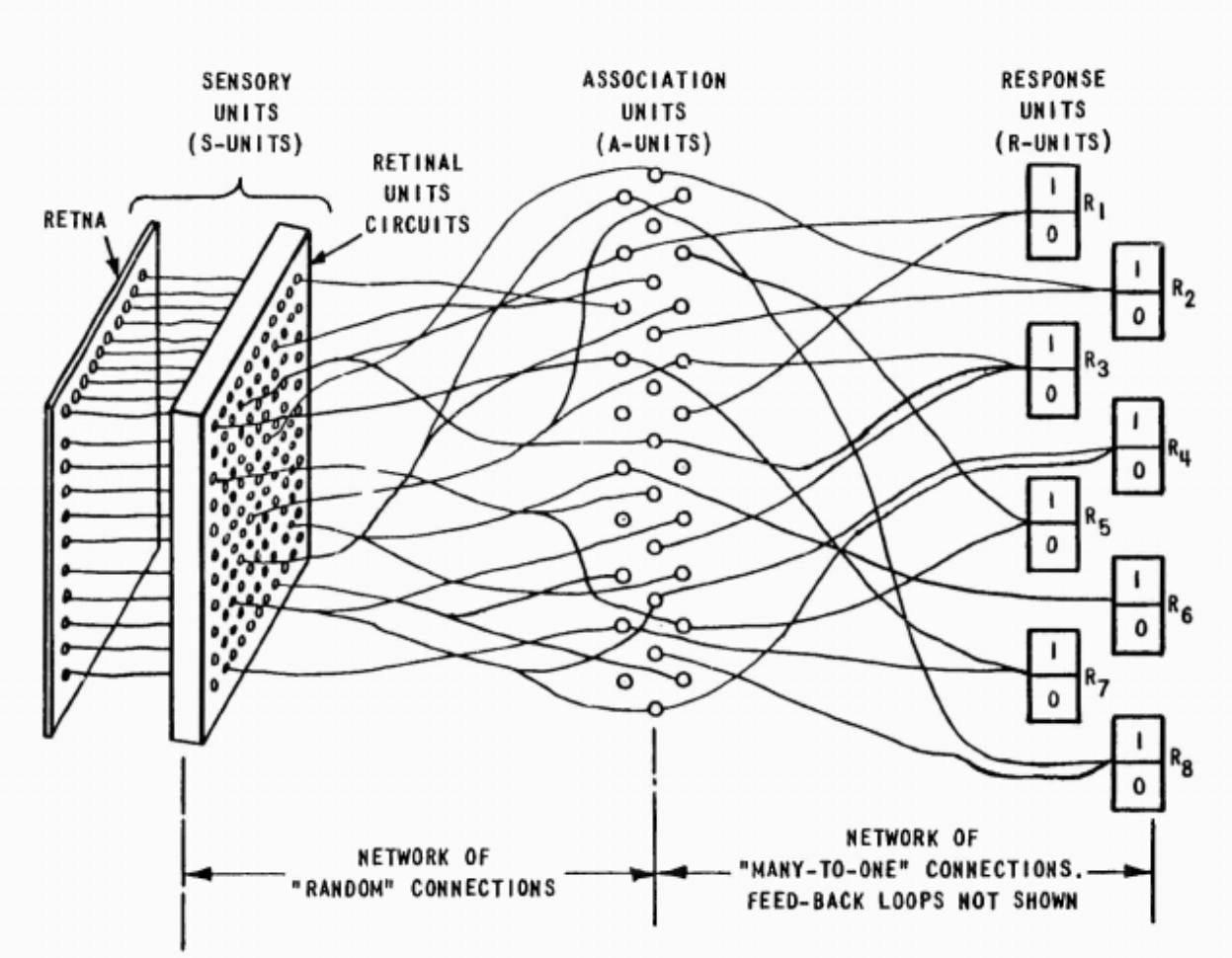


Figure 3: Organização do Mark I, retirado de seu manual de uso original

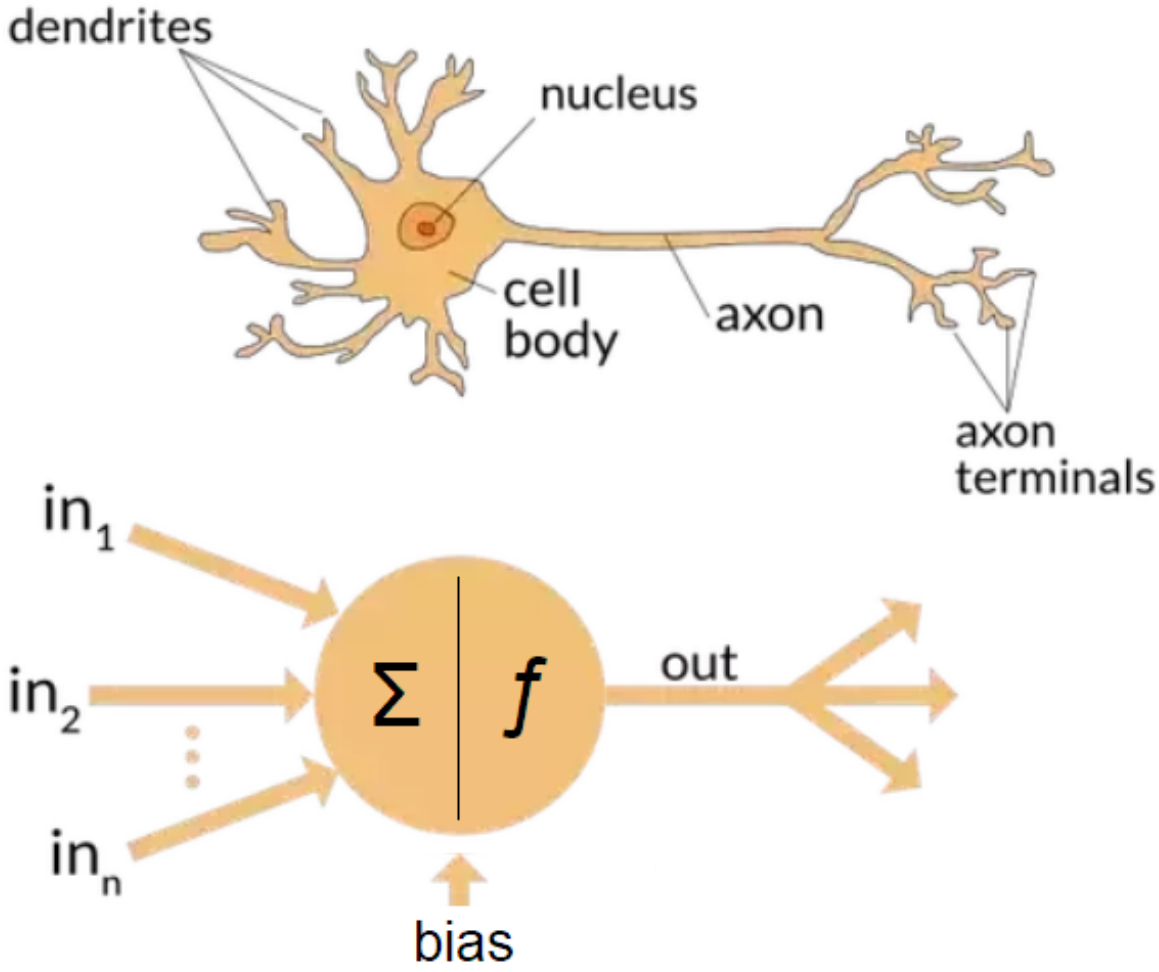


Figure 4: Neurônio biológico e seu correspondente artificial usando estrutura e notações atuais

Rosenblatt protagonizava calorosos debates sobre inteligência artificial na comunidade científica junto a Marvin Minsky, um amigo da adolescência. Em 1969, Minsky e um matemático (Seymour Papert) publicaram um livro centrado no Perceptron. (Perceptrons: An Introduction to Computational Geometry). Nele, provaram que o neurônio artificial era incapaz de resolver problemas não-lineares do tipo XOR. Para um problema eXclusive OR (OU eXclusivo) o neurônio deve disparar diante do estímulo A ou do estímulo B, porém não diante de ambos.

O impacto foi devastador sobre o otimismo vigente e se passou um período de 10 anos de baixíssima produção, conhecido como idade das trevas do conexionismo. A retomada dos neurônios artificiais aconteceu somente na década de 80. Infelizmente, Rosenblatt morreu prematuramente em 1972 num acidente de barco, não presenciando o renascimento dos perceptrons.

Sabendo das origens do modelo, é curioso que a maioria dos cursos introduzam perceptrons do ponto de vista puramente matemático, apontando a semelhança com neurônios como mera curiosidade. Pelo contrário, a inspiração em neurônios biológicos e posterior sucesso nas tarefas designadas fala em favor de um fantástico caso de sucesso para engenharia reversa.
A Natureza, através de evolução por seleção natural, é a verdadeira mãe desse algoritmo.

Criando neurônios

Mark I foi criado para reconhecimento visual, podendo ser considerado avô da visão computacional.
Possuía um campo de entrada fotossensível de 20x20 (400) células de Sulfeto de Cádmio, as unidades S. Ao reagir com a luz, CdS emite um elétron:

$$CdS + \gamma \rightarrow e^{-} + CdS^{+}$$

Caso a célula seja ativada, envia o sinal eletrônico a uma unidade intermediária A. A unidade intermediária, por sua vez, transmite um sinal eletrônico à saída. A intensidade do sinal é regulada por sucessos prévios. Esse processo ficará mais claro com a implementação a seguir.

Imaginemos que a imagem acima tenha 10 pixels de altura e 10 de largura.
Para simplificação, 10 x 10 pixels em preto e branco (100 pixels com valores entre 0,preto, e 255, branco). Esses pixels podem ser esticados e vistos como uma matriz x de dimensão $[100x1]$ com valores entre 0 e 255 em cada elemento.
Vamos simular uma imagem deste tamanho gerando uma matriz de dimensão 10x10 com 100 valores naturais aleatórios (entre 0 e 255) no R:

```
>set.seed(2600)
>my.image.data <- sample(0:255,100,replace=T)
>x <- matrix(my.image.data,10,10)
```

	1	2	3	4	5	6	7	8	9	10
1	66	43	199	168	79	33	108	17	51	161
2	11	187	252	8	190	247	69	112	165	129
3	146	97	26	203	112	143	20	134	240	56
4	60	128	185	119	222	107	39	250	230	85
5	38	112	141	49	201	210	46	59	35	59
6	71	143	54	15	190	94	99	33	222	35
7	17	35	146	204	14	15	201	236	172	17
8	130	64	98	42	169	5	73	24	240	32
9	232	40	60	140	46	210	163	199	116	100
10	193	62	106	8	3	152	233	170	74	153

Eis a nossa imagem [10x10]. O computador lê os valores entre 0 (preto) e 255 (branco), dispondo para nós o sinal visual correspondente.

Em nosso exemplo hipotético, o classificador precisa saber se uma imagem apresentada corresponde à de um navio ou não.

Em regressão linear múltipla, calculamos um peso β para cada variável. O racional aqui é parecido: ponderamos cada pixel por seus respectivos pesos. Em analogia, cada imagem é uma observação de 100 variáveis.

O neurônio deve disparar (output $y = 1$) caso seja um navio ou permanecer em repouso ($y = -1$) caso não seja.

Matematicamente, é uma multiplicação da matriz de valores da imagem x_j, de dimensão $[100x1]$ por uma matriz $W_{[100X1]}$ que traz i pesos (weights) estimados para cada pixel para cada classe. Então, forçamos o resultado para +1 ou -1 com uma função de ativação (ϕ).

$$y = \phi(W^T X)$$

Usaremos a função *Heaviside step*:

$$\phi(x) = \begin{cases} +1 & se \quad x \geq 0 \\ -1 & se \quad x < 0 \end{cases}$$

Em R:

```
library(magrittr)
# Heaviside
>phi_heavi <- function(x){ifelse(x >=0,1,-1)}
# Iniciando pesos com base em distribuição normal
>my_weights <- rnorm(100)/100
>w <- matrix(my_weights,100,1)
# Multiplicacao usando o operador %*%
>as.vector(x) %*% w
# Score
         [,1]
[1,] 20.19958
# Funcao de ativacao usando %>% para encadeamento
>as.vector(x) %*% w %>% phi_heavi
```

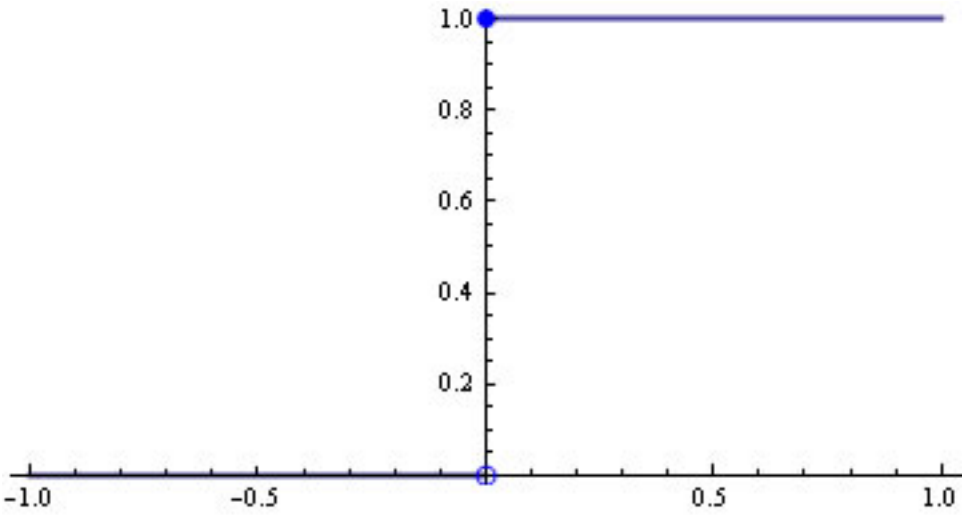


Figure 5: Heaviside step function

```
      [,1]
[1,]    1
```

Para o exemplo acima, nosso neurônio com pesos aleatórios foi ativado para o estímulo aleatório x. Inicialmente, estabelecemos pesos aleatórios a partir de uma distribuição normal.
Então, o objetivo é observar as respostas corretas em várias imagens x_i e alterar os valores de W para que os scores maiores sejam os das classes corretas.

O processo de treino é bastante simples:
Seja x_{i_j} o i-ésimo pixel da observação j. E w_0 o peso correspondente inicial, o peso atualizado, w' é:

$$w' = w_0 + \Delta w$$

Em que Δw indica o magnitude e o sentido da modicação no peso.
Aceitemos, por enquanto, a fórmula:

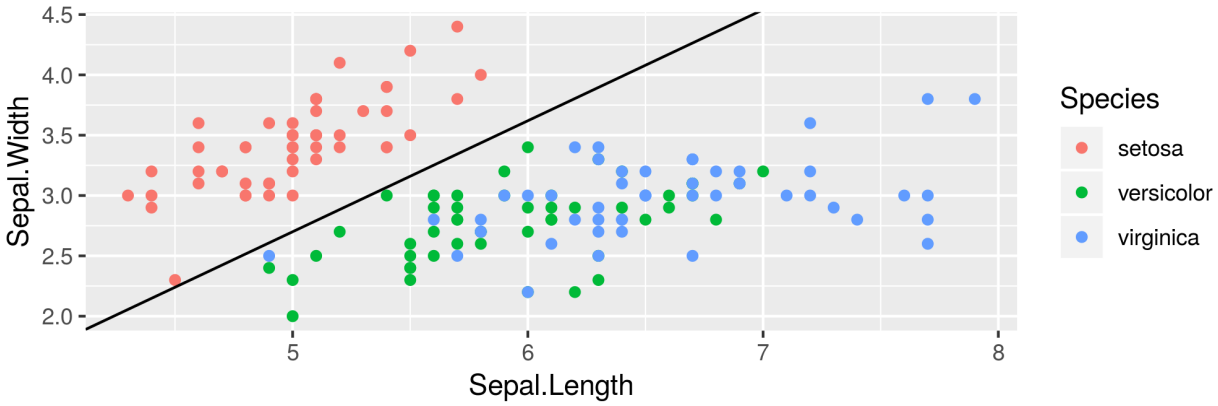
$$\Delta w_i = \eta(scorej - output_j)x_i$$

Em que x_{i_j} é o valor do i-ésimo pixel, w_i é o i-ésimo peso e η uma constante chamada *taxa de aprendizagem* (learning rate), que determina o tamanho dos incrementos feitos pelo algoritmo. Mostraremos a derivação dessa equação a seguir.

Se os dados são linearmente separáveis, o algoritmo converge com um número suficiente de exemplos.

Assim, funciona para separar flores *setosa* de outra classe, mas não teriamos bons resultados separando virginica de versicolor.

```
>ggplot(iris,aes(x=Sepal.Length,y=Sepal.Width,color=Species))+
geom_point()+ geom_abline(slope = 0.92,intercept = -1.9)
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
7
8
Sepal.Length
Species
setosa
versicolor
virginica

Auto MaRk I

Usando as abstrações acima, codificamos nosso perceptron em R, o Auto MaRk I.
Argumentos: Exemplos (x, vetor de números reais) e estados esperados (y, disparar = 1 vs. não disparar = -1) devem ter mesmo tamanho. **Eta:** Número especificando constante de aprendizagem.

Auto MaRK I inicializa um peso aleatório para cada entrada e, numa ordem aleatória, percorre os exeplos atualizando os pesos.

```
library(magrittr)
>mark_i <- function(x, y, eta) {
  # inicializa pesos randomicos de distribuicao normal
  w <- rnorm(dim(x)[2]) # numero de pesos = numero de colunas em x
  ypreds <- rep(0,dim(x)[1]) # inicializa predicoes em 0
    # Processa as observacoes em x de forma aleatoria
    for (i in sample(1:length(y),replace=F)) {
      # predicao
      ypred <- sum(w * as.numeric(x[i, ])) %>% phi_heavi
      # update em w
      delta_w <- eta * (y[i] - ypred) * as.numeric(x[i, ]) #nota: x[i,] sera multiplicado como matr
      w <- w + delta_w
      ypreds[i] <- ypred # salva predicao atual
    }
  print(paste("Weights: ",w))
  return(ypreds)
}
```

Vamos testá-lo para o problema proposto, separando flores *setosa* de *versicolor*. Preparação de dados:

```
>train_df <- iris[1:100, c(1, 2, 5)]
>train_df[, 4] <- -1
>train_df[train_df[, 3] == "setosa", 4] <- 1
>names(train_df) <- c("s.len", "s.wid", "species","target")
>head(train_df)
     s.len s.wid species target
1   5.1   3.5  setosa      1
2   4.9   3.0  setosa      1
3   4.7   3.2  setosa      1
4   4.6   3.1  setosa      1
5   5.0   3.6  setosa      1
6   5.4   3.9  setosa      1
> train_df[60:65,]
   s.len s.wid    species target
60   5.2   2.7 versicolor     -1
61   5.0   2.0 versicolor     -1
62   5.9   3.0 versicolor     -1
63   6.0   2.2 versicolor     -1
64   6.1   2.9 versicolor     -1
65   5.6   2.9 versicolor     -1
>x_features <- train_df[, c(1, 2)]
>y_target <- train_df[, 4]
```

E então, podemos ativá-lo:

```
>y_preds <- mark_i(x_features, y_target, 0.002)
[1] "Weights:  0.309434266643425"
[2] "Weights:  -0.426791898133975"
```

```
> table(y_preds,train_df$target)
y_preds -1  1
     -1 19  1
      1  31 49
> y_preds
  [1]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 [25]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 -1  1  1  1  1  1  1
 [49]  1  1 -1  1  1 -1  1 -1 -1  1  1  1 -1  1 -1  1  1  1  1  1  1  1 -1 -1
 [73]  1 -1 -1  1 -1 -1  1  1 -1  1 -1  1  1  1  1  1  1 -1  1  1 -1  1 -1  1
 [97] -1  1  1 -1
```

```
> y_preds <- mark_i(x_features, y_target, 0.05)
[1] "Weights:  -1.12748454064396"
[2] "Weights:  1.35197455996465"
> table(y_preds,train_df$target)
y_preds -1  1
-1 30 21
1  20 29

> y_preds <- mark_i(x_features, y_target, 0.1)
[1] "Weights:  -2.08944843785222"
[2] "Weights:  3.35800090343738"
> table(y_preds,train_df$target)
y_preds -1  1
-1 36 14
1  14 36

> y_preds <- mark_i(x_features, y_target, 0.01)
[1] "Weights:  -0.250410476080629"
[2] "Weights:  0.447470183281492"
> table(y_preds,train_df$target)
y_preds -1  1
-1 43 16
1   7 34
```

Chamamos η de hiperparâmetro. A escolha de valores para hiperparâmetros é um dos desafios em aprendizagem estatística. Uma maneira trivial é testar muitos valores possíveis e observar o desempenho, entretanto isso não exequível para grandes volumes de dados e/ou muitos parâmetros. Existem diversos processos heurísticos e algoritmos para encontrar valores ótimos.

Uma forma popular para otimizar o treinamento é particionar o dataset em pedaços e apresentar os particionamentos (epochs) repetidas vezes ao classificador ou acumular os erros de epochs ao invés de exemplos individuais.

Gradient Descent

No exemplo anterior, atualizamos os pesos com uma fórmula contendo taxa de aprendizagem (η) e outros parâmetros: a função de erro entre score desejado($score$) e output $E = d(score_j, output_j)$; valor da entrada (x_i).

$$w_i' = w_i + \Delta w_i$$

Δw_i pode ser obtido usando o conceito de Gradient Descent. Levando em conta cada j-ésima observação, definimos uma função de perda L expressando a soma dos erros nos n exemplos e minimizamos ela.

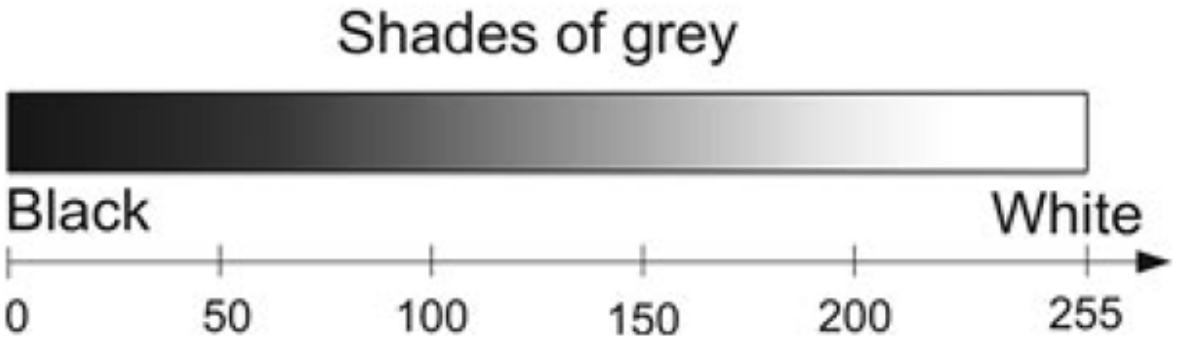
$$min(L) = min \sum_{j}^{n} E(score_j, output_j))$$

. Calculamos o valor dos pesos atuais e percorremos o espaço em direção a um valor mínimo local. Se a superfície L for convexa, acharemos uma solução ótima.

Usaremos para nossa função de erro a distância euclidiana entre score desejado e output. O score desejado é a resposta ótima o output é um produto entre pesos e entrada[ˆ22]:

$$E = d_{eucl.}(score_j, output_j) = (score_j - output_j)^2$$

Notem que o processo envolve implementar uma função de erro entre resultados da rede e um espaço virtual de scores ótimos. O sucesso do treinamento depende de uma correspondência entre a função de distância escolhida e a distância real no espaço em que os dados foram gerados. Não sabemos se isso reflete a realidade. No exemplo, cada pixel reflete um sinal de 0 a 255.
A figura abaixo mostra a correspondência entre valores da medida e escala visual.



A intuição para sensibilidade à luz pode ser percebida num intervalo contínuo entre incidência total de luz (valores extremos de branco, medida: 255) e ausência total (valores extremos de preto, medida: 0). Supondo que podemos atribuir um rótulo a cada tom de cinza e que esse conjunto é ordenável pela *clareza*, dizemos que há isomorfismo de ordem entre os conjuntos.

Isso implica que a distância eulidiana deve funcionar em nossas medidas como nos números reais $\mathbf{R}$. Resta saber se a projeção das observações é linearmente separável. É intuitivo para seres humanos saber quais problemas serão separáveis: basta imaginar a tarefa de diferenciar tipos de imagens com uma regua numa tela em preto e branco.

Para descobrir o valor mínimo de L, vamos encontrar polos através de derivadas. Ou, seu equivalente para funções de múltiplas variáveis (espaços multidimensionais), o gradiente(∇). É o produto escalar das derivadas parciais daquela função.

Para cada observação x_j, a derivada parcial da função de perda em relação a um peso w_i expressa a taxa de variação no erro global em função daquele peso. $\frac{d}{dw_i}L(w_i) = \frac{d}{dw_i}\frac{1}{n}\sum_j nE(score_j, output_j)$

Sabemos então se devemos ajustar o peso para cima ou para baixo, assim com a magnitude do passo. Algebricamente, modificaremos w seguindo o inverso do gradiente. A taxa de aprendizagem é um hiperparâmetro que regula artificialmente o tamanho desse passo:

$$\Delta w_i = -\eta\frac{dL}{dw_i}$$

$$= -\eta\frac{d}{dw_i}\frac{1}{n}\sum_j^n E(score_j, output_j)$$

Lembrando que o erro é dado pela distância euclidiana:

$$= -\eta\frac{d}{dw_i}\frac{1}{n}\sum_j^n (score_j - output_j)^2$$

.

Fazemos $f(x) = (score_j - output_j)$ e $g(x) = x^2$, de maneira que

$$L = \frac{1}{n}\sum_j^n E(score_j, output_j) = (g \circ f)$$

$$= \frac{1}{n}\sum_j^n (score_j - output_j)^2$$

Podemos resolver $\frac{d}{dw_i}L$ aplicando a regra de cadeia $(g \circ f)' = (g' \circ f)f'$ e a 'regra do tombo' para derivadas de polinômios ($\frac{d}{dx}(x^n) = nx^{n-1}$).

Então,

$$f' = \frac{d}{dw_i}(score_j - output_j)$$

O output é dado pelo produto escalar entre pesos w_j e entradas x_j:

$$f' = \frac{d}{dw_i}(score_j - w_j \cdot x_j)$$

O score desejado não depende dos pesos, portanto a primeira derivativa é 0.

$$f' = 0 - \frac{d}{dw_i}w_j \cdot x_j$$

$$= -\frac{d}{dw_i}\sum_{i,j}^n w_{i,j} * x_{i,j}$$

$$== -\frac{d}{dw_i}(w_0 * x_0 + ... + w_i * x_i + w_n * x_n)$$

Os termos não dependentes de w_i também são zerados e:

$$f' = -\frac{d}{dw_i}w_i x_i$$

A função a ser derivada agora descreve uma relação linear (polinômio de grau 1) em w_i e temos:

$$f' = (-x_{i,j})$$

Sabendo f', buscamos o outro termo em $(g \circ f)'$:

$$(g' \circ f) = 2(score_j - output_j)^{2-1}$$

$$= 2(score_j - output_j)$$

Por fim, a derivada parcial da função de perda para o i-ésimo peso w_i:

$$\frac{dL}{dw_i} = \sum_{j}^{n} \frac{d}{dw_i}(score_j - output_j)^2$$

.

$$= \sum_{i,j}^{n} 2(score_j - w_j \cdot x_j))(-x_{i,j})$$

Escalamos L' por $-\frac{1}{2} * \eta$:

$$-\frac{1}{2} * \eta \frac{dL}{dw_i} = \frac{1}{2} * \eta 2(score_j - output_j)x_j$$

$$\Delta w_i = \eta \sum_{j} n(score_j - w \cdot x)(x_j)$$

Como implementamos antes no Auto MaRK I.

```
(...)
ypred <- sum(w * as.numeric(x[i, ])) %>% phi_heavi
delta_w <- eta * (y[i] - ypred) * as.numeric(x[i, ])
w <- w + delta_w
(...)
```

Início de textos em construção.

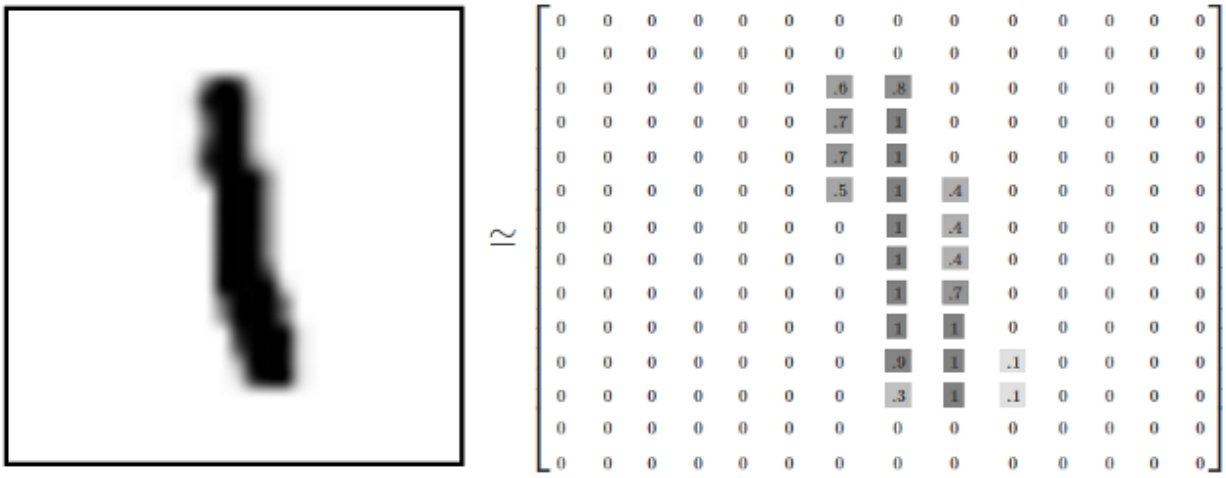


Figure 6: Exemplo de “1” em letra cursiva e sua representação numa matriz 2x2. http://colah.github.io/posts/2014-10-Visualizing-MNIST/

Deep learning

Intuições

Com o aprendizado através de exemplos, otimizamos otimizamos nosso classificador (mudando pesos W) para minimizar a perda usando aproximações. A função de perda é menor quando temos pontuações (votos) maiores para as classes certas. SVMs têm bom desempenho em diversas estruturas de dados, especialmente quando a arquitetura é otimizada por um usuário experiente. Onde entram as redes neurais?

Going Deep

As versões reais da maioria dos conceitos criados por seres humanos não são idênticas umas às outras. Em outras palavras, não existe um conjunto rígido de regras para classificarmos a maior parte das entidades ao nosso redor. Muitas entidades são diferentes, porém similares o suficiente para pertencer a uma mesma categoria.

Todos são naturalmente reconhecidos como felinos, mas apresentam variações de tamanho, cor e proporção em todo o corpo.

Esse é um problema interessante e antigo. Alguns filósofos acreditam que abstrações humanas são instâncias de um conceito mais genérico: mapas biológicos contidos em redes neuronais (Paul Churchland, Plato's Camera).
Esses mapas estão associados de forma hierarquizada. Numerosos padrões em níveis inferiores e um número menor em camadas superiores.
No caso da visão, neurônios superficiais captam pontos luminosos. O padrão de ativação sensorial enviado ao córtex visual primário é o primeiro mapa, que é torcido e filtrado caminho cima.

Neurônios intermediários possuem configurações que identificam características simples: olhos e subcomponentes da face. Por fim, temos camadas mais profundas, ligadas a abstrações.

Deduzindo superfícies

Um classificador deve capturar essa estrutura abstrata a partir de modelos matemáticos tratáveis. Para examinarmos esse aspecto, usemos um exemplo. O gráfico abaixo representa milhares de amostras com: (1) a curva diária natural de um hormônio (em vermelho) e a curva sob uso de esteroides anabolizantes (azul).

Como hipotéticos membros de uma comitê atlético, nosso objetivo aqui é, dada uma amostra, saber se o atleta está sob efeito de esteroides. Quando experimentamos, normalmente haverá ruídos (erros) na medida e receberemos medições imprecisas da curva. Variações na dieta daquele dia, micções, sudorese, stress e outros fatores.

Usamos o tempo (t, eixo horizontal) e nível hormonal (β, eixo vertical).

Numa regressão logística simples, fazemos essa classificação com base nas probabilidades de uma função sigmoide. Temos uma probabilidade (valor entre 0 e 1). $P(h, \beta) = 1/(1 + exp - (i + t * h + \beta * y + \epsilon))$. ϵ representa o erro e i é uma constante.

Em uma linha de R:

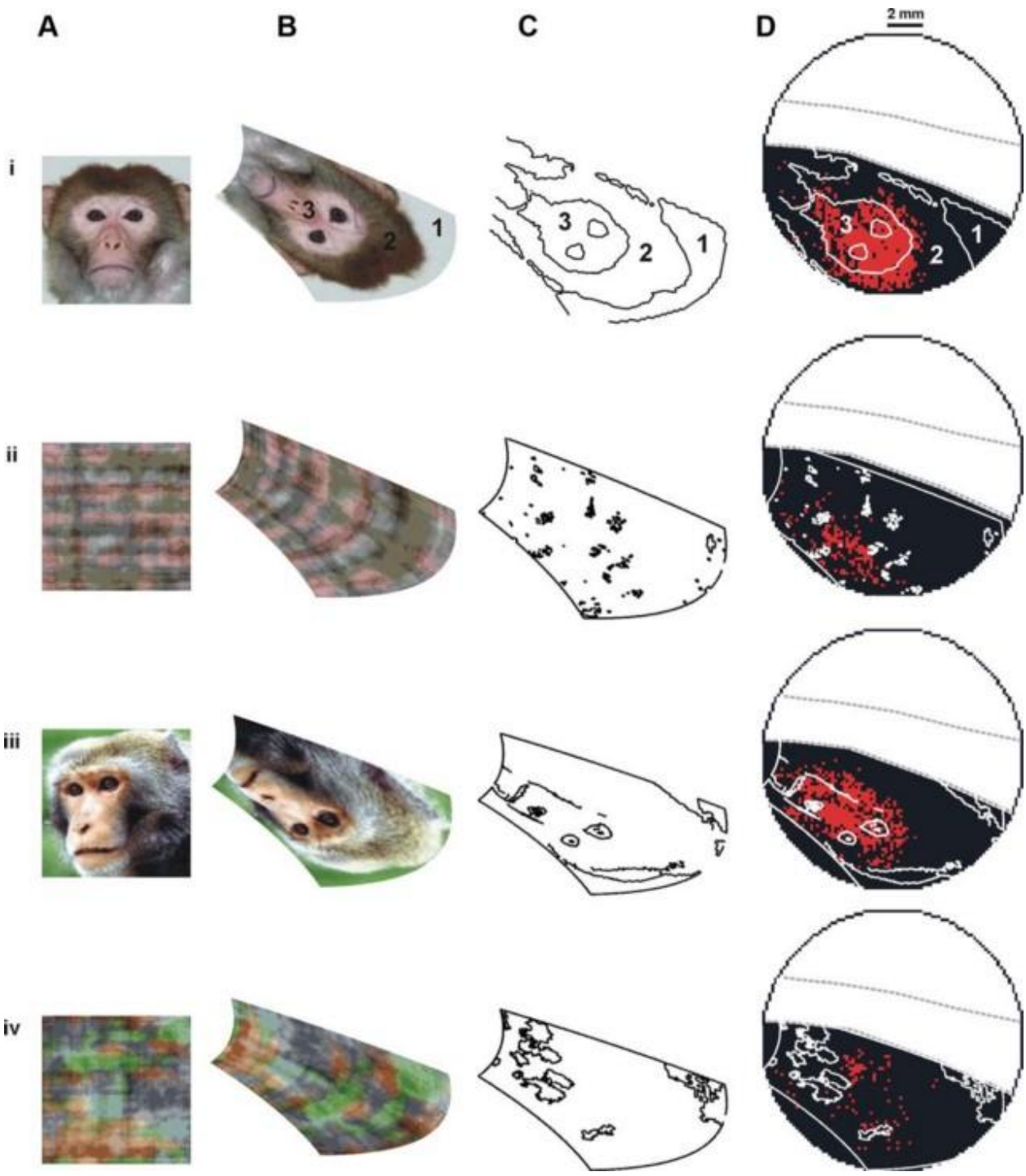
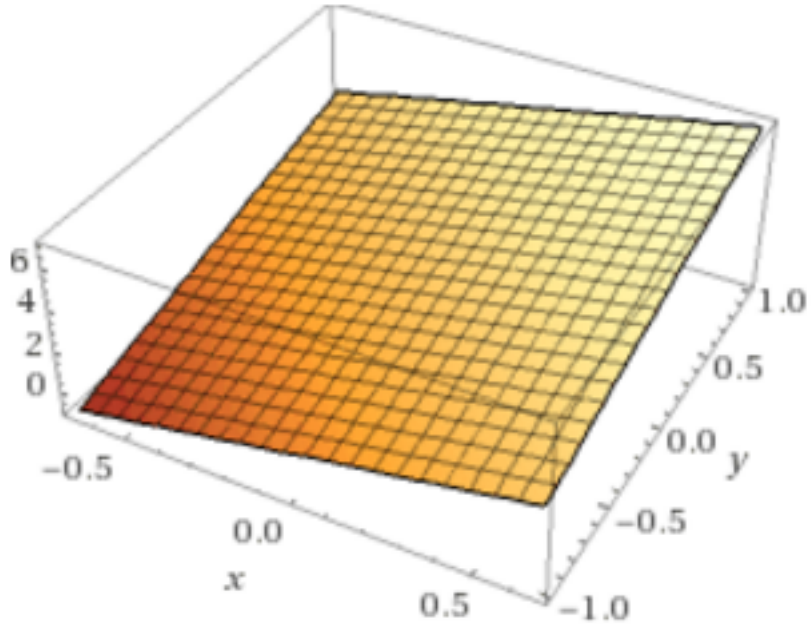


Figure 7: Resposta a estímulos visuais em V1 de Macaca fascicularis http://www.jneurosci.org/content/32/40/13971

Figure 8: Retirado de: https://www.youtube.com/watch?v=SeyIg6ArS4Y

$$logist.fit < -glm(type_dic\ beta + tempo, family = binomial, data = inv.ds)$$

A vantagem de usar essa modelagem é que temos uma relação direta entre o inverso dessa função (P^(-1), "logito") e a combinação linear dos nossos parâmetros: $logit(P(x)) = i + t * x + \beta * y + \epsilon$ Em outras palavras, o processo de estimação é parecido com o da regressão linear, que é facilmente tratável. Outra consequência é que assumimos que a distinção entre classes (com base no logito, log odds) pode ser dada por um limite. Este tem uma relação linear com nossas variáveis. Estimamos a magnitude e o sentido dessas relações pelos parâmetros da regressão.



Podemos imaginar que o log odds (z, eixo vertical) cresce linearmente com uma combinação de duas variaveis (x e y). Notem que a superfície definida pelo nossa equação/modelo é um plano. z = 3 + 3x + 2y. Plotado no Wolfram Alpha Estimamos qual seria a posição na reta dada por aquela medida e usamos um limite de decisão (decision boundary) linear. Voltando ao nosso exemplo, seria difícil capturar as diferenças usando apenas esta estratégia.

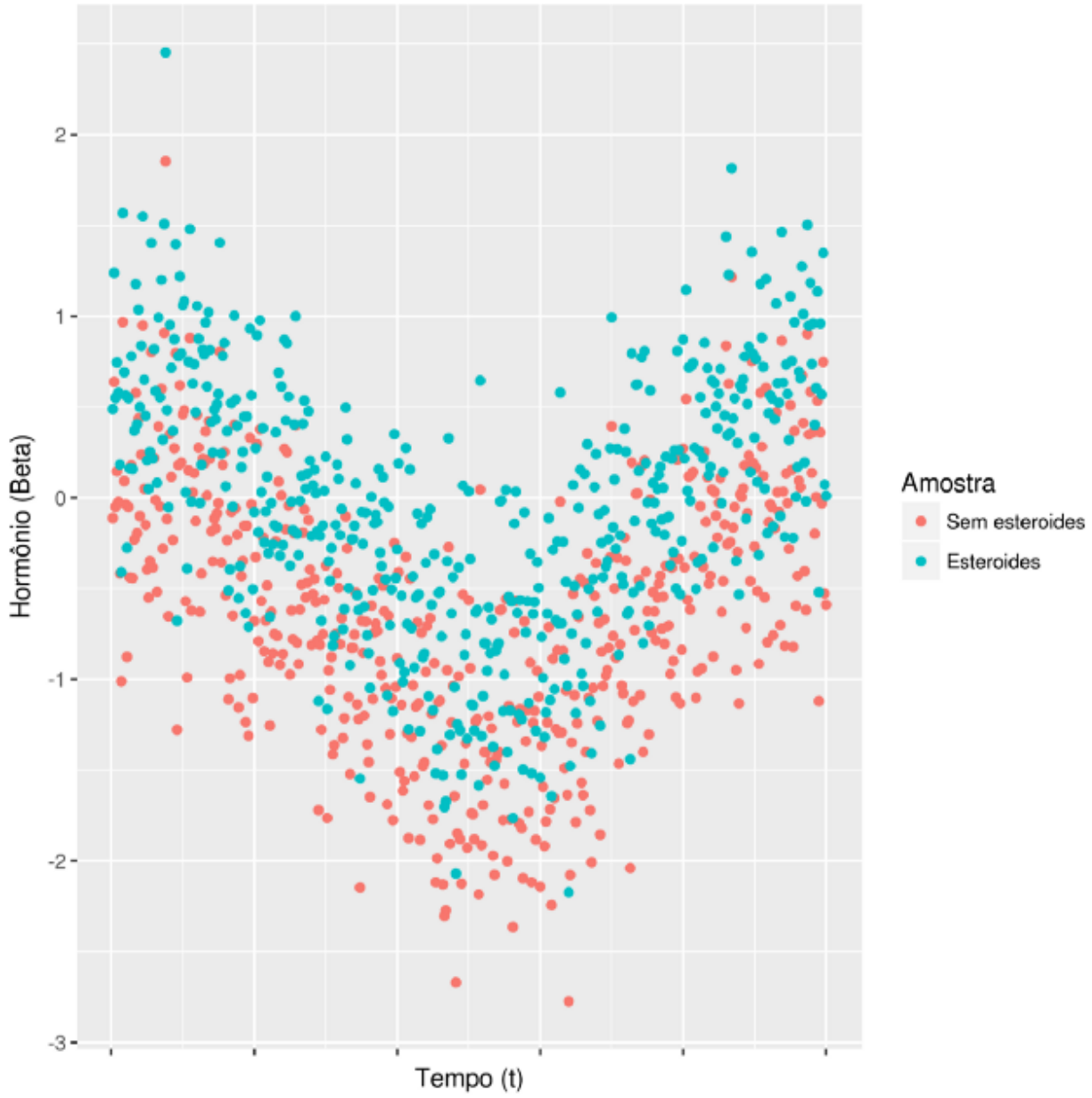
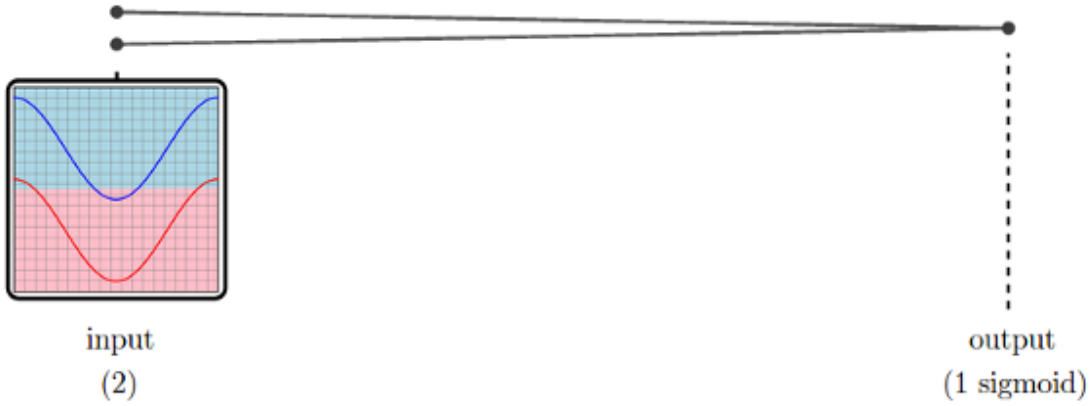


Figure 9: Exemplo inspirado no texto de Chris Olah (http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/)



Acima, um neurônio sigmoide, que equivale à regressão logística. É como o plano anterior, mas visto de cima, dividimos ele em duas regiões para classficação. http://colah.github.io/posts/2015-01-Visualizing-Representations/ Por que? O classificador linear otimiza suas respostas levando em conta apenas o valor absoluto da medida hormonal. Isto é, valores acima de um limite serão considerados dopping, não considerando horário. Matematicamente, o coeficiente para o tempo foi ajustado em 0. Mudar isso tornaria a reta divisória inclinada em relação ao eixo x, piorando a classificação.

Podemos verificar isso diretamente através dos parâmetros estimados em nosso modelo de regressão.

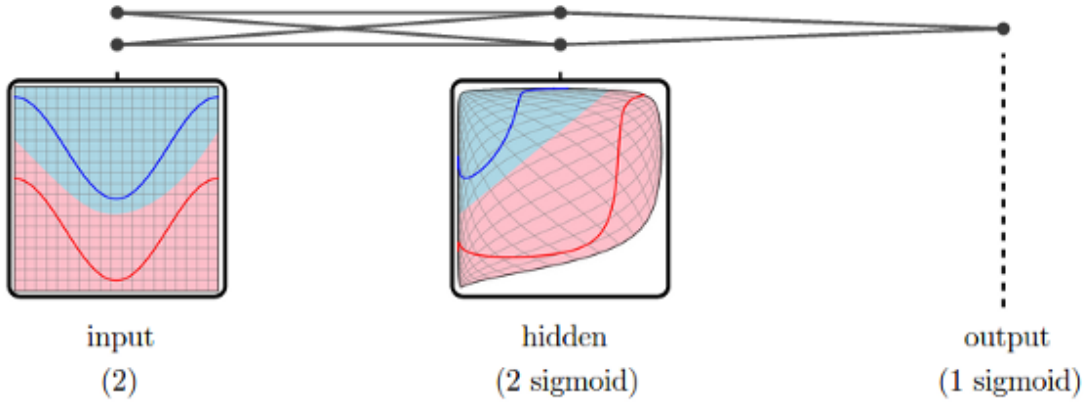
```
> summary(logist.fit)
Call:  glm(formula = type_dic ~ beta + tempo, family = binomial, data = inv.ds)
Coefficients:
 (Intercept)     beta        tempo
-0.8752803   -3.6195723   -0.0001221 # Próximo a zero
Degrees of Freedom: 999 Total (i.e. Null);  997 Residual
Null Deviance:     1386
Residual Deviance: 774.4  AIC: 780.4
> prob=predict(logist.fit,type=c("response"))
> inv.ds$prob=prob
> curve <- roc(type_dic ~ prob, data = inv.ds)
> curve

Call:
 roc.formula(formula = type_dic ~ prob, data = inv.ds)
Data: prob in 500 controls (type_dic 0) < 500 cases (type_dic 1).
Area under the curve: 0.8767
```

Quem poderá nos ajudar?

A solução é introduzir termos polinomiais de grau mais alto ($x^2, x^3 \ldots$), interações ou usar funções mais complexas. Aí corremos o risco de realizar sobre ajuste. Deixar o sinal dos confundir e fazer um modelo complexo que não funciona em novos exemplos. E o que acontece se conectarmos classificadores simples hierarquicamente?

A resposta de uma unidade é usada como a entrada de outras. Quando processamos o sinal em etapas, cada camada modifica os dados para as camadas posteriores, transformando e filtrando/dando forma.

As camadas intermediárias permitem a transformação gradual do sinal, e o sistema acerta usando apenas dois classificadores simples (sigmoides). No exemplo acima, temos uma camada de 2 neurônios entre o input e o out-

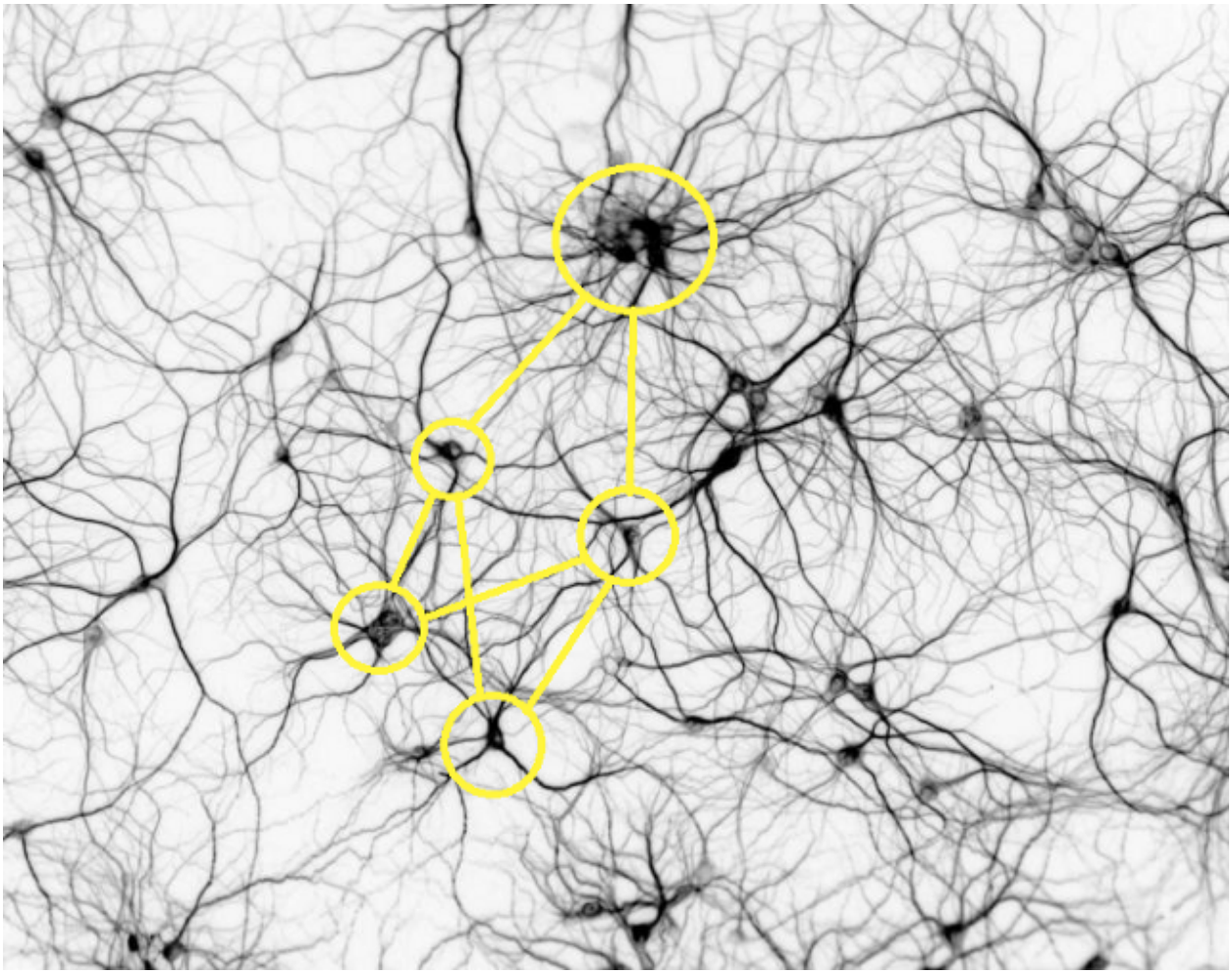


put.

Agora, a primeira camada (hidden) modifica a entrada com duas unidades sigmoides e a segunda camada pode classificar corretamente usando apenas uma reta, algo que era impossível antes. Em tese, esse modelo pode capturar melhor as características que geraram os dados (flutuação hormonal ao longo do dia).

Neurônios

Notem que o diagrama acima lembra uma rede neural. Esse tipo de classificador foi inspirado na organização microscópica de neurônios reais e acredita-se que seu funcionamento seja de alguma forma análogo. A arquitetura de redes convolucionais (convolutional neural networks), estado da arte em reconhecimento de imagens, foi inspirada no córtex visual de mamíferos (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1557912/). Outros modelos bio inspirados (Spiking neural networks, LTSMs. . .) apresentam desempenhos inéditos para tarefas complexas e pouco estruturadas, como reconhecimento de voz e tradução de textos. A teoria mais aceita é de que o maquinário neural dos animais foi desenhado por processos evolutivos, como a seleção natural. Assim, apresenta coloridas formas de complexidade a depender da tarefa desempenhada.



Como podemos ver, as redes biológicas são complexas, com até dezenas de bilhões de unidades de processamento paralelas conectadas. Zona destacada possui grafo isomorfo ao descrito no texto. Modificado de http://www.rzagabe.com/2014/11/03/an-introduction-to-artificial-neural-networks.html

Nos modelos profundos (deep) de reconhecimento de rosto, neurônios de camadas superficiais capturam bordas, ângulos e vértices, camadas intermediárias detectam presença de olhos, boca, nariz. Por fim, camadas ao final da arquitetura decidem se é um rosto ou não e a quem ele pertence.

Eficiência e aplicações

Podemos demonstrar formalmente que uma rede neural com apenas uma camada interna é capaz de aproximar qualquer função. A prova não é lá essas coisas, já que, no fundo o que fazemos é criar uma tabela de consulta (lookup table) para os valores de entrada e saída usando os neurônios. Na prática, é difícil obter boas performances. Tão difícil que redes neurais passaram décadas esquecidas. Se você rodar o modelo abaixo, baseado no nosso exemplo, verá que a acurácia é próxima da regressão logística. É necessário algum conhecimento e tempo para afinar os detalhes. Normalmente, depende da qualidade e da quantidade dos dados.

```
# Neural Net para o exemplo
library(deepnet)
inv.ds$tempo.norm <- normalize(inv.ds$tempo)
deep.log.dbn <- dbn.dnn.train(
 x=as.matrix(inv.ds[,c("beta","tempo.norm")]),
 y=as.numeric(as.character(inv.ds$type_dic)),
```

```
hidden = c(2), activationfun = "sigm",
learningrate=2.65, momentum=0.85, learningrate_scale=1,
output = "sigm", numepochs=3, batchsize= 11)
```

As redes neurais passaram algum tempo esquecidas, até que algumas reviravoltas (http://people.idsia.ch/~juergen/who-invented-backpropagation.html) permitiram o treinamento eficaz delas redes. Algoritmos para melhorar o treinamento, assim como arquiteturas econômicas ou especialmente boas em determinadas tarefas. Além disso, o uso de processadores gráficos (GPU), desenhados para as operações de álgebra linear que discutimos (com matrizes) permitiu treinar em um volume maior de dados.

Backpropagation

Uma vez que o texto é sobre deep leaning, precisamos falar de backpropagation . Como vimos nas partes 1 e 2, o treinamento consiste em ajustar os pesos W do classificador (SVM) para minimizar a função que calcula nosso erro E. Como alguém de olhos vendados em uma ladeira, podemos dar um passo e saber medir qual o efeito sobre a nossa altura (subimos ou descemos, + ou -), assim como a intensidade (magnitude numérica: 50cm ou 70 cm). A partir daí, definimos uma regra para movimentação.

Como vimos, podemos encarar a rede neural como uma sequência de funções plugadas. Se o primeiro nó tem q(x,y), o segundo, f, tem valor f(q(x,y) ou f o q.

$q(x, y) = 3x + 2y$ #camada inferior
$f(z) = z^2$ #camada superior
$f(q(x, y)) = q^2 = (3x + 2y)^2$ # input inferior para superior

Podemos calcular o efeito de mudanças inter nodos com a regra de cadeia funções compostas. Isto é, podemos obter o gradiente de erro no nodo de hierarquia mais alta (f), com respeito a uma das variáveis de entrada (x) na hierarquia mais baixa. A operação é computacionalmente barata, bastando multiplicar as derivadas parciais dos erros em cada parte.

$$\frac{df}{dx} = \frac{df}{dq}\frac{dq}{dx}$$

É possível calcular de forma recursiva, portanto local e paralela, ao longo das camadas. Fazendo o mesmo acima para df/dy, teremos os valores de $[df/dx; df/dy]$ que é precisamente nosso gradiente.

```
# Valor duplo (x,y) para inputs
x=1
y=3
q = 3*x + 2*y # primeira camada
f = q^2 # segunda camada
# Backprop - Mudanças em hierarquia superior
# dadas por entradas de camadas inferires
dfdq = 2*q # derivada de x^2 ; variação de f em função de q
dqdx = 3 # Derivada de 3x ; variação de q em função de x
dqdy = 2 # Derivada de 2x ; variação de q em função de y
# Obter gradiente de f(x,y) multiplicando as parciais
dfdx = dfdq*dqdx
dfdy = dfdq*dqdy
grad = c(dfdx,dfdy)
> grad
[1] 24 16
```

Usando essa lógica, calculamos os gradientes para a função de erro e treinamos o modelo.

Referência Para uma história completa: J. Schmidhuber. Deep Learning in Neural Networks: An Overview. Neural Networks, 61, p 85–117, 2015. (Based on 2014 TR with 88 pages and 888 references, with PDF & LATEX source & complete public BIBTEX file).

http://web.csulb.edu/~cwallis/artificialn/History.htm https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html https://rpubs.com/FaiHas/197581