Redes Neurais e Deep Learning

Redes neurais

Uma rede neural busca padrões em massivas bases de dados. O exemplo clássico de uma rede neural é o recocinecimento de padrões. Aqui iremos usar o reconhecimento de dígitos escritos à mão. O primeiro passo é alimentar essa RN com diversos padrões (dados) que seriam colocados nos neurônios de entrada (a depender do número de pixels da imagem) e depois iremos padronizar esse tipo de entrada com uma camada de 10 neurônios de saída.

A fórmula dos pesos de cada conexão som os neurônios da nossa rede se dá pela seguinte relação $(1) : N_o = E_1W_1 + E_2W_2 + ... + E_nW_n + b$.

Importante notar que cada neurônio de entrada se comunica uma vez com cada neurônio de saída. Assim, como temos uma image imagem de exemplo de 784px com 10 neurônios de saída e 10 neurônios de viés, temos uma quantidade de 7850 variáveis, dadas pela soma $(784 * 10) + 10$.

Como a equação (1) pode resultar em um número muito grande, tanto negativo como positivo, iremos utilizar a fuinção sigmóide para parametrizar essa saída para uma intervalo entre 0 e 1.

A conta em (1) então será repetida em todos os neurônios da RN em cada conexão W utlizando um peso diferente, primeiramente de forma aleatória, para buscar o melhor parâmetro que maximize o nosso modelo. Assim, o maior valor conseguido em (1) vai nos indicar o neurônio que mais vezes foi ativado com aquele padrão. Um valor igual a um seria uma simulação perfeita.

Quando formos determinar o peso de cada uma das conexões feitas pelos neurônios, o que temos que nos atentar é qual parte do pixel aciona qual neurônio e criar, assim, uma padronização dos pesos que nos gere um modelo omitizado que reconheça as diversas variações possíveis para um número escrito à mão.

Tomando como exemplo um 4 e um 9 percebemos que temos dois padrões bastante parecidos de números e, assim, teremos neurônios ativados com valores parecidos e próximos. Nesse exemplo, devemos nos focar nas diferenças sutis entre ambos, dando mais peso para as regiões que determinam a diferença dos dois dígitos do que para as regiões que apontam as semelhanças. Uma forma de calibar essa rede para entender essa sutileza é atribuir um valor negativo a parte superior do pixel, assim, pixels com valores negativos naquela região indicariam um 4 e pixels com valores positivos naquela região indicariam se tratar de um 9.

Essa calibração não é feita manualmente, claro. Como temos mais de 7 mil variáveis, seria impossível conseguir dar conta de tudo. Mesmo hnuma rede simples (de uma camada com 10 neurônios como essa).

A nossa estratégia aqui vai ser iniciar com valores aleatórios - que pode ser gerado usando um $rand$ em Python, por exemplo. Esses neurônios irão receber valores entre 0 e 1 que nos indicarão, incialmente, o erro desse modelo aleatório - e espera-se um erro grande - quando combinados com o gabarito que se espera nos neurônios de saída.

Assim teremos a tabela abaixo com a representação dos neurônios da nossa RN:

INPUT	PESO		OUTPUT	PESO
N0	0.26	-	N0	0.00
N1	0.05	-	N1	0.00
N2	0.18	-	N2	1.00
N3	0.89	-	N3	0.00
N4	0.23	-	N4	0.00
N5	0.08	-	N5	0.00
N6	0.74	-	N6	0.00
N7	0.52	-	N7	0.00
N8	0.41	-	N8	0.00
N9	0.33	-	N9	0.00

Quando temos uma enmtrada 2 a saída esperada é a da direita, com um valor de 1 para essa imagem. O valor da esquerda é o nosso valorm gerador aleatoriamente. A diferença entre o esperando e o recebido é o nosso erro. A partir do erro podemos calcular a nossa $f(x)$ do custo computacional para essa rede.

$$N_2 = \sigma(E_1W_1 + ... + E_nW_n) \tag{1}$$

$$Custo_2 = \Sigma(erro)^2 \tag{2}$$

$$Custo_2 = (0.26 - 0)^2 + (0.05 - 0)^2 + ... + (0.18 - 0)^2 + 1 \tag{3}$$

$$Custo_2 = 2.05 \tag{4}$$

E assim, podemos escrever essa função de custos como

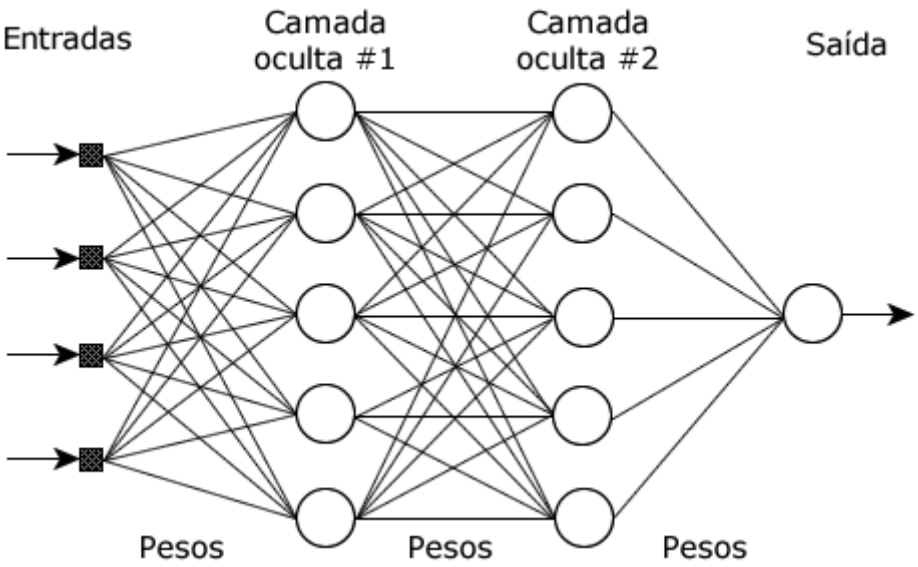
$$\Sigma C_t = Custo_t \tag{5}$$

Diminuindo a tabela da esquerda da tabela da direita, teremos o erro aproximado. O quadrado do somatório dos erros nos dará o custo/erro total da nossa rede. No caso especifico dessa primeira iteração, obtivemos um erro de 2.05 (alto), o ideal é nos aproximarmos de um erro 0. Assim, a cada nova rodada (iteração) iremos aprimorando o nosso modelo de modo que ele se aproxime de zero.

Assim, podemos escrever a nossa $f(x)$ como sendo $Custo = \Sigma\Sigma(erros)^2$

Deep learning

Quando falamos de deep learning estamos falando de uma rede neural com mais de uma camada intermediária. Quando mais neurônios intermediários, mais "profunda" é a nossa rede neural.



Nesse exemplo acima temos a camada oculta 1 $(NC1)$ e a camada oculta 2 $(NC2)$ que tem seus pesos calculados da seguinte forma:

$$NC1 = \sigma(\Sigma EW_E + b_e) \tag{6}$$

$$NC2 = \sigma(\Sigma NC_1 W_1 + b_1) \tag{7}$$

$$NC3 = \sigma(\Sigma NC_3 W_3 + b_3) \tag{8}$$

Nessa rede poderemos ter:

- 30 neurônios em cada camada oculta
- Cara neurônio se comunicando com todos os neurônios da próxima camada (cada conexão com um peso diferente)
- Podemos dizer que é um modelo de RN densa, onde cada neurônio se comunica com todos os outros
- Esse modelo tem aproximadamente 25000 variáveis
- Como não entendemos os passos intermediários (camadas ocultas) chamamos esse modelo de "caixa preta"

A matemática da rede neural

Cada neurônio de saída pode ser escrito como $S_j = \sigma(\Sigma W_j * A_m + B_j)$

De forma matricial temos:

$$\begin{bmatrix} S_1 & S_2 & ...S_y \end{bmatrix}$$

$$\begin{bmatrix} A_1 & A_2 & ...A_m \end{bmatrix}$$

$$\begin{bmatrix} W_1 & W_2 & ...W_m \\ W_1 & W_2 & ...W_n \\ ... & & \\ W_1 & W_2 & ...W_z \end{bmatrix}$$

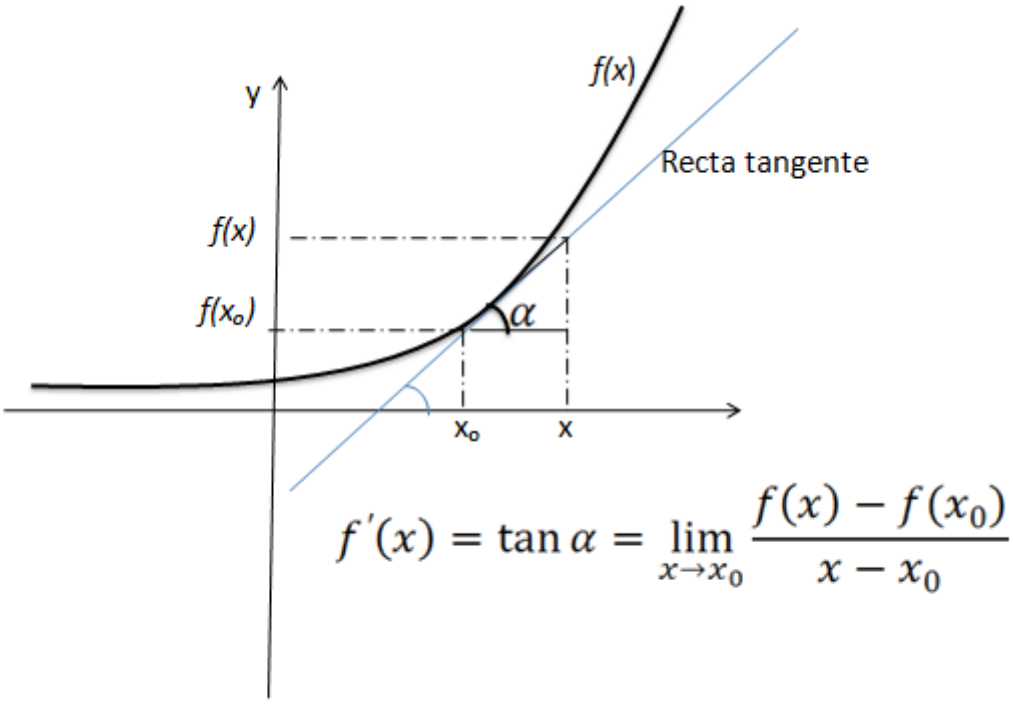
E, por último:

$$\begin{bmatrix} B_1 & B_2 & ...B_m \end{bmatrix}$$

E assim temos que $S_j = \sigma(Wa + b)$ que é uma soma matricial.

A função $f(x)$ do custo vai ser dada por $C = \Sigma(S_j - y_j)^2$ Assim, substituindo $\Sigma(W_j A_m + b_j)$ por S_j temos $S_j = \sigma(S_j \sigma)$

Por exemplo, usando a derivada segunda da função temos:



Assim, na primeira rodada o valor de peso de W_j recebe um valor do peso anterior, $W_m i$ subtraído pela derivada naquele ponto. Ou seja, se a derivada de $\partial(c)/\partial(W_j)$.

Um salto significativo para o peso certo. Contudo, na medida em que nos aproximamos do melhor valor, essa derivada vai ficando menor.

Para a derivada $\partial(c)/\partial(W_j)$, quando maior o valor, pior para o peso W_j. Assim, percebemos que precisaremos de mais uma iteração em $i + 1$ para calibrar o modelo melhor. Assim teremos:

$$\partial(c)/\partial(W_j) = \partial(c)/\partial(S_j) * \partial(S_j)/\partial(W_j) * \partial(S_j\sigma)/\partial(W_j) \tag{9}$$

Substituindo:

$$\partial(c)/\partial(W_j) = 2 * \Sigma(S_j + y + j) * \sigma(\Sigma(S_j\sigma) * \Sigma a_n \tag{10}$$

E, por último, substituindo em $S_J\sigma$:

$$\partial(c)/\partial(W_j) = 2 * \Sigma(S_j + y + j) * \sigma(\Sigma(W_j A_j + b_j) * \Sigma a_n \tag{11}$$

Finalmente, a derivada que atualiza a nossa rede neural depende dos pesos em função do custo. Essa relação vai ser proporcional ao erro (e aos pesos da camada anterior de neurônios).

De forma final, calculamos a derivada da função custo em b:

$$\partial(c)/\partial(b_j) = 2 * \Sigma(S_j + y + j) * \sigma(\Sigma(W_j A_j + b_j) * 1 \quad (12)$$

Ambas as equações são faceis de implementar em Python. De forma análoga calculamos as derivadas das camadas anteriores, assim teremos as funções finais:

Para a camada $W_k m$:

$$\partial(c)/\partial(W_k m) = 2 * \Sigma(S_j + y + j) * \sigma(\Sigma(W_j A_j + b_j) * \Sigma' W_j * \sigma'(\Sigma W_k A_k + b_m) * \Sigma A_k \quad (13)$$

E em relação a b_m:

$$\partial(c)/\partial(W_k m) = 2 * \Sigma(S_j + y + j) * \sigma(\Sigma(W_j A_j + b_j) * \Sigma' W_j * \sigma'(\Sigma W_k A_k + b_m) * 1 \quad (14)$$

Analogamente, temos o mesmo raciocinio para a primeira camada de neurônios, apenas adicionando mais um somatório para essa camada. E em relação ao viés, a última derivada é igual 1, logo, cortamos o último termo.

Como a primeira derivada é a que mais se comunica, uma alteração em um peso inicial na entrada C_1 impacta em toda a rede neural, como é esperado.

Gradiente descendente estocástico

Esse gradiente serve para contornar problemas computacionais. A ideia dele é chegar ao ponto de mínimo da função. Assim, com cada iteração damos um passo em direção a esse ponto de mínimo através da derivada parcial. O GDE usa uma fração dos dados pata otimizar a função, ainda que esse não seja o menor valor possível. Contudo, a discussão toda recai no custo computacional envolvido para obter o valor ótimo (o melhor valor possível) e o quanto isso custará (quantas iterações). Quanto mais iterações, maior o custo: $mini - batch \rightarrow trade - off$.

Assim, quanto maior o batch (lote), maior é a demora; quanto menor o batch (lote), mais rápido. O que nos indica que o mini-batch deve ser ajustado de acordo com o problema que estamos modelando.