Redes Neurais – Parte II

1. Perceptron de Múltiplas Camadas (MLP)

- Se utilizarmos neurônios do tipo *perceptron* numa única camada, seu potencial de atuação será limitado pela já mencionada capacidade de separação linear. Com a adição de camadas intermediárias, essa limitação pode ser superada. Esse ponto é fundamental para que se compreenda a razão de ser da rede neural conhecida como *perceptron de múltiplas camadas* (MLP, do inglês *multilayer perceptron*). Um exemplo de rede desse tipo, com duas camadas intermediárias, é dado na Fig. 1.

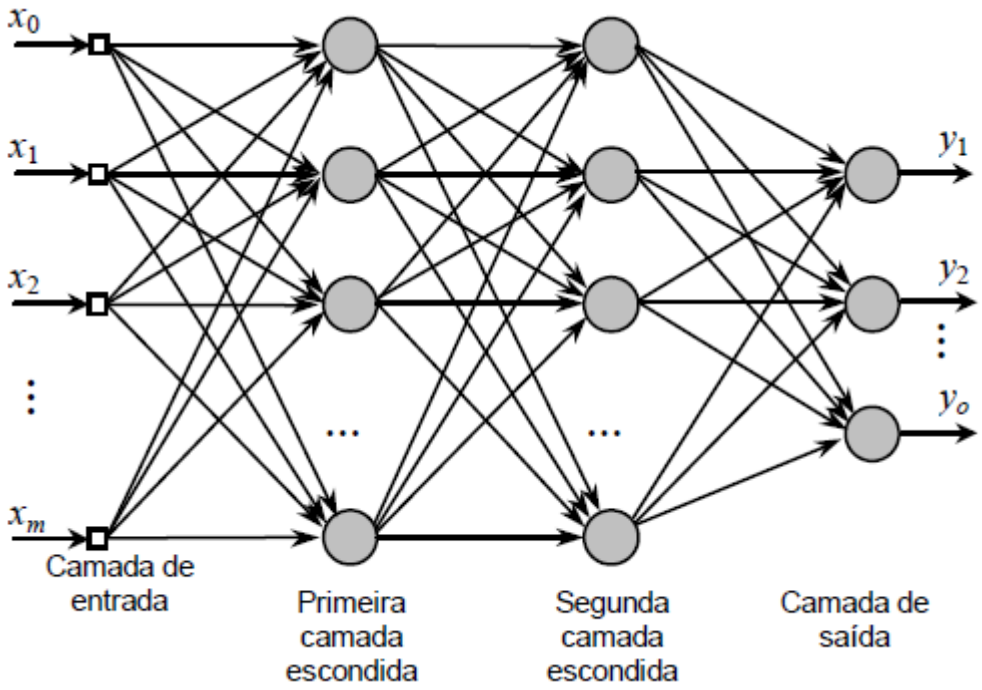

Fig. 1 – Exemplo de uma rede MLP com duas camadas intermediárias (ou escondidas, ocultas).

- A *camada de entrada* nada mais é que a ilustração da passagem dos atributos à rede. As *camadas intermediárias* realizam mapeamentos não-lineares que, idealmente, vão tornando a informação contida nos dados mais "explícita" do ponto de vista da tarefa que se deseja realizar. Por fim, os neurônios da *camada*

de saída combinam a informação que lhes é oferecida pela última camada intermediária.

1.1. Funções de Ativação

- MLPs são *perceptrons de múltiplas camadas*: portanto, naturalmente, tais redes têm por base o modelo de neurônio do *perceptron*. Esse modelo, discutido no tópico anterior, é novamente ilustrado na Fig. 2.

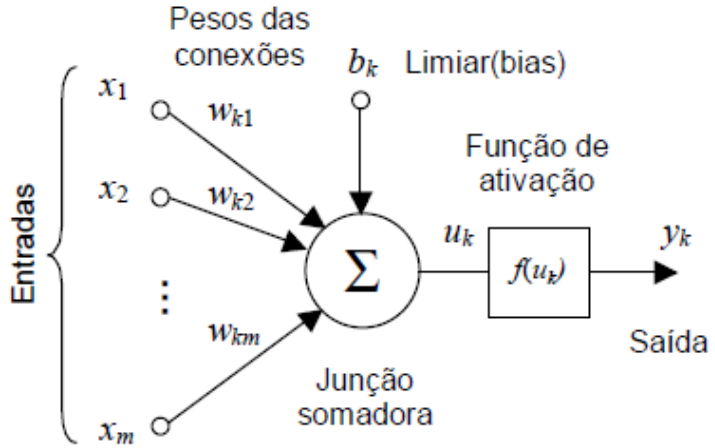

Fig. 2 – Modelo do *perceptron*.

- Não é usual empregar a função degrau "estrita" como função de ativação em MLPs. Até o advento das redes profundas, a regra era utilizar duas funções que são, em essência, "degraus suaves": a função logística e a função tangente hiperbólica. A função logística tem a seguinte expressão:

$$y_k = f(u_k) = \frac{e^{pu_k}}{e^{pu_k} + 1} = \frac{1}{1 + e^{-pu_k}}$$

- Sua derivada* é:

$$\frac{dy_k}{du_k} = py_k(1 - y_k) > 0$$

- Essa função e sua derivada são mostradas na Fig. 3. Note que o parâmetro p torna o "degrau" mais ou menos suave.

* A derivada será importante, como veremos, no processo de aprendizado da rede.

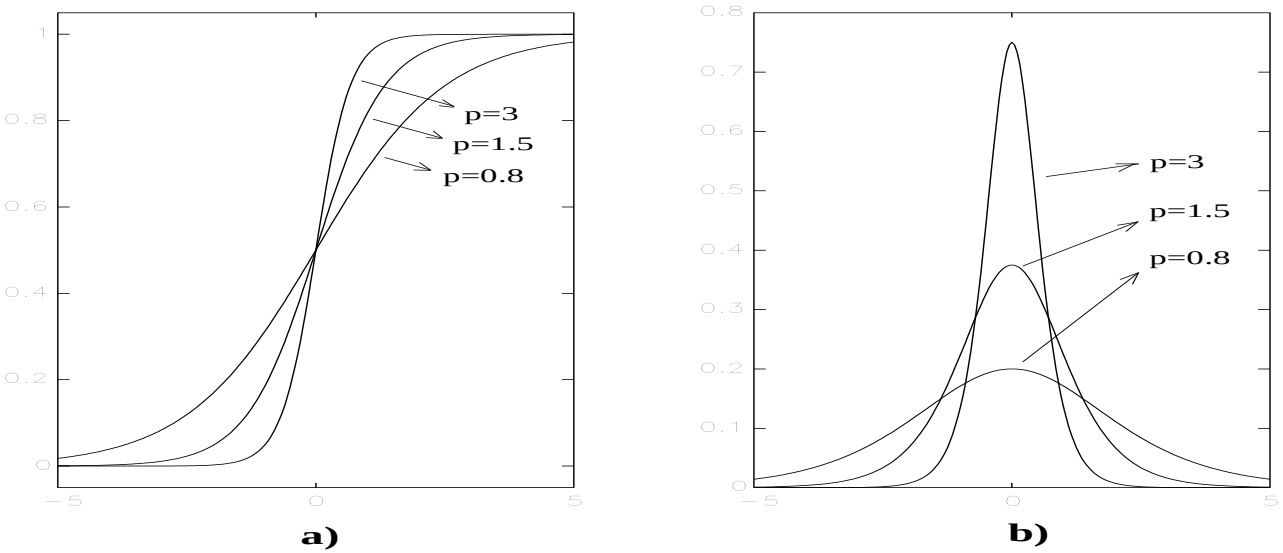

Fig. 3 – Função Logística e sua Derivada.

- A tangente hiperbólica tem a seguinte expressão:

$$y_k = f(u_k) = \tanh(pu_k) = \frac{e^{pu_k} - e^{-pu_k}}{e^{pu_k} + e^{-pu_k}}$$

- A derivada é:

$$\frac{dy_k}{du_k} = p(1 - (y_k)^2) > 0$$

- Essa função e sua derivada são mostradas na Fig. 4. Mais uma vez, o parâmetro p controla a suavidade do degrau.

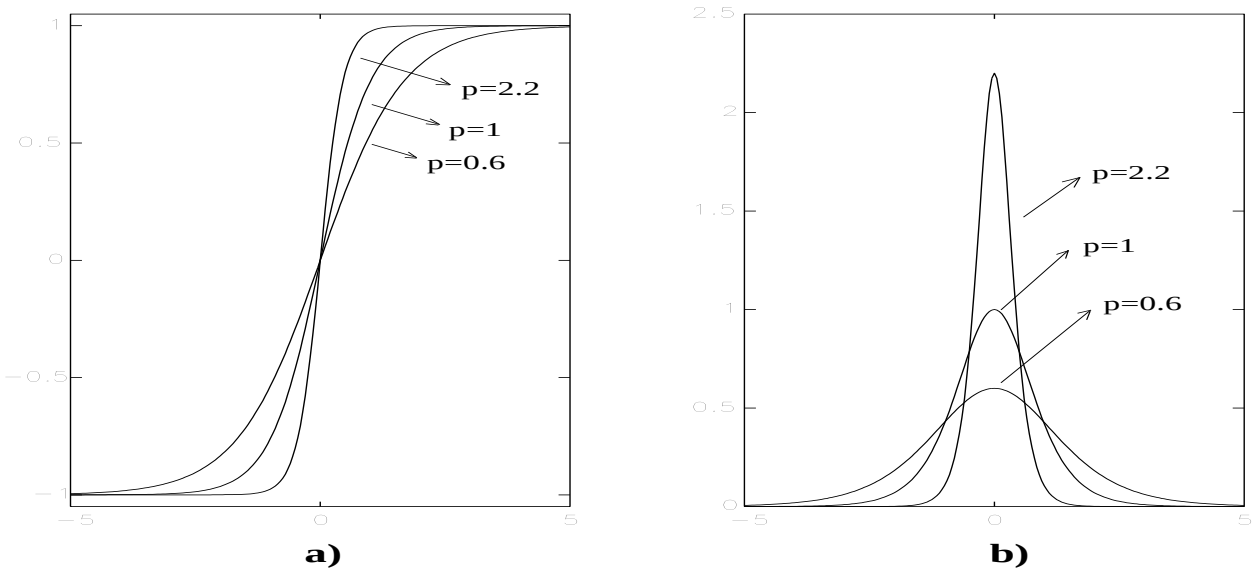

Figura 4 – Função Tangente Hiperbólica e sua Derivada.

- Embora as duas funções apresentadas sejam clássicas na área de redes neurais, com o advento de redes profundas, outra função passou a ser a preferida por uma série de questões numéricas e computacionais, a *função retificadora*†:

$$y = f(u_k) = \max(0, u_k)$$

† Um neurônio dotado de função de ativação retificadora é chamado de ReLU (*rectified linear unit*).

- Essa função é ilustrada na Fig. 5. Uma versão modificada é a que leva à *leaky* ReLU:

$$y = f(u_k) = \begin{cases} \max(0, u_k)\,, se\ u_k > 0 \\ \sigma u_k, \quad \text{alhures} \end{cases}$$

sendo σ um valor pequeno (e.g. 0,01) (MAAS ET AL., 2013). A modificação evita que uma eventual nulidade do gradiente impeça uma unidade de sofrer adaptação em certo período de tempo.

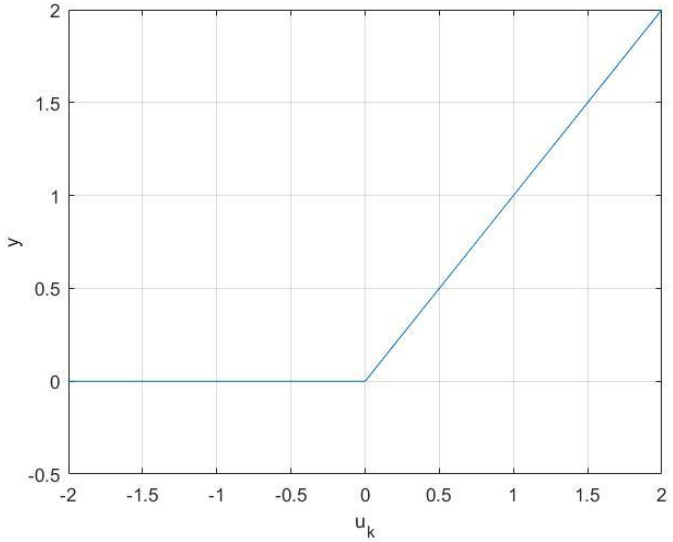

Fig. 5 – Função Retificadora.

1.2. Estudo de Caso: MLPs com uma Camada Intermediária

- Para que possamos aprofundar nossa visão acerca de uma MLP, consideremos o caso de uma rede com apenas uma camada intermediária e camada de saída linear. A estrutura é mostrada na Fig. 6.
- Percebe-se que cada saída dessa rede é composta de uma combinação linear de saídas da camada intermediária. As saídas da camada intermediária, por sua vez, são funções não-lineares das entradas. Matematicamente,

$$\widehat{s_k} = \sum_{j=1}^{n} w_{kj} f\left(\sum_{i=0}^{m} v_{ji} x_i\right) + w_{k0} = \sum_{j=1}^{n} w_{kj} f\left(\mathbf{v}_j^T \mathbf{x}\right) + w_{k0} = \mathbf{w}_k^T \mathbf{f}\left(\mathbf{v}_j^T \mathbf{x}\right) + w_{k0}$$

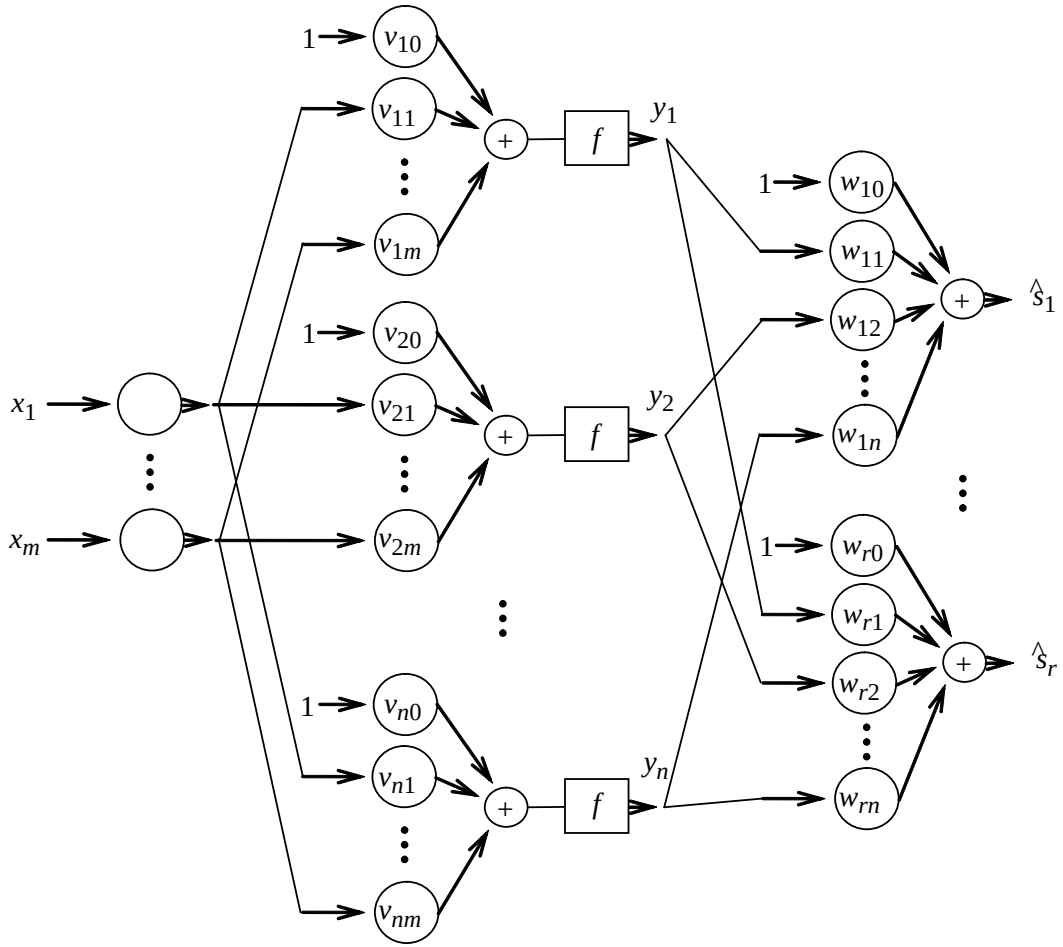

Fig. 6 – MLP com uma Camada Intermediária.

- É interessante analisarmos o tipo de mapeamento gerado por cada neurônio, ou seja, as funções que são linearmente combinadas pela camada de saída. A saída do j-ésimo neurônio da camada intermediária é:

$$y_j = f(\mathbf{v}_j{}^T \mathbf{x})$$

- Percebemos que a função de ativação $f(\cdot)$ determina o "perfil" do mapeamento. Já os pesos sinápticos $\mathbf{v}_j$ determinam a "orientação" do mapeamento gerado. A Fig. 7 ilustra isso.

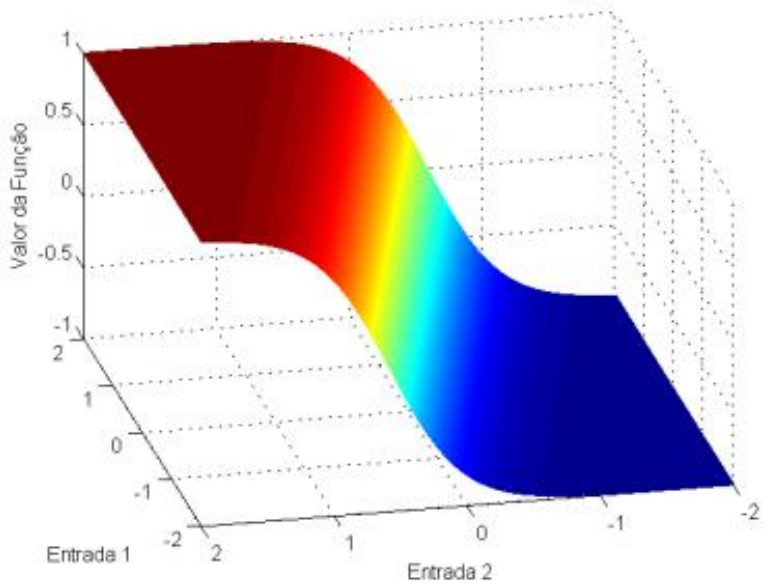

Figura 7 – Função Gerada por um Neurônio.

- Funções desse tipo são chamadas de *funções de expansão ortogonal* (*ridge functions*). Variando os pesos da camada intermediária e os pesos da camada de saída, forma-se um repertório de funções com diferentes escalas e orientações. A partir desse repertório, por meio de algoritmos de aprendizado, a rede neural constrói mapeamentos capazes de resolver problemas. Um mapeamento simples é mostrado na Fig. 8.

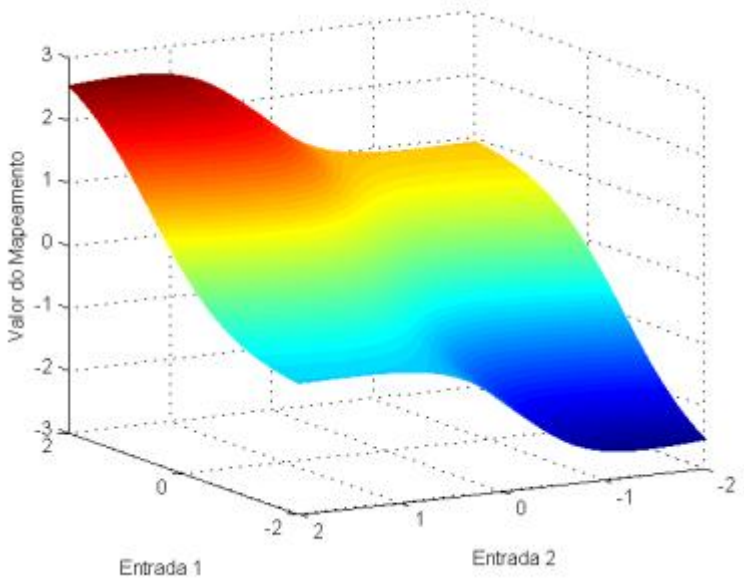

Fig. 8 – Mapeamento com Duas Funções de Expansão Ortogonal.

1.3. Capacidade de Aproximação Universal

- Redes MLP possuem o apelo da inspiração biológica, mas não seriam amplamente utilizadas se não tivessem significativo potencial de aproximar mapeamentos entrada-saída.
- Essas redes possuem *capacidade de aproximação universal*, ou seja, são capazes de aproximar qualquer mapeamento contínuo num domínio compacto com um nível de erro arbitrariamente pequeno. Interessantemente, mesmo nossa modesta rede com uma camada intermediária (com função de ativação tangente hiperbólica, por exemplo) e camada de saída linear já possui essa capacidade, como mostrado por Cybenko (1989). Vejamos esse bonito resultado em mais detalhe (VON ZUBEN, 2007).

- Partamos de uma função de ativação $f(\cdot)$ contínua, não-constante, limitada e monotonicamente crescente (tanto a tangente hiperbólica quanto a função logística são possibilidades). Consideremos que a camada intermediária da rede seja composta de M neurônios tipo *perceptron* com essa função de ativação. Imaginemos que se deseje aproximar uma função contínua $g(x_1, x_2, \ldots, x_m)$ que tenha por domínio o hipercubo unitário m-dimensional $\mathbf{I}_m = (0,1)^m$. Se denominarmos o mapeamento realizado pela rede $F(x_1, x_2, \ldots, x_m)$, o resultado fundamental é que haverá um valor mínimo M e valores de pesos sinápticos tais que:

$$|F(x_1, x_2, \ldots, x_m) - g(x_1, x_2, \ldots, x_m)| < \varepsilon$$

para qualquer $\varepsilon > 0$.

2. Aprendizado em Redes Neurais

- Em aprendizado de máquina, é oportuno pensarmos em estrutura, critério e método de otimização. Em nosso caso, a MLP é a estrutura. Há diferentes critérios, e teremos mais a dizer sobre isso mais adiante. Consideraremos aqui o processo de otimização, ou seja, de adaptação dos pesos sinápticos à luz do critério escolhido.
- Consideremos, sem perda de generalidade, que o problema corresponde a uma tarefa de minimização de uma função custo $J(\mathbf{w})$ com respeito a um vetor de parâmetros $\mathbf{w}$. O problema de aprendizado em redes neurais pode ser formulado como:

$$\min_{\mathbf{w}} J(\mathbf{w})$$

- Geralmente, esse processo de otimização é conduzido de maneira iterativa, o que dá sentido mais natural à noção de aprendizado (como um processo gradual). Há vários métodos de otimização aplicáveis, mas, sem dúvida, os mais usuais são aqueles baseados nas derivadas da função custo.
- Dentre esses métodos, existem os de *primeira ordem* e os de *segunda ordem*. Os métodos de primeira ordem são baseados nas derivadas de primeira ordem da função custo, geralmente agrupadas no vetor gradiente:

$$\nabla J(\mathbf{w}) = \begin{bmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \frac{\partial J(\mathbf{w})}{\partial w_2} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{bmatrix}$$

- O gradiente aponta na direção de maior crescimento da função: caminhar em direção contrária a ele é uma forma adequada de buscar iterativamente a minimização. Destarte, temos a seguinte forma básica:

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \alpha \nabla J[\mathbf{w}(k)]$$

  sendo α o passo de adaptação (*learning rate*) e $\nabla(\cdot)$ o operador gradiente. A escolha do passo é importante e teremos mais a dizer sobre esse tema adiante, mas vale destacar que passos muito grandes podem levar à instabilidade, enquanto passos muito pequenos podem levar a uma convergência demasiadamente lenta (GOODFELLOW ET AL., 2016).
- Os métodos de segunda ordem, por sua vez, têm por base, direta ou indiretamente, também a informação trazida pela segunda derivada da função custo. Essa informação está contida na matriz hessiana $\mathbf{H}$:

$$\mathbf{H}(\mathbf{w}) = \nabla^2 J(\mathbf{w}) = \begin{bmatrix} \frac{\partial^2 J(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 J(\mathbf{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 J(\mathbf{w})}{\partial w_1 \partial w_n} \\ \frac{\partial^2 J(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 J(\mathbf{w})}{\partial w_2^2} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\mathbf{w})}{\partial w_n \partial w_1} & \cdots & \cdots & \frac{\partial^2 J(\mathbf{w})}{\partial w_n^2} \end{bmatrix}$$

- De posse da hessiana, é possível fazer uma aproximação de Taylor de ordem dois da função custo, o que leva à seguinte expressão para adaptação dos pesos:

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \alpha \mathbf{H}^{-1}[\mathbf{w}(k)] \nabla J[\mathbf{w}(k)]$$

  Essa expressão requer que a matriz hessiana seja inversível, e também que seja definida positiva a cada iteração. Se necessário, há métodos numéricos para "forçá-la" a isso.

- Uma vez que a aproximação de Taylor com informação de ordem dois é mais ampla que aquela subjacente aos métodos de primeira ordem, a tendência é que

um método de segunda ordem convirja em menos passos que um método de primeira ordem.

- Entretanto, o cálculo exato da matriz hessiana pode ser complicado em vários casos práticos. Não obstante, há um conjunto de métodos de segunda ordem que evitam esse cálculo direto, como os *métodos quase-Newton* ou os *métodos de gradiente escalonado*, que representam uma espécie de compromisso entre complexidade e desempenho (VON ZUBEN, 2007).

2.1. Mínimos Locais, Mínimos Globais e Pontos de Sela

- Um ponto importante é que todos esses métodos são *métodos de busca local*, ou seja, têm convergência esperada para *mínimos locais*. Para entendermos o que é um mínimo local, vejamos a Fig. 9. Há dois mínimos:

- Um deles é uma solução ótima em relação a seus vizinhos, ou seja, um *mínimo local*.
- O outro também é uma solução ótima em relação a seus vizinhos, mas é, ademais, a solução ótima em relação a todo o domínio considerado. É, dessa forma, um *mínimo global*.

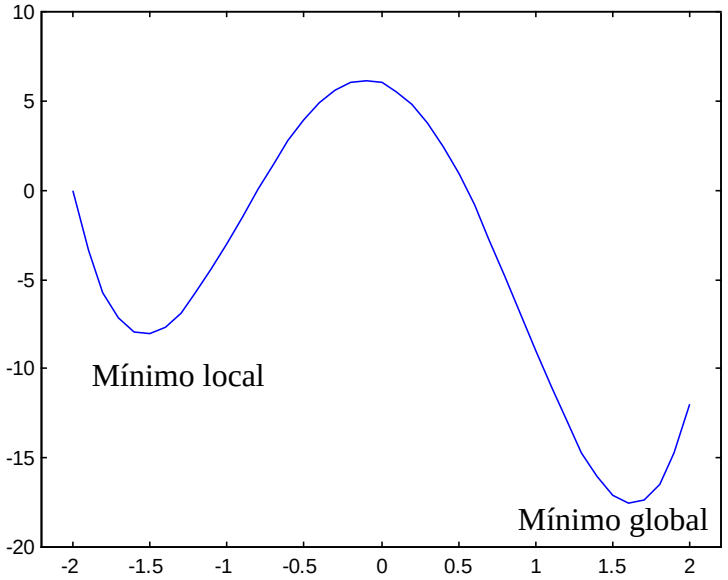

Fig. 9 – Mínimo Local e Mínimo Global.

- Para valores adequados do passo de adaptação α, um mínimo local tende a atrair o vetor de parâmetros quando este se encontra em sua vizinhança. De maneira mais rigorosa, dizemos que cada mínimo tem sua *bacia de atração*.
- Devemos ainda mencionar os chamados *pontos de sela*, que são pontos que, em algumas direções são atratores, mas em outras não. Embora, a longo prazo, o algoritmo não vá convergir para esses pontos, ele pode passar um longo período sendo atraído por eles, o que prejudica seu desempenho. A Fig. 10 mostra um exemplo.

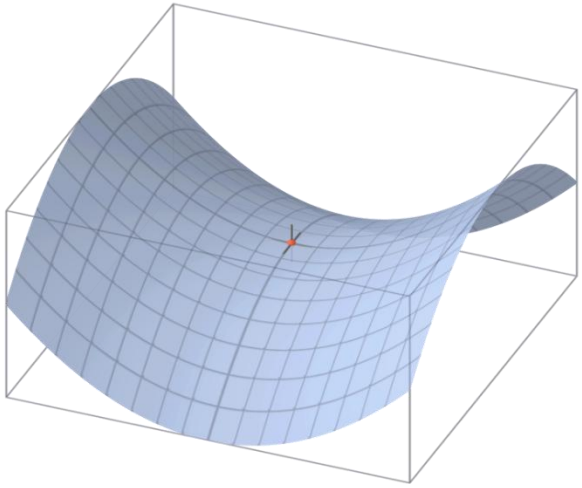

Fig. 10 – Exemplo de Ponto de Sela.

3. Retropropagação de Erro (*Error Backpropagation*)

- Conforme exposto, os métodos fundamentais de aprendizado em redes neurais são baseados no cálculo de derivadas da função custo com respeito aos pesos sinápticos. Busca-se, fundamentalmente, encontrar o conjunto de pesos que minimize a medida de erro escolhida.
- A chave, desse modo, é encontrar uma maneira de calcular o vetor gradiente da função custo com respeito aos pesos sinápticos. A tarefa pode parecer óbvia, mas não é o caso. Para que entendamos o porquê, consideremos uma notação que será doravante valiosa: o peso $w_{i,j}^{m}$ corresponde ao j-ésimo peso do i-ésimo neurônio da m-ésima camada.
- Nessa notação, obter o vetor gradiente significa calcular, de maneira genérica, $\frac{\partial J}{\partial w_{i,j}^{m}}$, ou seja, calcular essa derivada para todos os pesos de todos os neurônios.

- A Fig. 11 ilustra um exemplo de como uma MLP é vista segundo essa concepção.

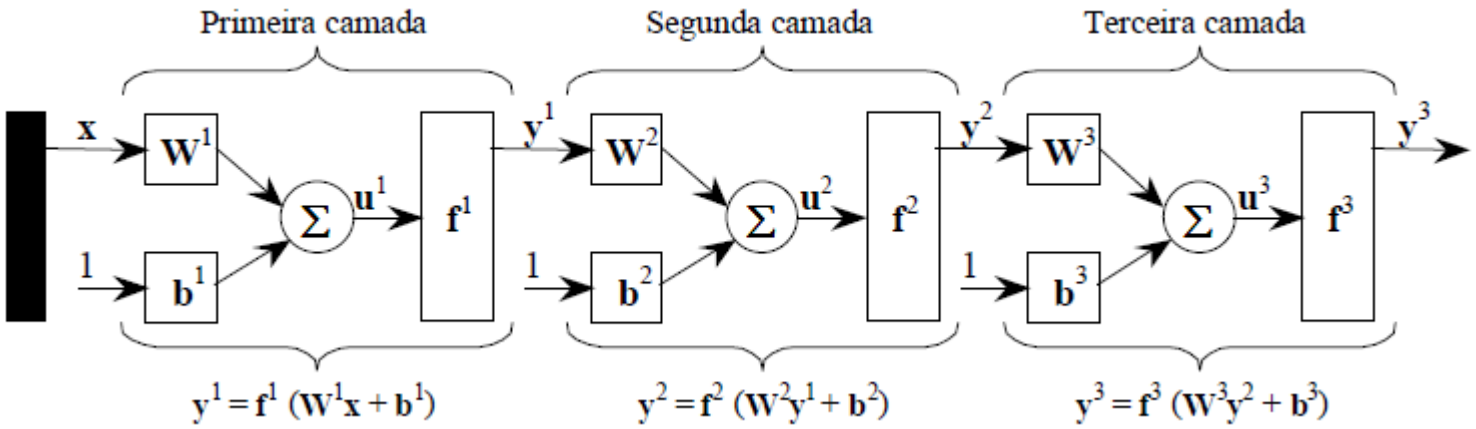

Fig. 11 – Exemplo de MLP e da Notação Usada.

- Note que o mapeamento realizado pela rede, no exemplo, é:

$$\mathbf{y}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

- Para trabalharmos com mais liberdade, suporemos, sem perda de generalidade, que a função custo escolhida é a mais clássica de todas, o erro quadrático médio

(EQM). Assumamos que a última camada da rede (para nós, a M-ésima camada) tenha uma quantidade genérica (N_M) de neurônios.

$$J(\cdot) = \frac{1}{N_{dados}} \sum_{k=1}^{N_{dados}} \sum_{j=1}^{N_M} e_j^2(k) = \frac{1}{N_{dados}} \sum_{k=1}^{N_{dados}} \sum_{j=1}^{N_M} \left(d_j(k) - y_j^M(k)\right)^2$$

- Devemos derivar a função custo com respeito aos pesos, mas estes não aparecem de maneira explícita na expressão de $J(\cdot)$. Para fazer com que sua dependência emerja de maneira clara, será preciso recorrer a aplicações sucessivas da importante *regra da cadeia*. Na elegante notação de Leibniz, essa regra nos informa que:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Por exemplo, considere que tenhamos $z = \exp(x^2)$ e queiramos obter $\frac{dz}{dx}$. Podemos fazer $y = x^2$ e usar a regra da cadeia:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = \exp(y)\,2x = 2x\exp(x^2)$$

- Voltando à nossa expressão do EQM, vemos que as saídas da última camada aparecem de maneira direta. Isso significa que é simples obter as derivadas de $J(\cdot)$ com respeito aos pesos da camada de saída. No entanto, quando se busca avaliar as derivadas com respeito aos pesos das camadas anteriores, a situação é mais complexa, pois não existe mais uma dependência direta. Como atribuir a cada neurônio "longínquo" seu devido crédito na composição da saída e do erro?
- Essa "caminhada de trás para a frente", da saída (na qual se gera o erro) para a entrada, tendo por base a regra da cadeia, corresponde ao processo conhecido como *retropropagação de erro* (*error backpropagation*, ou simplesmente *backpropagation*). Vejamos de maneira mais sistemática como ele se dá.

- Primeiramente, observemos um fato fundamental. Calcular a derivada do erro quadrático médio com respeito a um peso qualquer é‡:

$$\frac{\partial J}{\partial w_{i,j}^{m}} = \frac{\partial \sum_{k=1}^{N_{dados}} \sum_{j=1}^{N_M} e_j^2(k)}{\partial w_{i,j}^{m}} = \sum_{k=1}^{N_{dados}} \sum_{j=1}^{N_M} \frac{\partial e_j^2(k)}{\partial w_{i,j}^{m}}$$

- Isso significa que precisamos calcular a expressão gradiente apenas para o k-ésimo dado, pois o gradiente médio será uma média de "gradientes particulares" (ou "locais") associados a cada amostra.

3.1. Prolegômenos

- Consideremos novamente nossa derivada geral $\frac{\partial J}{\partial w_{i,j}^{m}}$ (um elemento genérico do gradiente). Usando a regra da cadeia, podemos escrever:

‡ Note que omitimos a divisão pelo número de amostras porque isso não afeta a otimização.

$$\frac{\partial J}{\partial w_{i,j}^m} = \frac{\partial J}{\partial u_i^m}\frac{\partial u_i^m}{\partial w_{i,j}^m}$$

- A primeira derivada do produto do segundo membro é a derivada do custo com respeito à ativação do i-ésimo neurônio da m-ésima camada. Essa grandeza será chamada de *sensibilidade* e denotada pela letra grega δ (delta). Assim:

$$\delta_i^m = \frac{\partial J}{\partial u_i^m}$$

- Esse termo é único para cada neurônio. O outro termo, por sua vez, varia ao longo das entradas do neurônio em questão. Uma vez que vale o modelo "tipo perceptron", a ativação é uma combinação linear das entradas:

$$u_i^m = \sum_{j \in entradas} w_{i,j}^m y_j^{m-1} + b_i^m$$

Portanto,

$$\frac{\partial u_i^m}{\partial w_{i,j}^m} = y_j^{m-1}$$

Caso a derivada seja com respeito ao *bias*, ter-se-á:

$$\frac{\partial u_i^m}{\partial b_i^m} = 1$$

- Assim, **<u>todas as derivadas com respeito a pesos sinápticos são produtos de um valor delta por uma entrada (ou, no caso de termos de bias, pela unidade)</u>.**

  Por favor, marque este ponto: voltaremos a ele.

$$\frac{\partial J}{\partial w_{i,j}^m} = \frac{\partial J}{\partial u_i^m}\frac{\partial u_i^m}{\partial w_{i,j}^m} = \delta_i^m y_j^{m-1}$$

ou

$$\frac{\partial J}{\partial b_i^m} = \frac{\partial J}{\partial u_i^m}\frac{\partial u_i^m}{\partial b_i^m} = \delta_i^m$$

- São os valores de delta que trazem dificuldades, pois a outra derivada é trivial (apenas o valor de uma entrada). A estratégia será a seguinte: começaremos pela saída (onde o erro é gerado) e encontraremos uma regra recursiva que gere os valores de delta para os neurônios das camadas anteriores até a primeira camada intermediária. Esse será o processo de retropropagação.

3.2. Retropropagando

- Agruparemos todos os valores δ_i^m de uma camada num vetor $\boldsymbol{\delta}^m$. Encontraremos uma regra que fará a transição $\boldsymbol{\delta}^m \rightarrow \boldsymbol{\delta}^{m-1}$. Calcularemos o vetor da última camada $\boldsymbol{\delta}^M$ e, de maneira recursiva, obteremos os vetores de todas as camadas – esse é o processo de retropropagação (*backpropagation*).
- Calculemos então $\boldsymbol{\delta}^M$. Considerando N_M saídas, temos, para o i-ésimo elemento:

$$\delta_i^M = \frac{\partial \sum_{k=1}^{N_M} e_k^2}{\partial u_i^M} = \frac{\partial \sum_{k=1}^{N_M} \left(d_k - y_k^M\right)^2}{\partial u_i^M} = -2\left(d_i - y_i^M\right)\frac{\partial y_i}{u_i^M} = -2\left(d_i - y_i^M\right)\dot{f}^M\left(u_i^M\right)$$

- Matricialmente,

$$\boldsymbol{\delta}^M = -2\dot{\mathbf{F}}^M(\mathbf{u}^M)(\mathbf{d} - \mathbf{y})$$

sendo a matriz $\dot{\mathbf{F}}^M(\mathbf{u}^M)$ definida como

$$\dot{\mathbf{F}}^M(\mathbf{u}^M) = \begin{bmatrix} \dot{f}^M(u_1^M) & 0 & \cdots & 0 \\ 0 & \dot{f}^M(u_2^M) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^M(u_{N_M}^M) \end{bmatrix}$$

- A aplicação sucessiva da regra da cadeia leva a uma recursão que, em termos matriciais / vetoriais, é simples:

$$\boldsymbol{\delta}^m = \dot{\mathbf{F}}^m(\mathbf{u}^m)(\mathbf{W}^{m+1})^T\boldsymbol{\delta}^{m+1}$$

3.2.1. Exemplo:

- Considere uma rede com uma camada intermediária e apenas um neurônio na camada de saída, como a mostrada na Fig. 12. As entradas de bias foram omitidas por uma questão de simplicidade.
- Temos $M = 2$. Devemos, portanto, começar calculando $\boldsymbol{\delta}^2$. Perceba que essa sensibilidade é um escalar porque há apenas um neurônio na camada de saída. Consideremos um único dado com entrada $\mathbf{x} = (x_1, x_2)$ e saída desejada d. Primeiramente, temos de supor que a rede terá certa configuração de pesos, de modo que, quando a entrada $\mathbf{x}$ for imposta, será possível calcular todos os sinais pertinentes ao longo da rede (até a saída). Essa é a *etapa direta* (*forward*).

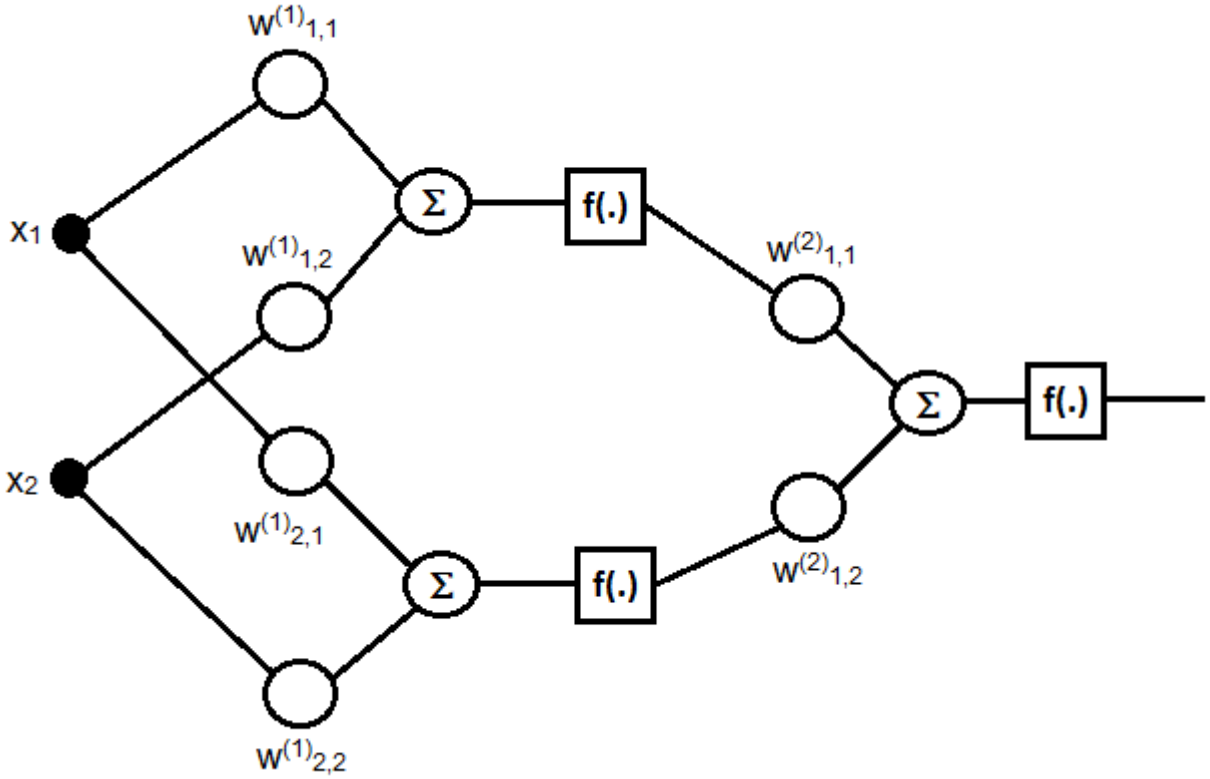

Fig. 12 – Exemplo de MLP.

- Teremos então a saída ${y_1}^{2}$ [§]. O erro pode ser então calculado:

§ Note que o $(\cdot)^2$ aqui não significa "ao quadrado", mas sim a indicação de que se trata de uma saída da camada $M = 2$.

$$e = d - {y_1}^2$$

- De posse do erro, podemos calcular o delta do neurônio solitário da camada de saída:

$$\delta^2 = -2(d - y_1^2)\dot{f}(u_1^2)$$

- Temos, portanto, nossa primeira sensibilidade. Agora, usaremos a recursão para retropropagar o erro até a camada anterior. A fórmula nos diz:

$$\boldsymbol{\delta}^1 = \dot{\mathbf{F}}^1(\mathbf{u}^1)(\mathbf{W}^2)^T\delta^2$$

- Acabamos de calcular δ^2. Vale $(\mathbf{W}^2)^T = \begin{bmatrix} {w^{(2)}}_{1,1} & {w^{(2)}}_{1,2} \end{bmatrix}^T$ e

$$\dot{\mathbf{F}}^1(\boldsymbol{u}^1) = \begin{bmatrix} \dot{f}(u_1^1) & 0 \\ 0 & \dot{f}(u_2^1) \end{bmatrix}$$

- Portanto,

$$\boldsymbol{\delta}^1 = \begin{bmatrix} w_{1,1}^{(2)}\dot{f}(u_1^1) \\ w_{1,2}^{(2)}\dot{f}(u_2^1) \end{bmatrix}\delta^2$$

- Antes de seguirmos adiante, tentemos chegar ao mesmo resultado aplicando a regra da cadeia "nua e crua". Nossa função custo é o EQM de uma amostra só:

$$J = e^2 = (d - {y_1}^2)^2$$

- Por favor, atente mais uma vez para o fato de que o "2" em ${y_1}^2$ não é um quadrado, mas um número de camada. Para encontrarmos δ^2, precisamos achar a derivada do custo com respeito a u_1^2. Apliquemos a regra da cadeia.

$$\frac{\partial J}{\partial u_1^2} = \frac{dJ}{de}\frac{\partial e}{\partial y_1^2}\frac{\partial y_1^2}{\partial u_1^2} = (2e)(-1)\dot{f}(u_1^2)$$

- Agora, encontremos a sensibilidade de um dos neurônios da camada intermediária (por exemplo, o neurônio 1). Podemos partir da derivada que acabamos de obter e ir adiante.

$$\delta_1^1 = \frac{\partial J}{\partial u_1^1} = \frac{\partial J}{\partial u_1^2}\frac{\partial u_1^2}{\partial y_1^1}\frac{\partial y_1^1}{\partial u_1^1} = \delta^2 w_{1,1}^2 \dot{f}(u_1^1)$$

- Antes de seguirmos adiante, peço ao leitor que *volte ao ponto assinalado como pedido*. Observe que, para calcular o gradiente, não basta calcular os δ's: é preciso multiplicá-los pelas entradas correspondentes (observando que os bias estão simbolicamente ligados a entradas fixas em +1).

4. Algoritmos de Aprendizado – Visões Práticas

- Não seria exagero dizer que os elementos básicos do aprendizado através de redes neurais foram apresentados. Há, sem embargo, uma série de aspectos práticos que é importante comentar de modo a deixar o leitor mais familiarizado com a prática atual da comunidade.
- Começaremos falando da questão da estimação do vetor gradiente.

4.1. Estimação: *Online, Batch, Minibatch*

- Conforme temos visto neste tópico, a base para o aprendizado em redes MLP é a obtenção do vetor gradiente e o estabelecimento de um processo iterativo de busca. Vimos que a obtenção do gradiente pode se dar num processo de retropropagação em que há uma parte direta (*forward*) de apresentação de um dado e obtenção da resposta da rede e uma etapa de retropropagação em que se calculam as derivadas necessárias.
- Vimos também que se calcula o gradiente associado a cada padrão e que a combinação de todos esses gradientes locais leva ao gradiente estimado para o conjunto de dados inteiro. No entanto, surge aqui um questionamento interessante: o que é melhor, usar o gradiente local e já dar um passo de otimização ou reunir o "gradiente completo" e então dar um passo único e mais preciso?

- Nesse questionamento, desenham-se dois polos: uma estimação *online* do gradiente (padrão-a-padrão) e uma estimação *batch* (em batelada). Vejamos primeiramente a noção geral de adaptação com estimação *online*, como expressa no seguinte algoritmo, um método clássico de primeira ordem.

- Defina uma condição inicial para o vetor de pesos $\mathbf{w}$ e um passo α pequeno;
- Faça $k = 0, t = 0$ e calcule $J(\mathbf{w}(k))$;
- Enquanto o critério de parada não for atendido, faça:
    - Ordene aleatoriamente os padrões de entrada-saída;
    - Para l variando de 1 até N, faça:
        - Apresente o padrão l de entrada à rede;
        - Calcule $J_l(\mathbf{w}(t))$ e $\nabla J_l(\mathbf{w}(t))$;
        - $\mathbf{w}(\mathrm{t}+1) = \mathbf{w}(\mathrm{t}) - \alpha \nabla J_l(\mathbf{w}(t))$; $t = t + 1$;
    - $k = k + 1$;
    - Calcule $J(\mathbf{w}(k))$;

Algoritmo 1 – Adaptação Online.

- O outro extremo seria, como dito, utilizar todo o conjunto de dados para estimar o gradiente antes de dar o passo do processo iterativo. Essa é a ideia de *batch*. O algoritmo 2 ilustra o *modus operandi* correspondente (novamente considerando uma metodologia de primeira ordem).
- Nas modernas redes profundas, usadas amiúde em problemas com enormes conjuntos de dados, a regra é adotar o caminho do meio dos *minibatches*. Nesse caso, a adaptação é realizada com um gradiente estimado a partir de um meio-termo entre uma amostra e o número total de amostras (via de regra um valor relativamente pequeno em métodos de primeira ordem). As amostras que devem compor o *minibatch* são aleatoriamente tomadas do conjunto de dados. O algoritmo 3 ilustra isso.

- Defina uma condição inicial para o vetor de pesos $\mathbf{w}$ e um passo α pequeno;
- Faça $k = 0$ e calcule $J(\mathbf{w}(k))$;
- Enquanto o critério de parada não for atendido, faça:
    - ♦ Para l variando de 1 até N, faça:
        - ▪ Apresente o padrão l de entrada à rede;
        - ▪ Calcule $J_l(\mathbf{w}(t))$ e $\nabla J_l(\mathbf{w}(t))$;
    - ♦ $\mathbf{w}(k+1) = \mathbf{w}(k) - \frac{\alpha}{N}\sum_{l=1}^{N} \nabla J_l(\mathbf{w}(k))$;
    - ♦ $k = k + 1$;
    - ♦ Calcule $J(\mathbf{w}(k))$;

Algoritmo 2 – Adaptação em Esquema *Batch*.

- Defina uma condição inicial para o vetor de pesos $\mathbf{w}$ e um passo α pequeno;
- Faça $k = 0$ e calcule $J(\mathbf{w}(k))$;
- Enquanto o critério de parada não for atendido, faça:
    - ♦ Para l variando de 1 até m, faça:
        - ▪ Apresente o padrão l de entrada, amostrado para compor um *minibatch*, à rede;
        - ▪ Calcule $J_l(\mathbf{w}(t))$ e $\nabla J_l(\mathbf{w}(t))$;
    - ♦ $\mathbf{w}(k+1) = \mathbf{w}(k) - \frac{\alpha}{m}\sum_{l=1}^{m} \nabla J_l(\mathbf{w}(k))$;
    - ♦ $k = k + 1$;
    - ♦ Calcule $J(\mathbf{w}(k))$;

Algoritmo 3 – Adaptação Baseada em *Minibatch*.

4.2. Busca: Tema e Variações

Há vários algoritmos baseados no gradiente que podem ser empregados para otimizar os parâmetros de uma rede neural. Ater-nos-emos aqui a alguns métodos usuais na literatura moderna, muito centrada em aprendizado profundo.

4.2.1. Método do Gradiente Estocástico (*Stochastic Gradient Descent, SGD*)

- Na seção 4.1, vimos que o método online utiliza uma única amostra para estimar o gradiente da função custo. Este tipo de estimador é o que gera a noção de *gradiente estocástico*. Caso utilizemos *minibatches*, também teremos uma estimativa do gradiente, o qual, a rigor, seria determinístico apenas se usássemos todos os dados (no caso *batch*). Por esse motivo, métodos de primeira ordem como os vistos são conhecidos como métodos de *stochastic gradient descent* (SGD).

- A tarefa de escolha do passo é complicada e levanta o conhecido compromisso entre velocidade de convergência e estabilidade / precisão. Pode-se usar um valor fixo, mas é considerado usual (GOODFELLOW ET AL., 2016) que se adote um método de variação linear decrescente de um valor α_0 a um valor α_τ (da iteração 0 à iteração τ):

$$\alpha_k = \left(1 - \frac{k}{\tau}\right)\alpha_0 + \frac{k}{\tau}\alpha_\tau$$

- Após a τ-ésima iteração, pode-se deixar o valor de passo fixo. Naturalmente, a definição dos valores necessários é (mais) um problema "a ser tratado caso-a-caso".

4.2.2. Momento

- O uso de um termo de momento numa metodologia de gradiente pode ser interessante por trazer, para o ajuste de pesos em determinada iteração,

informação de gradientes anteriores acumulados. Isso, em certas situações, melhora a característica de convergência.

- Não nos preocuparemos com a relação de "momento" em otimização com "momento" na física de Newton (daí o nome) – passaremos, sem mais delongas, à exposição do algoritmo resultante. Faremos isso no contexto de uma metodologia SGD com *minibatch* (GOODFELLOW ET AL., 2016).
- Partamos de um esquema de aprendizado em *minibatch*, como o algoritmo 3. Seja $\mathbf{g}$ o gradiente estimado para o minibatch e $\mathbf{v}$ um termo de velocidade. A velocidade é atualizada:

$$\mathbf{v} \leftarrow \varphi\mathbf{v} - \alpha\mathbf{g}$$

  A atualização dos pesos é, então:

$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$$

- O efeito do termo de momento pode ser visto como algo que se acumula de acordo com a regra de uma progressão geométrica. Portanto, podemos pensar

em seu efeito de aceleração no sentido contrário do gradiente à luz do termo $\frac{1}{1-\varphi}$. Valores típicos de φ são 0,5, 0,9 e 0,99. Também é possível planejar uma progressão de φ com o número de iterações (GOODFELLOW ET AL., 2016).

4.2.3. Momento de Nesterov

- O *método do momento de Nesterov* (GOODFELLOW ET AL., 2016) pode ser visto, essencialmente, como uma variação do método exposto na seção anterior em que a estimativa do vetor gradiente não é feita sobre o vetor de parâmetros **w**, mas sim sobre $\mathbf{w} + \varphi\mathbf{v}$. Esse termo adicional funciona como um fator de correção que pode beneficiar, em alguns casos, a velocidade de convergência.

4.2.4. Modelos com Passo de Adaptação (Learning Rate) Adaptativo

- Como exposto anteriormente, o passo de adaptação é um hiperparâmetro difícil de se ajustar bem e bastante relevante para o sucesso do treinamento de uma rede. Isso motivou o surgimento de um conjunto de métodos com mecanismos capazes de modificá-lo dinamicamente. Dentre as técnicas mais populares dessa classe estão o AdaGrad, o RMSProp e o Adam (de "*adaptive moments*") (GOODFELLOW ET AL., 2016).

4.3. Algumas Palavras sobre Inicialização dos Pesos

- Uma vez que os métodos de treinamento de redes neurais MLP são iterativos, eles dependem de uma inicialização. Como os métodos são de busca local, a inicialização pode afetar drasticamente a qualidade da solução obtida. Também

pode haver variações expressivas do ponto de vista da velocidade de convergência.

- Um ponto importante da inicialização é "quebrar a simetria" entre as unidades (GOODFELLOW ET AL., 2016), já que os neurônios de uma camada são equivalentes do prisma da arquitetura. Isso, naturalmente, sugere uma *abordagem aleatória*.
- Os pesos tipicamente são obtidos de distribuições gaussianas ou uniformes. A ordem de grandeza desses pesos levanta diversos aspectos (GOODFELLOW ET AL., 2016):
    a) Pesos de maior magnitude criam maior distinção entre unidades (quebra de simetria); por outro lado, pode causar problemas de instabilidade.
    b) Pesos de maior magnitude favorecem a propagação de informação; por outro lado, causam preocupações do ponto de regularização.

c) Pesos de magnitude elevada podem levar os neurônios (no caso de sigmoides como a tangente hiperbólica e a função logística) a operarem numa região de saturação, comprometendo a convergência.

- Algumas heurísticas podem ser mencionadas (GOODFELLOW ET AL., 2016). Uma primeira seria, para uma camada com m entradas e n saídas, amostrar os pesos da seguinte forma:

$$w_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

sendo $U(\cdot)$ uma distribuição uniforme. Outra heurística seria:

$$w_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

- Sobre a inicialização dos termos de *bias*, aponta-se, em (GOODFELLOW ET AL., 2016), que valores nulos podem ser escolhidos na maioria dos casos.

5. Referências bibliográficas

CYBENKO, G., "Approximation by Superpositions of a Sigmoidal Function", Mathematics of Control, Signals and Systems, Vol. 2, No. 4, pp. 303 – 314, 1989.

GOODFELLOW, I., BENGIO, Y., COURVILLE, A., *Deep Learning*, MIT Press, 2016.

HAYKIN, S. **Neural Networks and Learning Machines**, 3rd edition, Prentice-Hall, 2008.

MAAS, A. L., HANNUN, A. Y., NG, A. Y. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". Proceedings of the 30th International Conference on Machine Learning (ICML), Atlanta, Georgia, USA, 2013.

VON ZUBEN, F. J., **Notas de Aulas do Curso "Redes Neurais" (IA353),** disponíveis em *http://www.dca.fee.unicamp.br/~vonzuben/courses/ia353.html*, 2007.