Universidade Estadual Paulista
FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA
JÚLIO DE MESQUITA FILHO

Departamento de Engenharia Mecânica

Estudo do *Perceptron* e aplicações de *SLP* e *MLP* em P*ython*

Disciplina: Tópicos especiais em termociências: computação científica com Python

Docente: Emanuel Rocha Woiski

Discente: Vínicius Roberto de Farias Costa-RA:132053551

Ilha Solteira - SP, Dezembro de 2017

RESUMO

O presente trabalho buscou realizar um estudo do *Perceptron* e desenvolver uma implementação em *Python* do mesmo. A aplicação desenvolvida apresenta duas partes: i) uma sobre o *perceptron* de uma única camada (*SLP*) e ii) a outra sobre o *perceptron* de múltiplas camadas (*MLP*). Na primeira parte, o objetivo foi treinar o *SLP* a realizar a soma de dois números reais passados a ele. Na segunda parte, o objetivo foi treinar o *MLP* a calcular a tangente hiperbólica de um valor real passado a ele. Para isso, em ambos os casos (*SLP e MLP*), utilizou-se o método do *gradiente descendente* para minimizar o *erro total* gerado pelo perceptron. No caso do *SLP*, o erro convergiu para o seu valor mínimo. No caso do *MLP*, o erro também convergiu para seu valor mínimo, no entanto de forma mais lenta do que no caso do *SLP*. Os algorítmos implementados no presente trabalho podem ser utilizados, em trabalhos futuros, para ensinar o perceptron a aproximar um conjunto de pontos no plano ou no espaço, com tendência linear ou não-linear.

INTRODUÇÃO TEÓRICA

PERCEPTRONS

O *Perceptron* (ou, mais precisamente, o *Perceptron* de uma única camada: *Single Layer Perceptron, SLP)* é um modelo matemático simplificado (inicialmente desenvolvido por Frank Rosenbatt) do que se acredita ser o funcionamento básico de um neurônio.

De forma simplificada, acredita-se que um neurônio se comporte da seguinte maneira: recebe um conjunto de sinais de entrada e, caso a combinação desses sinais ultrapasse um certo valor limite (*threshold),* o neurônio *dispara*, isto é, emite um sinal para o próximo ou próximos neurônios conectados a ele. Uma vez ultrapassado o threshold, no entanto, o neurônio sempre dispara com a mesma intensidade.

Embora a forma precisa como um neurônio combina os sinais de entrada possa ser bastante complexa, no modelo do *SLP* admite-se que cada sinal de entrada x_i tem associado a ele um peso, w_i, que quantifica a influência de cada sinal sobre o *perceptron*, de modo que o *output* gerado pelo *perceptron* é sempre uma função da soma:

$$\sum_{i=1}^{n} x_i w_i = X \cdot W \tag{1}$$

, onde n representa o número de sinais ou dados de entrada (*input*) e ponto representa o produto escalar.

Ou seja, quando alimentado com um vetor de sinais de entrada $X=(x_1, x_2, \dots x_n)$, ao qual está associado um vetor de pesos $W=(w_1, w_{2,} \dots w_n)$, o efeito ou *output*, *y*, gerado pelo *perceptron* é dado por:

$$y=f(X \cdot W) \tag{2}$$

, onde *f* é chamada de *função de ativação*.

A função de ativação mais simples é a função identidade, caso em que os *outputs* são dados simplesmente por:

$$y=X \cdot W \tag{3}$$

No entanto, mais abrangente em suas aplicações do que a função identidade, é a função linear:

$$y=X \cdot W+b \tag{4}$$

, onde *b* é uma constante que pode ser entendida como *bias,* ou seja, a tendência inicial ou "pré-disposição" do *perceptron*.

Outra função de ativação bastante comum – e que, teoricamente, mais se parece com o funcionamento do neurônio, uma vez que a intensidade do sinal de entrada não influencia a intensidade do sinal de saída – é a função degrau dada por:

$$y=\begin{cases} 0, \ se \ X \cdot W < threshold \\ 1, \ se \ X \cdot W \geq threshold \end{cases} \tag{5}$$

, o que é equivalente a:

$$y=\begin{cases} 0, \ se \ X \cdot W - threshold < 0 \\ 1, \ se \ X \cdot W - threshold \geq 0 \end{cases} \tag{6}$$

Para simplificar a notação, podemos unir o *threshold* ao vetor *X* e, para não alterar o resultado de (6), adicionar uma nova dimensão, inicializada com o valor -1, ao vetor dos pesos. Ou seja, fazemos $X=(x_1, x_2, \dots x_n, threshold)$ e $W=(w_1, w_{2,} \dots w_n, -1)$, de modo que (6) se torna simplesmente:

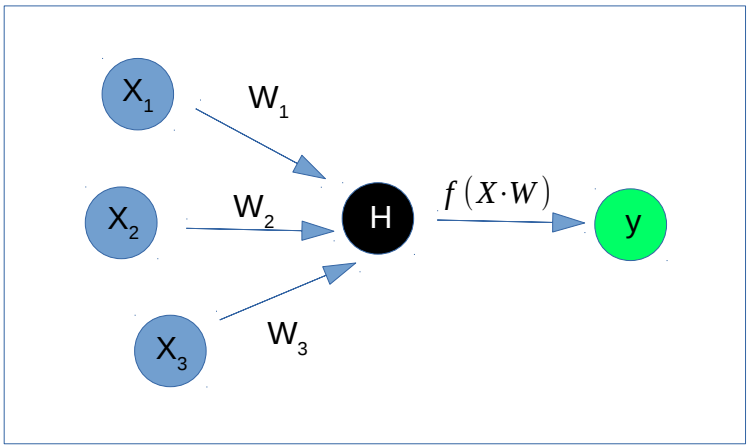
$$y=\begin{cases} 0, \ se \ X \cdot W < 0 \\ 1, \ se \ X \cdot W \geq 0 \end{cases} \tag{7}$$

Utilizando o mesmo procedimento, a função linear, dada pela equação (4), por sua vez, pode ser substituída pela função identidade, mantendo sua generalidade:

$$y = X_b \cdot W_b \tag{8}$$

, onde agora $X_b = (x_1, x_2, \ldots x_n, b)$ e $W_b = (w_1, w_2, \ldots w_n, 1)$.

Figura 1. Representação gráfica de um perceptron.



Fonte: elaborado pelo próprio autor

Na figura 1, está representado graficamente o funcionamento de um *perceptron*: os círculos em azul formam a camada de entrada ou *input layer, o* círculo verde representa a camada de saída ou *output layer* e o círculo preto representa a chamada "camada escondida" ou *hidden layer,* por não se tratar nem de *input* nem de *output* de dados.

ENSINANDO O PERCEPTRON

Para cada função de ativação e vetor de entrada predefinidos, o valor de *y* depende apenas do vetor de pesos *W*. Assim, variando-se os valores de cada componente w_i do vetor *W*, varia-se também o *output* produzido.

Dessa forma, pode-se ajustar o vetor *W* de modo que, para cada vetor X_j, o output produzido *y* seja um determinado *valor esperado* e_j associado a X_j. Podemos assim formar uma tupla dos pares (X_j, e_j), chamada de conjunto de treinamento ou aprendizado (*training set*). Por exemplo, caso se desejasse ensinar ao *perceptron* a função lógica *AND*, o training set seria: *(((0, 0), 0), ((0, 1), 0), ((1, 0), 0), ((1, 1), 1))*.

A pergunta então se torna: como ajustar os pesos de modo que o *perceptron* retorne a resposta esperada, e de uma forma eficiente?

A forma mais utilizada consiste em se ajustar os pesos de modo a minimizar o erro total, dado por:

$$E(W)=\sum_{j=1}^{s}(y_j-e_j)^2 \tag{9}$$

, onde *s* é o número de elementos do training set. Notamos que *E (W)* é sempre maior ou igual a zero e, portanto, seu mínimo global teórico é *E = 0*. No entanto, na prática, dadas as características do *training-set*, o mínimo global de *E(W)* em geral é maior do que zero. Além disso, nem sempre é fácil ou possível de se obter o mínimo *global* de *E(W)* por métodos analíticos. Dessa forma, comumente utilizam-se métodos numéricos para se obter mínimos *locais* de *E(W)*. Em geral, no entanto, esses mínimos locais se aproximam bastante do mínimo global.

Para minimizar a função definida por (9), um dos métodos numéricos comumente utilizados é o do *gradiente descendente*, que consiste em se levar o ponto $W_b=(w_1, w_{2,}...w_n,1)$ do espaço de fases de dimensão *n+1*, da forma mais rápida possível (direção oposta ao gradiente de $E(W)$) para o ponto de mínimo de $E(W)$. O gradiente de $E(W)$ com respeito a W é dado por:

$$\nabla_w E=(\frac{\partial}{\partial w_1}\sum_{j=1}^{s}(y_j-e_j)^2,\ ...,\ \frac{\partial}{\partial w_n}\sum_{j=1}^{s}(y_j-e_j)^2)=(\sum_{j=1}^{s}\frac{\partial}{\partial w_1}(y_j-e_j)^2,\ ...,\ \sum_{j=1}^{s}\frac{\partial}{\partial w_n}(y_j-e_j)^2)$$

Ou seja,

$$\nabla_w E=(\sum_{j=1}^{s}2(y_j-e_j)\frac{\partial y_j}{\partial w_1},\ ...,\ \sum_{j=1}^{s}2(y_j-e_j)\frac{\partial y_j}{\partial w_n}) \tag{10}$$

, uma vez que e_j não é função de *w*.

Cada valor y_j , por sua vez, é função dos pesos por meio de (2), $y_j=f(X_j\cdot W)$, de modo que:

$$\frac{\partial y_j}{\partial w_i}=\frac{\partial f}{\partial(X_j\cdot W)}\frac{\partial(X_j\cdot W)}{\partial w_i}=\frac{\partial f}{\partial(X_j\cdot W)}x_{ji} \tag{11}$$

, onde o termo x_{ji} corresponde ao i-ésimo elemento do j-ésimo vetor do training set. Para se determinar completamente (11), é necessário conhecer a função de ativação *f*. Caso *f* seja a própria função identidade, temos simplesmente:

$$\frac{\partial y_j}{\partial w_i}=x_{ji} \tag{12}$$

, e a equação (10) se torna:

$$\nabla_w E = \left(\sum_{j=1}^{s} 2(y_j - e_j) x_{j1},\ \ldots,\ \sum_{j=1}^{s} 2(y_j - e_j) x_{jn} \right) \tag{13}$$

A equação (13) é comumente utilizada em problemas de classificação ou regressão linear.

Caso o vetor de entrada de dados, e consequentemente, o vetor de pesos, contiverem apenas 3 elementos (dois referentes aos dados de entrada “em si” e o outro referente à bias), a equação (13) se torna:

$$\nabla_w E = \left(\sum_{j=1}^{s} 2(y_j - e_j) x_{j1}, \sum_{j=1}^{s} 2(y_j - e_j) x_{j2}, \sum_{j=1}^{s} 2(y_j - e_j) x_{j3} \right) \tag{13.a}$$

Utilizando o método do *gradiente descendente*, fornecemos o mesmo *training-set* um certo de número de vezes ou “rodadas” para realizar o treinamento do *perceptron*, o que corresponde, matematicamente, a aproximar o vetor *W* ao ponto de mínimo de *E(W)*. É importante que se passe o *mesmo training-set* várias vezes para o *perceptron*, pois, do contrário, em geral não se obtém convergência, uma vez que quando se altera o *training-set,* alteram-se os pontos de mínimo de *E(W)*. Dessa forma, ao fim de cada rodada, fazemos:

$$W = W - \eta \nabla_w E \tag{14}$$

, onde η define o tamanho do passo com o qual o vetor *W* se aproxima do ponto de mínimo de *E(W)*. Caso não se conheça uma forma de estimar um valor apropriado para η, sua determinação se faz por tentativa e erro. Deve-se notar que se o valor de η for muito pequeno, o vetor *W* demora para atingir o ponto de mínimo de *E(W)*. Se for muito grande, o vetor *W* “viaja” pelo espaço de fases sempre ultrapassando o ponto de mínimo, sem nunca convergir para o mesmo. Dessa forma, η deve ser ajustado de modo que se obtenha a convergência do vetor *W* e que isso se dê da forma mais rápida possível.

MULTI-LAYER PERCEPTRON E BACKPROPAGATION

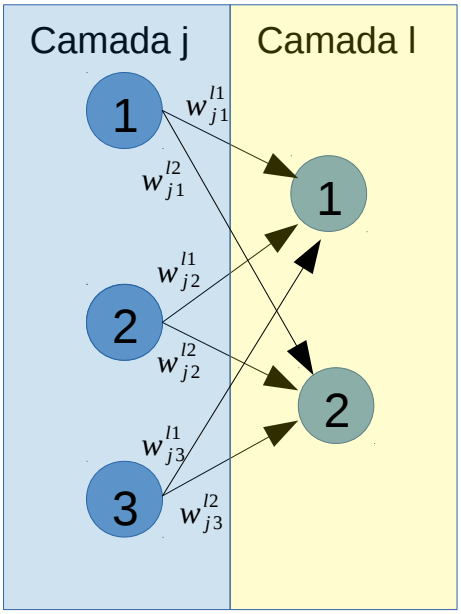
O *perceptron* de uma única camada é limitado quanto às suas aplicações, pois somente é capaz realizar de regressões lineares, isto é, somente consegue aproximar (ou separar) dados por meio de um hiperplano. Quando se deseja aproximar adequadamente dados que não apresentam comportamento linear, é necessário empregar um perceptron

de múltiplas camadas (*Multi-Layer Perceptron – MLP*), que nada mais é do que um perceptron com duas ou mais camadas escondidas (*hidden layers*).

Cada elemento que recebe dados de outros elementos da rede neural e os propaga por meio de uma função de ativação é definido como uma *unidade* da rede neural.

De forma semelhante ao *SLP*, para realizar o treinamento da rede, desejamos minimizar o *erro total* ajustando os pesos. No entanto, no caso do *MLP*, existem múltiplas camadas, e cada camada pode conter várias *unidades*, de modo que para cada par qualquer de *unidades* de camadas adjacentes da rede neural, está associado um peso (conforme representado na figura 2, entre as camadas hipotéticas *j* e *l*)

Figura 2. Representação gráfica de duas camadas hipotéticas adjacentes, *j* e *l*, e todos os pesos, *w*, associados aos seus elementos.



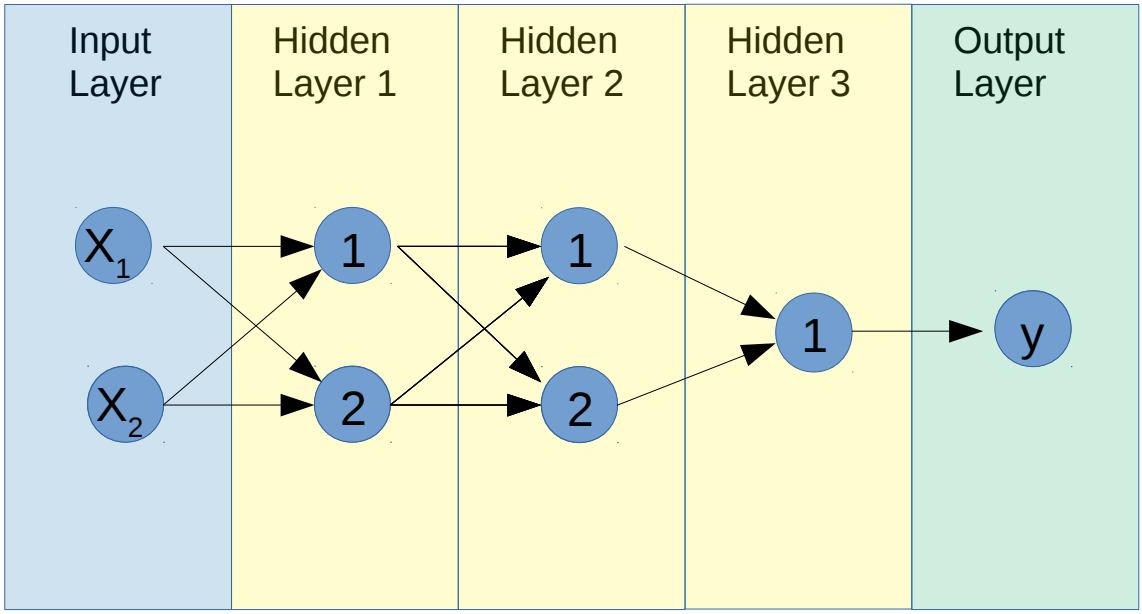
Fonte: elaborado pelo próprio autor

Assim, para minimizar o erro total *E*, precisamos determinar a derivada parcial de *E* em relação a cada peso w_{ji}^{lk} . Esse processo é mais complexo do que no caso do *SLP* e é conhecido como *Backpropagation* ou *Retropopagação*, que consiste, grosso modo, em se aplicar a regra da cadeia repetidamente através de todos os caminhos possíveis que são influenciados pelo peso em relação ao qual se está derivando.

Inicialmente, para tornar mais clara a ideia da retropropagação, vamos exemplificar como se obtém a derivada do erro total em relação a um determinado peso arbitrário, em um *MLP* hipotético com três hidden layers. Nesse *MLP*, as duas primeiras hidden layers possuem duas unidades cada, e a terceira, apenas uma. Além disso, esse

MLP receberá dois dados de entrada, ou seja, a camada de entrada terá duas unidades (veja a figura 3).

Figura 3. Representação do *perceptron* hipotético que servirá como exemplo para a demonstração da retropropagação.



Fonte: elaborado pelo próprio autor.

Para isso, definiremos uma notação que facilitará posteriormente a generalização do resultado obtido e a programação do *MLP* em *Python*.

Da mesma forma que no *SLP*, chamaremos de *y* o output produzido pelo *perceptron*. Numeraremos as camadas do *SLP* a partir da camada de entrada, que será a camada 1. Assim, na sequência, a primeira *hidden layer* será a camada 2, a segunda, a camada 3 e terceira, a camada 4. A camada de saída de dados será então a camada 5.

Dessa forma, chamaremos de x_j o j-ésimo valor de entrada. Além disso, para especificar o peso que sai da unidade i da camada j e vai para a unidade k da camada l, adotaremos a seguinte notação: w_{ji}^{lk} .

Chamaremos o valor que serve de entrada para a unidade i da camada j de s_{ji}. De forma correspondente, chamaremos a função de ativação dessa mesma unidade de f_{ji}.

Suponha então que desejamos calcular a derivada do erro gerado E_j por um determinado vetor de entrada X_j do *training-set* em relação ao peso w_{12}^{21} , ou seja, o peso que sai da unidade 2 da camada 1 e vai para a unidade 1 da camada 2.

Primeiramente, o erro quadrático causado por um vetor de entrada genérico X_j do *training-set*, é dado por:

$$E_j=(y_j-e_j)^2 \tag{15}$$

, onde, como antes, e_j é o output esperado para o vetor X_j do *training-set,* e y_j, de acordo com as definições acima, é dado por:

$$y_j=w_{31}^{41}f_{31}(s_{31})+w_{32}^{41}f_{32}(s_{32}) \tag{16}$$

, onde, por sua vez,

$$s_{31}=w_{21}^{31}f_{21}(s_{21})+w_{22}^{31}f_{22}(s_{22}) \tag{17}$$

, e

$$s_{32}=w_{21}^{32}f_{21}(s_{21})+w_{22}^{32}f_{22}(s_{22}) \tag{18}$$

Por sua vez, ainda,

$$s_{21}=w_{11}^{21}x_1+w_{12}^{21}x_2 \tag{19}$$

, e

$$s_{22}=w_{11}^{22}x_1+w_{12}^{22}x_2 \tag{20}$$

De (15) temos:

$$\frac{\partial E_j}{\partial w_{12}^{21}}=2(y_j-e_j)\frac{\partial y_j}{\partial w_{12}^{21}} \tag{21}$$

, onde, usando (16),

$$\frac{\partial y_j}{\partial w_{12}^{21}}=w_{31}^{41}\frac{\partial f_{31}}{\partial w_{12}^{21}}+w_{32}^{41}\frac{\partial f_{32}}{\partial w_{12}^{21}}=w_{31}^{41}\frac{\partial f_{31}}{\partial s_{31}}\frac{\partial s_{31}}{\partial w_{12}^{21}}+w_{32}^{41}\frac{\partial f_{32}}{\partial s_{32}}\frac{\partial s_{32}}{\partial w_{12}^{21}} \tag{22}$$

Mas, usando (17) e (18), chegamos a:

$$\frac{\partial s_{31}}{\partial w_{12}^{21}}=w_{21}^{31}\frac{\partial f_{21}}{\partial w_{12}^{21}}+w_{22}^{31}\frac{\partial f_{22}}{\partial w_{12}^{21}}=w_{21}^{31}\frac{\partial f_{21}}{\partial s_{21}}\frac{\partial s_{21}}{\partial w_{12}^{21}}+w_{22}^{31}\frac{\partial f_{22}}{\partial s_{22}}\frac{\partial s_{22}}{\partial w_{12}^{21}} \tag{23}$$

, e

$$\frac{\partial s_{32}}{\partial w_{12}^{21}}=w_{21}^{32}\frac{\partial f_{21}}{\partial w_{12}^{21}}+w_{22}^{32}\frac{\partial f_{22}}{\partial w_{12}^{21}}=w_{21}^{32}\frac{\partial f_{21}}{\partial s_{21}}\frac{\partial s_{21}}{\partial w_{12}^{21}}+w_{22}^{32}\frac{\partial f_{22}}{\partial s_{22}}\frac{\partial s_{22}}{\partial w_{12}^{21}} \tag{24}$$

Mas, de (19) e (20):

$$\frac{\partial s_{21}}{\partial w_{12}^{21}}=x_2 \tag{25}$$

, e

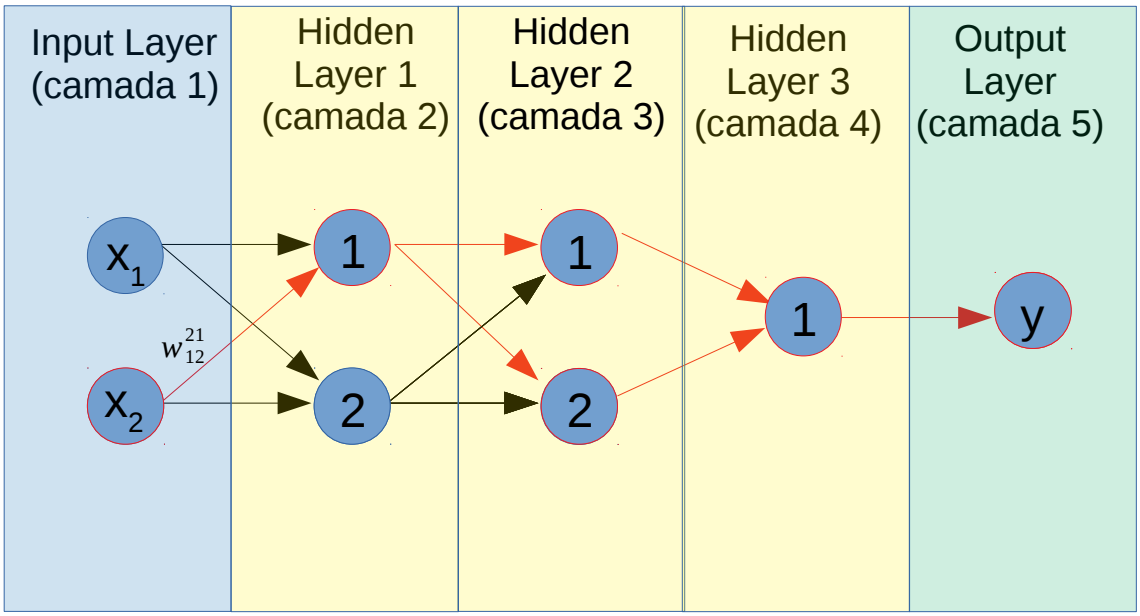
$$\frac{\partial s_{22}}{\partial w_{12}^{21}}=0 \tag{26}$$

Assim, combinando as equações (21), (22), (23), (24), (25) e (26), chegamos finalmente ao resultado desejado:

$$\frac{\partial E_j}{\partial w_{12}^{21}}=2\left(y_j-e_j\right)x_2\left(w_{31}^{41}\frac{\partial f_{31}}{\partial s_{31}}w_{21}^{31}\frac{\partial f_{21}}{\partial s_{21}}+w_{32}^{41}\frac{\partial f_{32}}{\partial s_{32}}w_{21}^{32}\frac{\partial f_{21}}{\partial s_{21}}\right) \tag{27}$$

A forma de se obter (27) pode então ser pensada da seguinte maneira: a partir do output, y_j, nos "movemos" para trás ao longo do perceptron (daí o nome *retropropagação*), aplicando a regra da cadeia ao longo de todos os caminhos *que são influenciados* pelo peso em relação ao qual se está derivando e, em seguida, somamos as derivadas obtidas para cada caminho. Isso fica claro em (27), pois aparecem apenas os pesos w_{31}^{41} , w_{32}^{41} , w_{21}^{31} e w_{21}^{32} , ou seja, apenas os pesos que *pertencem* ao caminho influenciado por w_{12}^{21} . O mesmo acontece com as derivadas das funções de ativação f_{ij} em relação às entradas s_{ij}. Como era esperado, da mesma forma, aparece somente o valor x_2, pois o peso w_{12}^{21} está conectado somente a esse valor de entrada. Na figura 4, destacamos em vermelho o "caminho influenciado" por w_{12}^{21} .

Figura 4. "Caminhos influenciados" por w_{12}^{21} .



Fonte: elaborado pelo próprio autor.

Podemos então, com base nessas observações, generalizar a derivada parcial do erro E_j em relação a um peso arbitrário conectado *à camada de entrada*, w_{1k}^{lm} , para um determinado vetor de entrada qualquer X_j do *training-set*:

$$\frac{\partial E_j}{\partial w_{1k}^{lm}} = 2(y_j - e_j) x_{jk} \sum_s \left[\prod_p \left(w \frac{\partial f}{\partial s} \right)_{s,p} \right] \tag{28}$$

, onde $\left(w \frac{\partial f}{\partial s} \right)_{s,p}$ representa o p-ésimo termo $w \frac{\partial f}{\partial s}$ do s-ésimo caminho possível (a partir da output layer, de trás para frente) influenciado por w_{1k}^{lm} .

Notamos, no entanto, que a equação (28) não representa o caso mais geral possível, pois o peso em relação ao qual se está derivando está conectado a uma unidade da camada de entrada. Caso o peso partisse de uma camada intermediária (*hidden layer*), depois de aplicarmos seguidamente a regra da cadeia a partir do output, o último termo a ser derivado seria:

$$\frac{\partial}{\partial w_{ik}^{lm}} (w_{ik}^{lm} f_{ik}(s_{ik})) = f_{ik}(s_{ik}) \tag{29}$$

, em vez de x_k. Dessa forma, substituindo (29) no lugar de x_k em (28), teríamos:

$$\frac{\partial E_j}{\partial w_{ik}^{lm}} = 2(y_j - e_j) f_{ik}(s_{ik}) \sum_s \left[\prod_p \left(w \frac{\partial f}{\partial s} \right)_{s,p} \right] \tag{30}$$

A equação (30) nos permite obter a derivada de E_j em relação a cada peso da rede neural. Ou seja, podemos obter o vetor gradiente $\nabla_w E_j$ para cada elemento *j* do *training-set* ou, fazendo a somatória de todos os gradientes, obter o gradiente do *erro total* para *todo* o *training-set*:

$$\nabla_w E = \sum_j \nabla_w E_j \tag{31}$$

A equação (31) nos permite então atualizar o vetor de pesos *W* na direção oposta ao gradiente, de forma semelhante ao *SLP*:

$$W = W - \eta \nabla_w E \tag{32}$$

DESENVOLVIMENTO

SINGLE LAYER PERCEPTRON – SLP

A presente aplicação busca treinar o *SLP* a realizar a soma de um par de números. Ou seja, dado um par qualquer de números reais *(a, b)*, esperamos que o *perceptron* gere sempre como output *a+b*. Embora a tarefa realizada pelo perceptron nesse caso seja bem simples e não exija o uso de um *perceptron*, ela serve para ilustrar como implementar um *SLP* para realizar outras tarefas não tão simples como regressão ou classificação linear.

O *training-set* é do tipo: $(([x_{11}, x_{12}], x_{11} + x_{12}), ([x_{21}, x_{22}], x_{21} + x_{22}), \ldots, ([x_{s1}, x_{s2}], x_{s1} + x_{s2}))$, onde s é o número de elementos do training-set. Para implementar essa aplicação, utilizamos o seguinte vetor de entrada, *X* (chamado de vetor das características), com $s = 100$ pares de números aleatórios, e unimos a ele um "vetor coluna" de 1's como vetor de bias, *b*:

```
s = 100
X = np.random.rand(s, 2)
b = np.ones(s)[:, None]
X = np.hstack((X, b))
```

Assim, o vetor dos valores esperados, e, que corresponde à soma de dois números, fica:

```
e = X.T[0]+X.T[1]
```

, de modo que nosso *training-set* é dado por (X_s, e_s).

Definimos também o número de rodadas ou vezes (*num_iter*) que o mesmo *training-set* alimentará o perceptron para o treinamento, bem como a taxa de aprendizagem, *n*, obtida por tentativa e erro e que corresponde àquela em que se obtém a convergência o mais rápido possível:

```
num_iter = 1000
n = 0.0065
```

Os pesos foram então inicializados com zeros:

```
w = np.zeros(3)
```

Assim, podemos realizar o treinamento do *perceptron*. Para isso, alimentaremos o perceptron *num_iter* vezes com o mesmo *training-set*. Cada elemento

grad[k] do vetor *grad* definido abaixo corresponde a *k-ésima* coordenada do vetor $\nabla_w E$ (definido pela equação 13.a), já multiplicada pela taxa de aprendizagem, *n*. Em cada rodada, varia-se *w* na direção contrária ao gradiente de *E*, de modo a levar *E* ao seu valor mínimo:

```
for i in range(num_iter):
  grad = (((2 * n * (np.dot(X, w) - e))[:, None]) * X).sum(axis=0)
  w -= grad
  print('rodada ', i + 1, ': ', 'vetor w:', w)
```

Buscou-se implementar o treinamento da forma mais "vetorizada" possível, de modo a se otimizar o código por meio de *numpy arrays*, com as regras de *broadcast,* diminuindo assim o tempo de processamento. O laço de *for* restante é indispensável neste caso. Podemos tentar realizar o treinamento sem ele, da seguinte forma: ao invés de se passar o mesmo *training-set* várias vezes, o que exige o laço de *for,* podemos aumentar o número de elementos do training set, e passá-lo ao perceptron uma única vez. No entanto, não se obtém convergência desse modo, conforme mostrado no arquivo *SLP.ipynb*, pelas razões já expostas na introdução teórica.

Depois de concluído o treinamento, o vetor de pesos convergiu para:

```
rodada  1000 :  vetor w: [  1.00000000e+00   1.00000000e+00  -5.16416078e-16]
```

Notamos que o resultado obtido para o vetor de pesos está próximo do esperado, pois matematicamente, dado um par de números x_1, x_2, o perceptron gera: $y = w_1x_1 + w_2x_2 + b$. Logo, para realizar simplesmente a soma de dois números, w_1 e w_2 devem convergir para 1 e *b*, para 0.

MULTIPLE LAYER PERCEPTRON – MLP

O objetivo da segunda aplicação foi ensinar o *perceptron* a calcular a tangente hiperbólica de um valor real qualquer passado a ele, usando a tangente hiperbólica como função de ativação das *hidden-layers*. Para isso, a princípio, desenvolvemos um *MLP* com duas *hidden-layers*, contendo a primeira delas duas unidades e a segunda, uma única unidade, uma vez que está conectada diretamente à camada de *output*. As duas unidades da primeira *hidden-layer* tiveram como função de ativação a tangente hiperbólica e a unidade da segunda hidden-layer teve como funcão de ativação a função linear. Assim, matematicamente, o *perceptron* tenta aproximar uma tendência não-linear combinando duas tangentes hiperbólicas. Com isso em mente, para testar o *MLP* implementado,

tentamos treinar o *MLP* para, a cada valor *x* passado a ele, retornar *tanh (x)*, pois temos certeza da *possibilidade* de convergência nesse caso (uma vez que estamos usando *tanh( )* como função de ativação).

O algoritmo desenvolvido para esse objetivo apresenta apenas duas *hidden-layers*, de forma fixa. O número de unidades da camada de entrada e da primeira *hidden-layer*, no entanto, podem ser modificados pelo usuário. Um algoritmo que admita um número genérico de camadas pode ser desenvolvido em um trabalho futuro.

Calculando o output

Inicialmente, organizamos os dados em matrizes de forma a vetorizar o código para o cálculo do output gerado pelo *perceptron*. A forma genérica para o vetor das características utilizado para o treinamento é:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1s} \\ x_{21} & & & x_{2s} \\ \vdots & & & \vdots \\ x_{(n1)1} & x_{(n1)2} & \cdots & x_{(n1)s} \end{bmatrix}$$

, onde *s* é o número de elementos do training-set e *n1*, o número de dados de entrada ou unidades da camada de entrada ou camada 1.

No caso particular implemetado em *Python*, há apenas um elemento de entrada, ou seja, a *input-layer* tem dimensão 1. Dessa forma, o vetor das características e o vetor dos *outputs* esperados (tangente hiperbólica dos valores de entrada) foram implementados da seguinte forma:

```
X=np.random.rand(s)[None,:]
e=np.tanh(X)
```

O vetor de pesos entre as camadas 1 e 2, ou seja, entre a camada de entrada e a primeira *hidden-layer* será definido por:

$$W_{12} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1(n1)} \\ w_{21} & & & w_{2(n1)} \\ \vdots & & & \vdots \\ x_{(n2)1} & x_{(n2)2} & \cdots & x_{(n2)(n1)} \end{bmatrix}$$

, onde *n2* é o número de unidades da camada 2. Notamos que o elemento $(w_{12})_{ij}$ de W_{12} representa o peso que conecta o *j-ésimo* elemento da primeira camada (*input-layer*) ao i-*ésimo* elemento da segunda camada (primeira *hidden-layer*).

No nosso caso particular, o vetor W_{12} implementado em *Python* foi:

```
W12=np.ones(l2*l1).reshape(l2,l1)
```

, onde *l1* é número de unidades da camada 1, ou seja, da *input-layer* e *l2*, o número de unidades da camada 2, ou seja, a primeira *hidden*-layer. Ambos podem ser definidos pelo usuário.

Os valores de entrada, *s*, para cada unidade *i* da camada 2, para cada elemento *j* do training-set, podem ser organizados na matriz $(s_2)_{ij}$. (S_2 por se tratar da entrada de dados para a segunda camada) Matematicamente, notamos que:

$$S_2 = W_{12} . X$$

, onde o ponto representa a multiplicação matricial.

Em *Python*, fizemos:

```
S2=np.dot(W12,X)
```

Em seguida, aplicamos a função de ativação, F_2 (F_2 por se tratar da função de ativação associada a segunda camada), *element-wise* ao vetor S_2, isto é, obtemos o vetor F_2. Os elementos $(f_2)_{ij}$ do vetor F_2 representam os valores de saída de cada unidade *i* da camada 2 para cada elemento *j* do training-set.

Em *Python*:

```
F2=np.tanh(S2)
```

Neste ponto, precisamos adicionar o efeito de *bias*, que corresponde, geometricamente, ao deslocamento ao longo do eixo *y* da curva ou superfície *y(X)*. Dessa forma, o vetor de *bias* deve ser introduzido depois da aplicação da função de ativação. Por isso, unimos ao vetor F_2, um "vetor linha" *b*, de tamanho *s* (*s* sendo o número de elementos do *training-set*) inicializado com 1's.

```
F2 = np.vstack((F2, b))
```

Finalmente, multiplicamos o vetor F_2 pelo vetor W_{23} (*element-wise* e não de forma matricial; por isso, representaremos esse produto por $F_2 * W_{23}$), sendo W_{23} o vetor de pesos que conectam a camada 2 à camada 3:

$$W_{23}=\begin{bmatrix} w_{11} \\ w_{21} \\ \vdots \\ w_{(n2+1)1} \end{bmatrix}$$

Em *Python*:

```
W23=np.ones((l2*l3)+1).reshape(l2+1,l3)
```

Notamos que o vetor W_{23} possui *n2+1* colunas, pois o peso $w_{(n2+1)1}$ multiplicará cada elemento do "vetor linha" *b* de bias unido ao F_2.

Cada coluna *j* do vetor $F_2 * W_{23}$ contém as *n2+1* entradas de valores para a terceira camada, para o *j-ésimo* elemento do *training-set*. Logo, se somarmos os elementos de $F_2 * W_{23}$ ao longo das linhas (*sum (axis=0)*), obtemos o vetor S_3, contendo apenas uma linha, em que cada elemento *i* representa o valor de entrada total para a camada 3 quando alimentamos o perceptron com o elemento *i* do *training-set*. Como a função de ativação da camada 3 é a função identidade (por ser a última *hidden-layer*), temos que os valores de output y_j são iguais às entradas $(s_3)_j$ da camada 3 para cada elemento *j* do training-set. Ou seja: $Y = S_3$. Dessa forma, conhecidos todos os pesos, determinados o output *Y*:

```
Y = (F2 * W23).sum(axis=0)
```

Backpropagation

Uma vez determinado o *output* gerado pelo *perceptron*, passamos à determinação do gradiente utilizando *backpropagation*. Utilizamos para isso as equações (28) e (30), desenvolvidas anteriormente. Primeiro determinamos o gradiente associado ao vetor W_{23} e, em seguida, propagamos o resultado obtido para trás (*backpropagation*) para calcular o gradiente associado ao vetor W_{12}.

Para determinar o gradiente, *grad*(W_{23}), associado ao vetor W_{23}, utilizamos a equação (30), notando que, neste caso, para qualquer w_{2k}^{3m} :

$$\sum_s \left[\prod_p \left(w \frac{\partial f}{\partial s} \right)_{s,p} \right] = 1$$

Logo, temos:

$$\frac{\partial E_j}{\partial w_{2k}^{3m}} = 2(y_j - e_j) f_{2k}(s_{2k})$$

Como estamos calculando o gradiente associado ao vetor W_{23}, podemos simplificar a notação, escrevendo:

$$\frac{\partial E_j}{\partial w_{km}} = 2(y_j - e_j) f_{2k}(s_{2k})$$

, onde w_{km} representa o peso conectando a unidade *k* da camada 2 com a unidade *m* da camada 3. Nesse caso, como a camada 3 é a última hidden-layer, ela possui apenas uma unidade, de modo que ficamos com:

$$\frac{\partial E_j}{\partial w_{k1}} = 2(y_j - e_j) f_{2k}(s_{2k})$$

Assim, para obter $\frac{\partial E_j}{\partial w_{k1}}$ para cada w_{k1} para cada elemento do *training-set*, isto é, obter o vetor $grad(W_{23})$, podemos fazer:

$$grad(W_{23}) = 2(Y - e) * F2$$

, onde *e* representa o vetor dos outputs esperados associado ao vetor das características *X*. Notamos que o elemento $(grad(W_{23}))_{ij}$ representa o valor $\frac{\partial E_j}{\partial w_{i1}}$ **.** Logo, se somarmos os elementos de $grad(W_{23})$ ao longo das colunas (*sum (axis = 1)*), obtemos um "vetor linha" em que cada elemento *i* representa o *gradiente total* associado ao peso w_{i1} entre as camadas 2 e 3, calculado sobre todos os elementos do *training-set.* Em *Python*, fazemos:

```
gradW23=(2*(Y-e)*F2).sum(axis=1)[:,None]
```

Fazemos o *[: , None]* para tornar possível o *broadcast* com o vetor W_{23}.

Assim, podemos atualizar o vetor W_{23}, pois o mesmo será utilizado no cálculo do gradiente associado ao vetor W_{12}, $grad(W_{12})$:

```
W23=W23-n*gradW23
```

O sinal negativo na expressão acima indica que levamos o vetor peso no sentido contrário ao do gradiente, pois o objetivo é levar o vetor peso ao ponto que minimize o erro total.

Passamos agora ao cálculo de *grad(W_{12})*. Utilizando a equação (28), temos:

$$\frac{\partial E_j}{\partial w_{1k}^{2m}} = 2(y_j - e_j)\, x_{kj} \sum_s \left[\prod_p \left(w \frac{\partial f}{\partial s} \right)_{s,p} \right]$$

Mas, nesse caso, temos:

$$\sum_s \left[\prod_p \left(w \frac{\partial f}{\partial s} \right)_{s,p} \right] = w_{2m}^{31} f_m'(s_m)$$

Ou seja,

$$\frac{\partial E_j}{\partial w_{1k}^{2m}} = 2(y_j - e_j)\, x_{kj}\, w_{2m}^{31} f_m'(s_m)$$

Como a função de ativação utilizada é a *tanh( )*, temos:

$$f_m'(s_m) = \tanh'(s_m) = 1 - \tanh^2(s_m)$$

Logo:

$$\frac{\partial E_j}{\partial w_{1k}^{2m}} = 2(y_j - e_j)\, x_{kj}\, w_{2m}^{31} (1 - \tanh^2(s_m))$$

Ou seja, para cada elemento *j* do training-set, temos o vetor:

$$(grad(W_{12}))_{km} = 2(y_j - e_j)\, x_{kj}\, w_{2m}^{31} (1 - \tanh^2(s_m))$$

Em *Python*:

```
gradW12 = 2 * (((((Y - e) * X) * W23[0:2, :]) * (1 - (np.tanh(S2)) ** 2)).sum(axis=1))[:, None]
```

Fizemos o *slice* no vetor W_{23} pois, para o cálculo de *grad(W_{12})*, a terceira linha (que contém o peso relativo a vetor de bias) não é necessária. De forma semelhante ao cálculo de *grad(W_{23})*, somamos ao longo das linhas para obter o gradiente total levando-se em conta todos os elementos do *training-set*. Em seguida, fizemos *[: , None]* para possibilitar o *broadcast* com W_{12}:

```
W12=W12-n*gradW12
```

Esse procedimento é repetido um certo número (*num_iter*) de vezes ou rodadas, da mesma forma que no *SLP*.

Depois de concluído o treinamento (com *num_iter = 10000*), os vetores pesos convergiram para:

```
rodada 9999 :
 W12:
 [[ 1.00000905]
 [ 1.00000905]]

 W23:
 [[  4.99999980e-01]
 [  4.99999980e-01]
 [ -2.27078743e-09]]
```

Ou seja, obtivemos aproximadamente $W_{12} = [1 , 1]$ e $W_{23} = [0.5, 0.5, 0]$. Assim, dado um valor de entrada real *x*, o *perceptron*, após o treinamento, retorna (destacamos em vermelho os pesos obtidos):

$$y = 0.5*\tanh(1*x) + 0.5*\tanh(1*x) + 0 = \tanh(x)$$

, conforme o esperado. Notamos, no entanto, que a convergência, nesse caso, é mais lenta em relação ao *SLP*, o que se deve ao fato de a taxa de aprendizagem ter que ser menor para possibilitar a convergência ($n = 0.00005$ contra $n = 0.0065$).

CONCLUSÃO

O presente trabalho desenvolveu com sucesso um *SLP* capaz de aprender a realizar a soma de dois números reais. O erro obtido convergiu para o seu valor mínimo rapidamente. No caso do *MLP*, também foi possível ensinar ao perceptron a calcular a tangente hiperbólica de um valor real *x: y = tanh (x)*, utilizando-se duas *hidden-layers,* com função de ativação *tanh( )*. A convergência, no caso do *MLP*, é mais lenta em relação ao *SLP*, pois a mesma só é possível a uma taxa de aprendizagem menor. Trabalhos futuros podem tentar ensinar o perceptron a aproximar um conjunto de pontos do plano ou do espaço com tendência linear ou não-linear, utilizando o algorítmo desenvolvido no presente trabalho. Além disso, pode-se tentar generalizar o código do *MLP* implementado para um número arbritário de hidden layers.

REFERÊNCIAS

HUMPHRYS, M. *Single-layer Neural Networks (Perceptrons)*. Disponível em: *http://computing.dcu.ie/~humphrys/Notes/Neural/single.neural.html*

ORR, G. B. *Error Backpropagation*. Disponível em: http://www.willamette.edu/~gorr/classes/cs449/backprop.html

ORR, G. B. *CS-449: Neural Networks*. Disponível em: http://www.willamette.edu/~gorr/classes/cs449/intro.html