CC BY

"Life can only be understood backwards; but it must be lived forwards" (Søren Kierkegaard).

Retropropagação

Paulo Ricardo Lisboa de Almeida

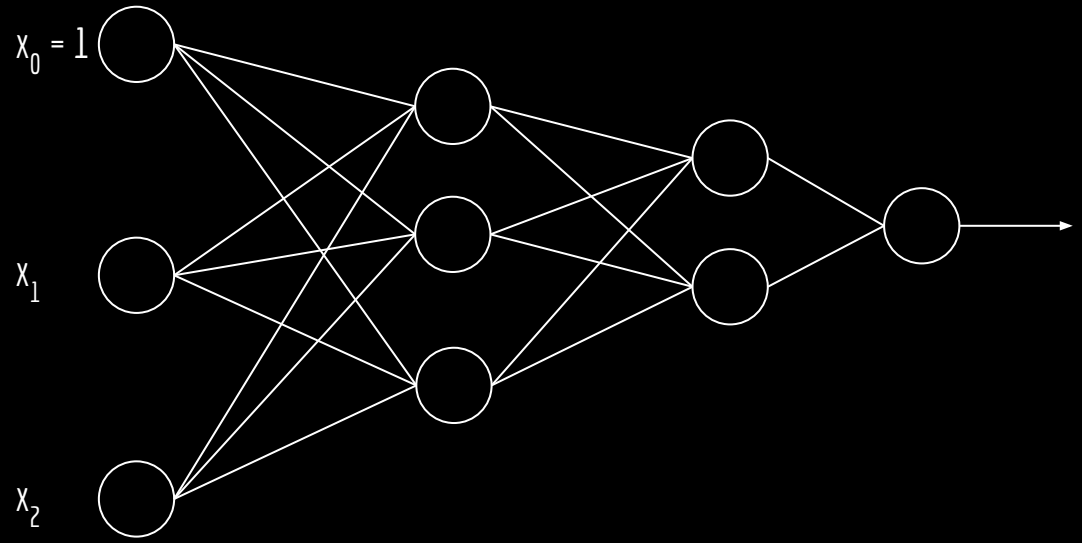
DSBD

UFPR

Minimizando o erro

Podemos interpretar um MLP como uma função de múltiplas variáveis.

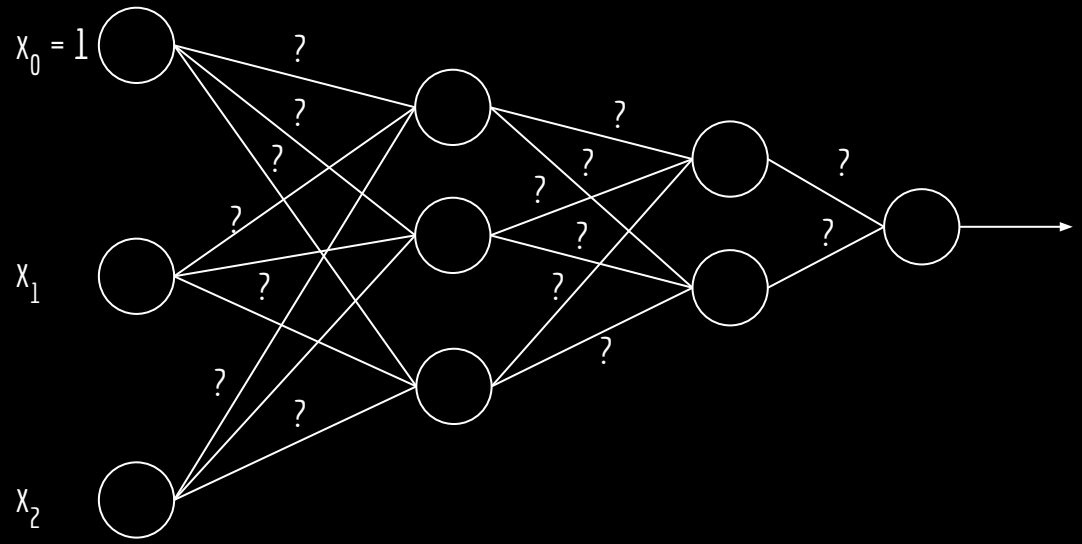
Recebe os dados de entrada (vetores), e gera uma (ou múltiplas) saída desejada.



Minimizando o erro

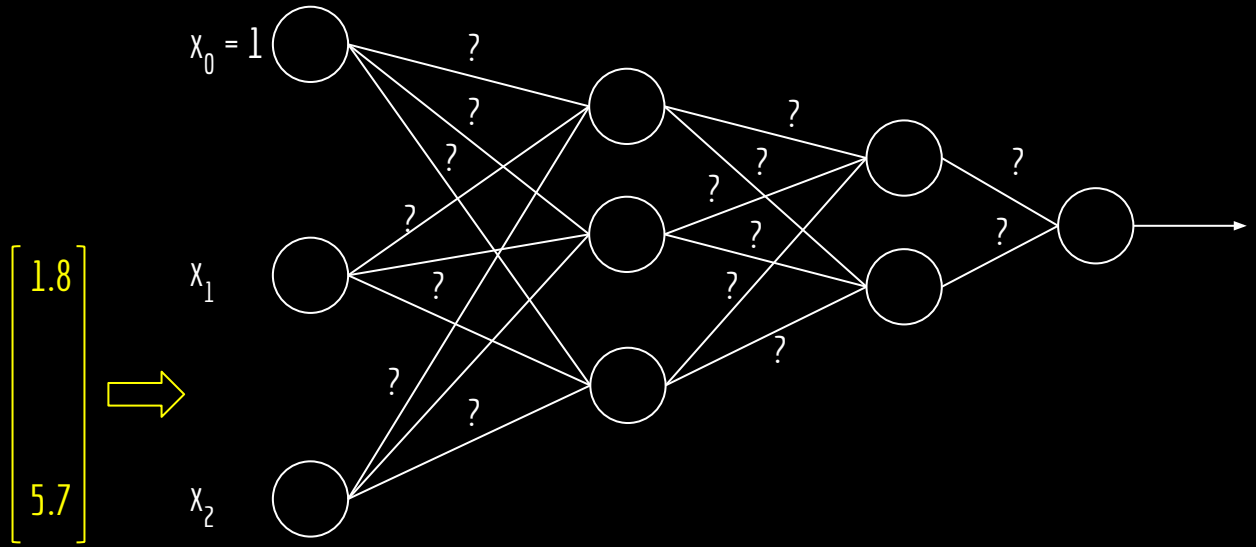
Os pesos do MLP definem a relação entre as entradas e saídas.

O desafio é usar os dados de treinamento para definir esses pesos.



Minimizando o erro

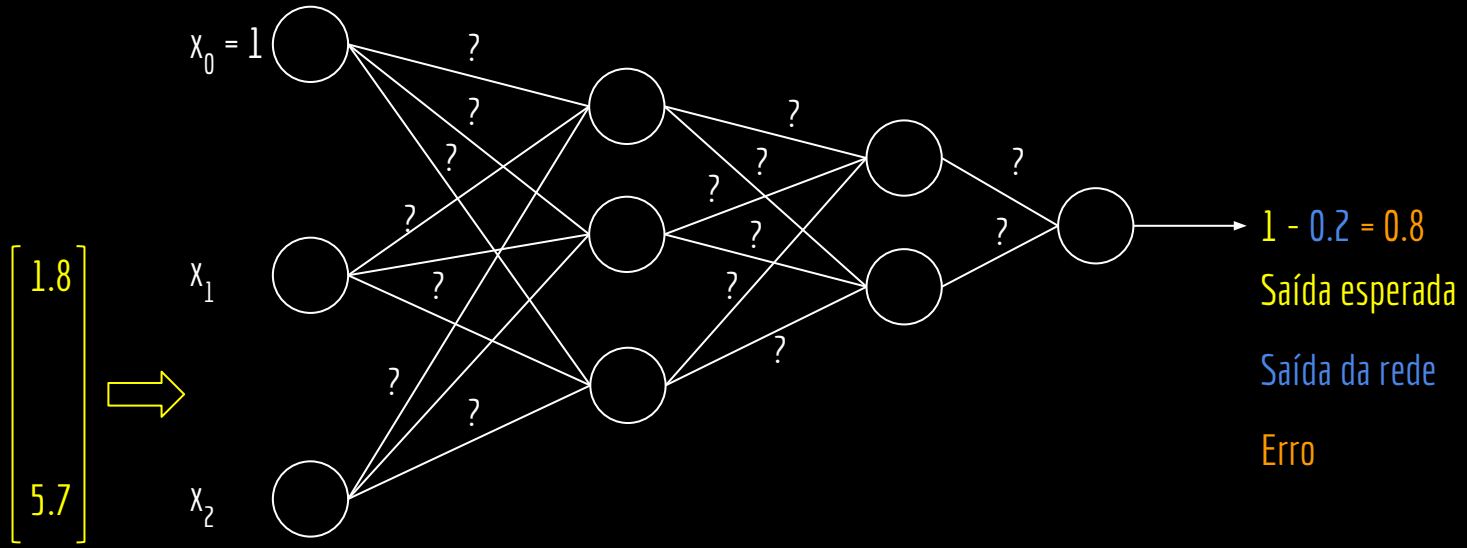
A ideia é inicializar os pesos aleatoriamente, e fazer um forward propagation de cada dado de treinamento.



Minimizando o erro

A ideia é inicializar os pesos aleatoriamente, e fazer um forward propagation de cada dado de treinamento.

Calcular a saída da rede e comparar com a saída esperada.

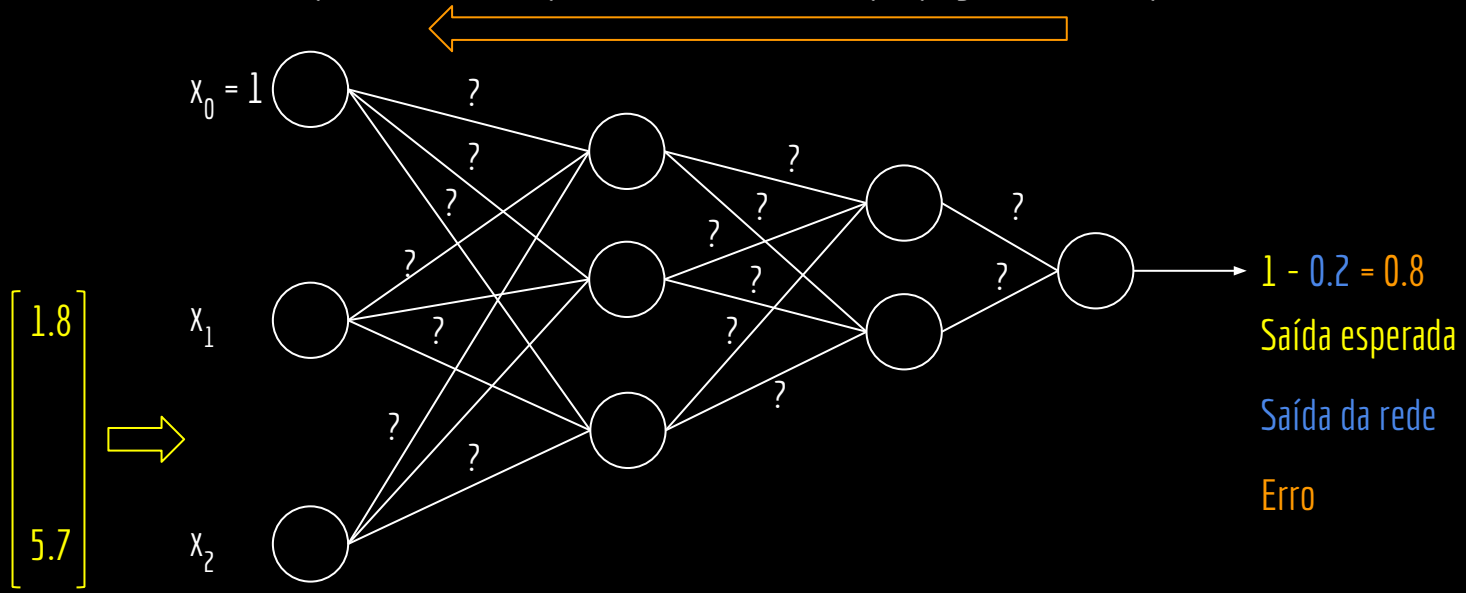


Minimizando o erro

A ideia é inicializar os pesos aleatoriamente, e fazer um forward propagation de cada dado de treinamento.

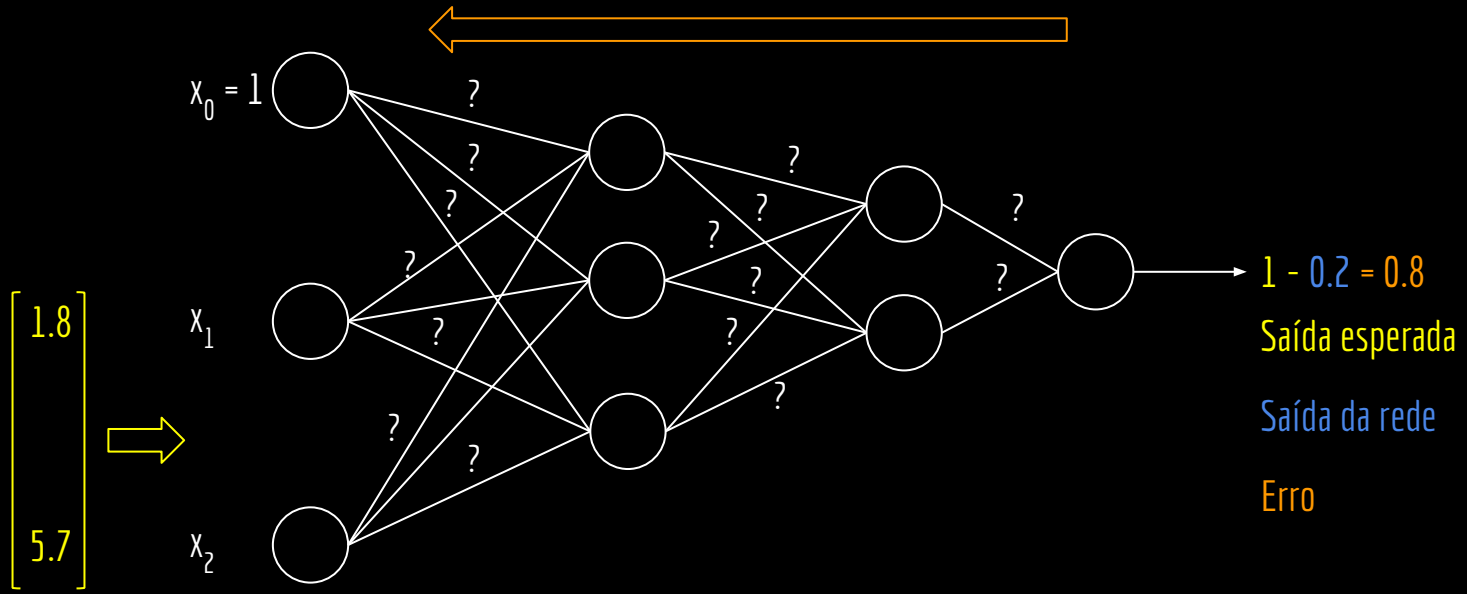
Calcular a saída da rede e comparar com a saída esperada.

Usar o erro para calibrar os pesos iterativamente, propagando o erro pela rede de **trás para frente**.



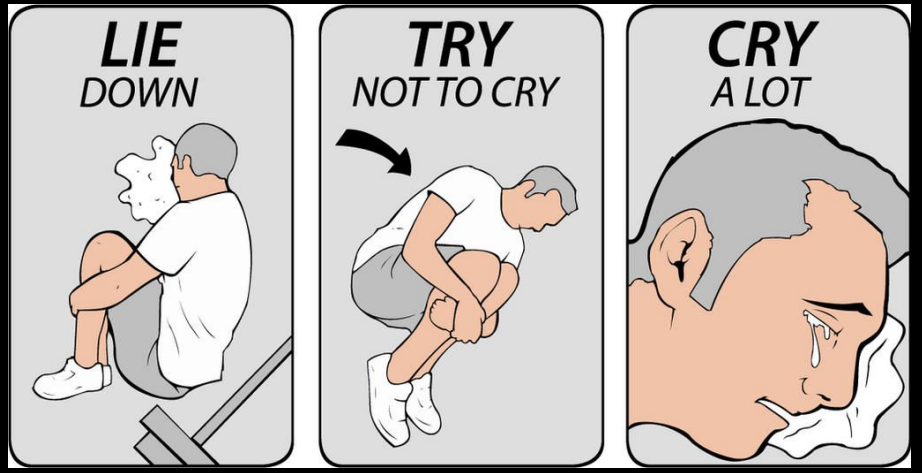
Backpropagation

Essa é a ideia do algoritmo de **Retropropagação** (**Backpropagation**).



Backpropagation

O algoritmo de retropropagação é baseado em conceitos de cálculo diferencial para cálculo de gradientes.

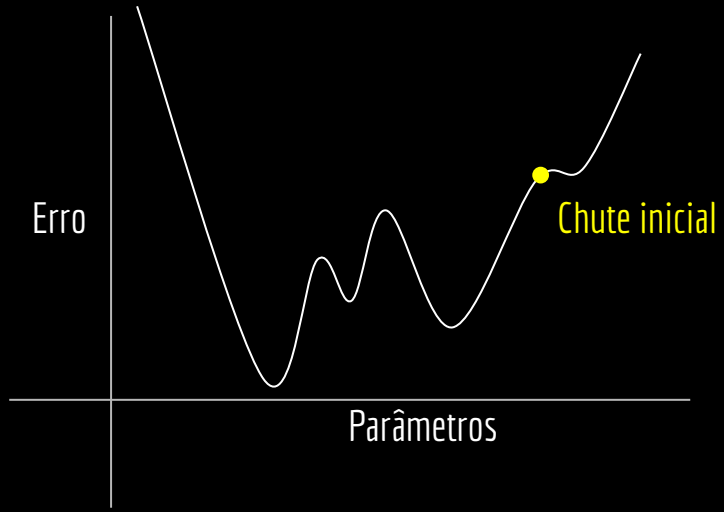


Superfície de otimização

Considere uma função que representa o erro.

O objetivo é encontrar o ponto que minimiza o erro nessa função.

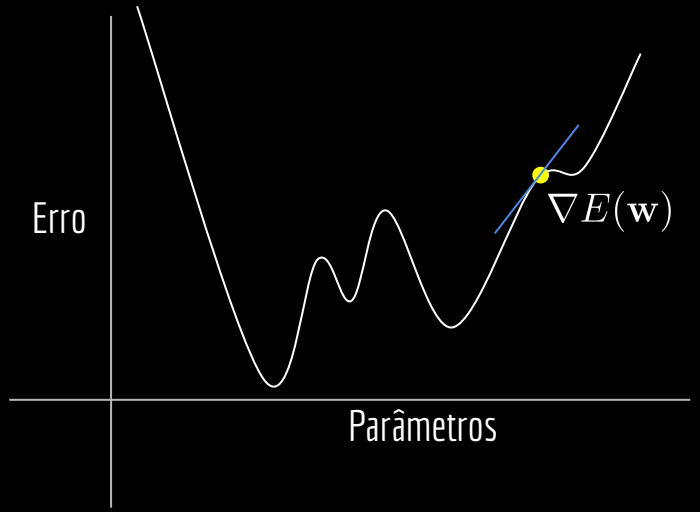
Como?



Superfície de otimização

Calcular a inclinação (derivada) da curva no ponto atual.

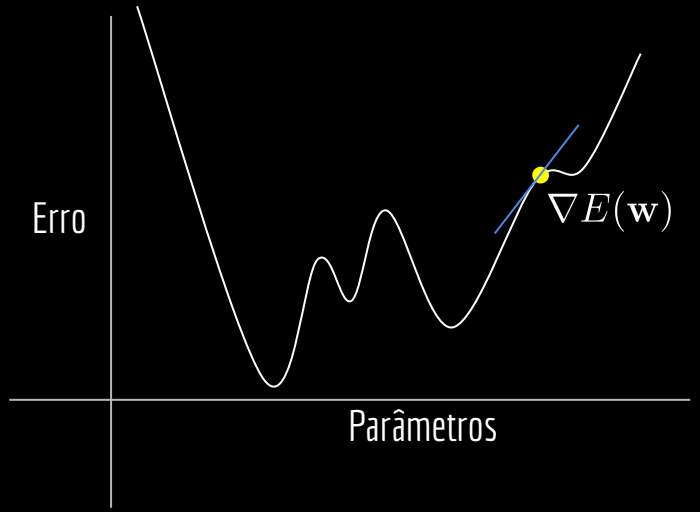
Utilizar o resultado para definir para onde mover o parâmetro.



Descida de Gradiente

O movimento será dado pelo ângulo da tangente, e pelo fator de aprendizagem η.

Descida de Gradiente.



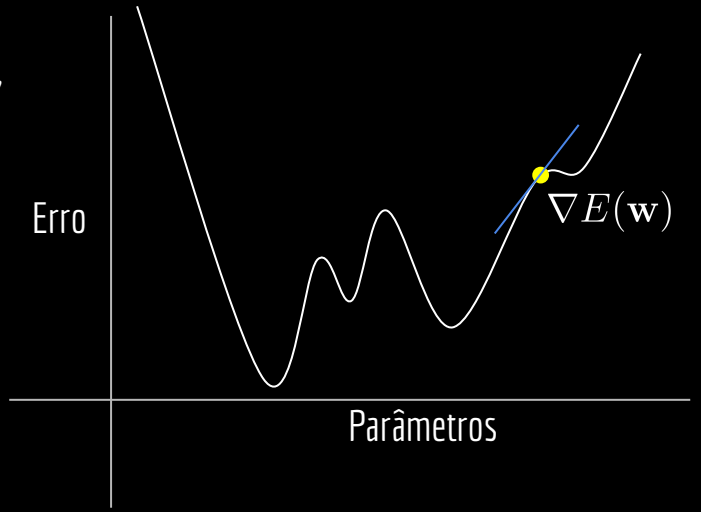
Perguntas

Como o fator de aprendizagem η influencia na minimização do erro?

É garantido que encontraremos um mínimo global?

É garantido que encontraremos um mínimo local?

Usando as ideias vistas até agora, ao treinar duas redes, ambas vão chegar no mesmo ponto de mínimo?



Mais sobre zeros de função

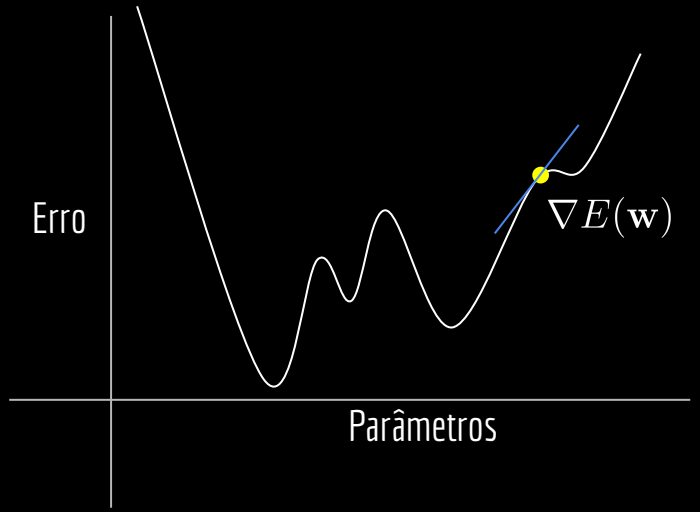
https://prlalmeida.com.br/ci202-2021-01/Aula5.pdf

https://prlalmeida.com.br/ci202-2021-01/Aula9.pdf

Gradiente

O gradiente $\nabla E(\mathbf{w})$ aponta para a direção que maximiza o erro.

Logo, multiplicamos por -1 para minimizar.



Função de ativação

Dado que vamos calcular o gradiente através de derivativas, não podemos usar a Função Degrau como função de ativação.

Por quê?

Função de ativação

Dado que vamos calcular o gradiente através de derivativas, não podemos usar a Função Degrau como função de ativação.

A função não é contínua.

Função de ativação

Veja uma lista de funções de ativação que podem ser usadas, por exemplo, com o Framework Pytorch.

https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Como exemplo, vamos usar a função Sigmóide (*Sigmoid*).

Sigmóide

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Faça você mesmo

Plote no matplotlib um gráfico para a função Sigmóide para valores entre -10 e +10.

Teste também outros intervalos.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

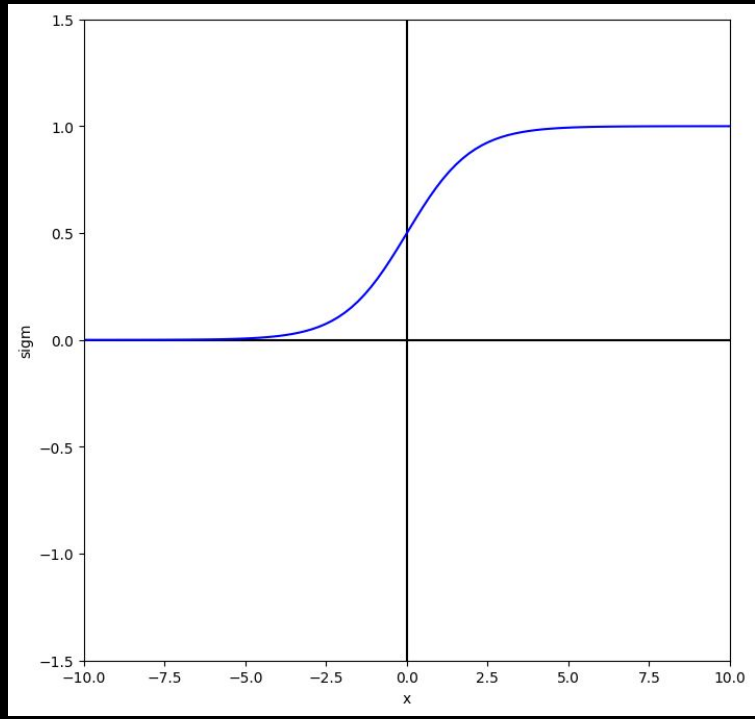
Faça você mesmo

Plote no matplotlib um gráfico para a função Sigmóide para valores entre -10 e +10.

Teste também outros intervalos.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

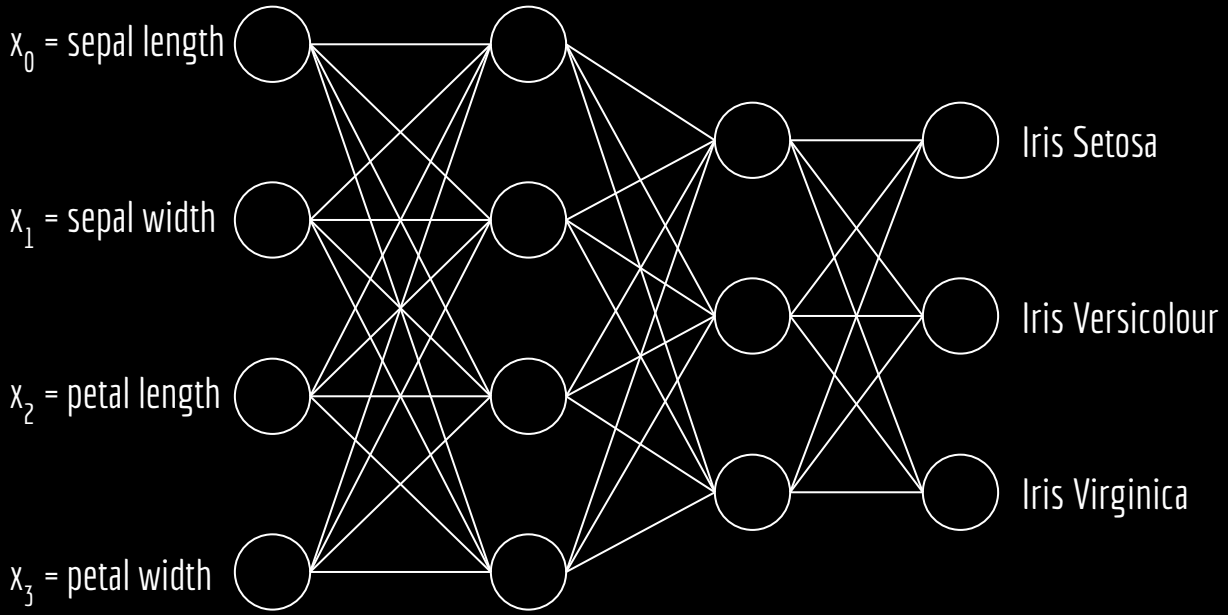
A função é contínua e tem um comportamento similar à Função Degrau de Heaviside.



Exemplo

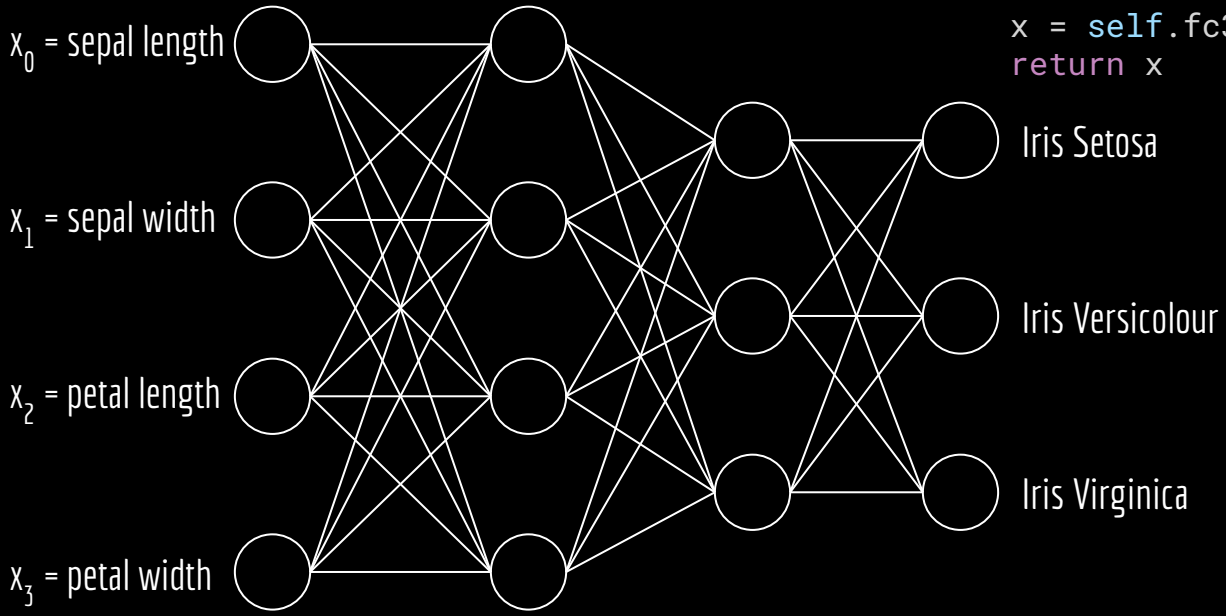
Rede para resolver o problema Iris Setosa, considerando todas 4 características e 3 classes de saída.

Obs.: para simplificar, vamos omitir o bias a partir de agora, **mas ele existe**.



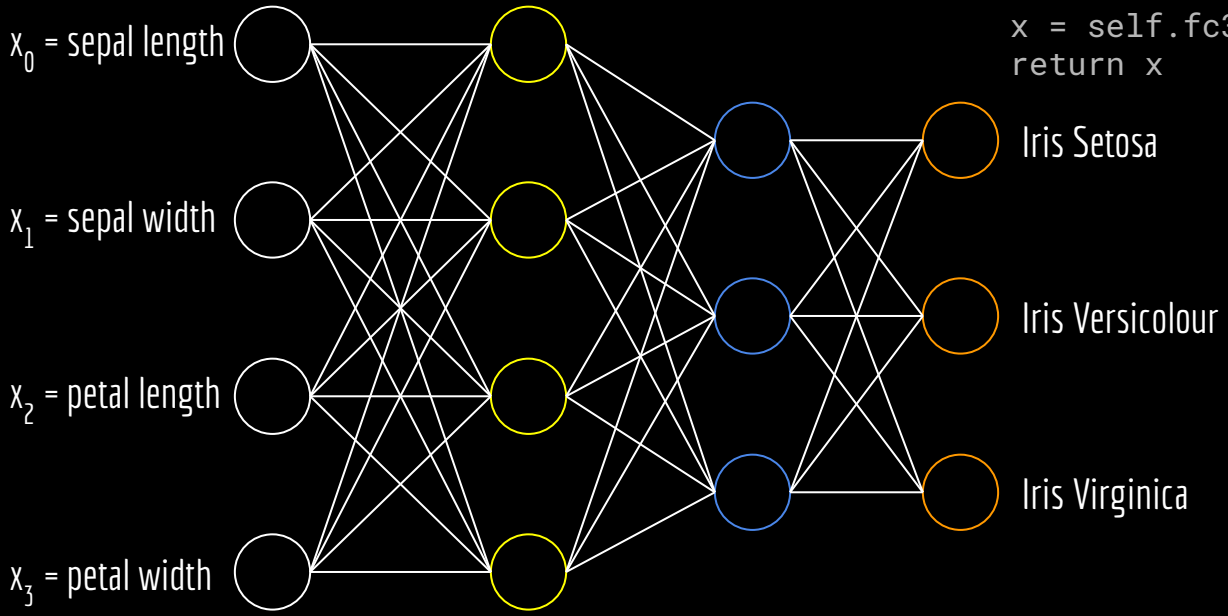
Pytorch

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(4, 4)
        self.fc2 = nn.Linear(4, 3)
        self.fc3 = nn.Linear(3, 3)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x
```



Pytorch

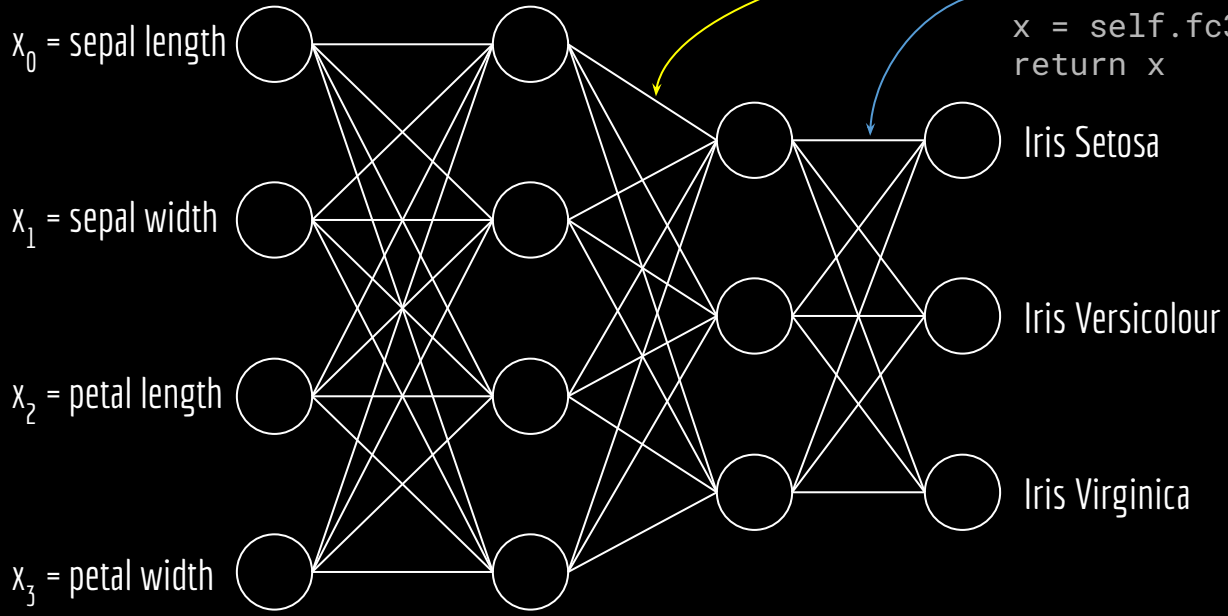
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(4, 4)
        self.fc2 = nn.Linear(4, 3)
        self.fc3 = nn.Linear(3, 3)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x
```



Pytorch

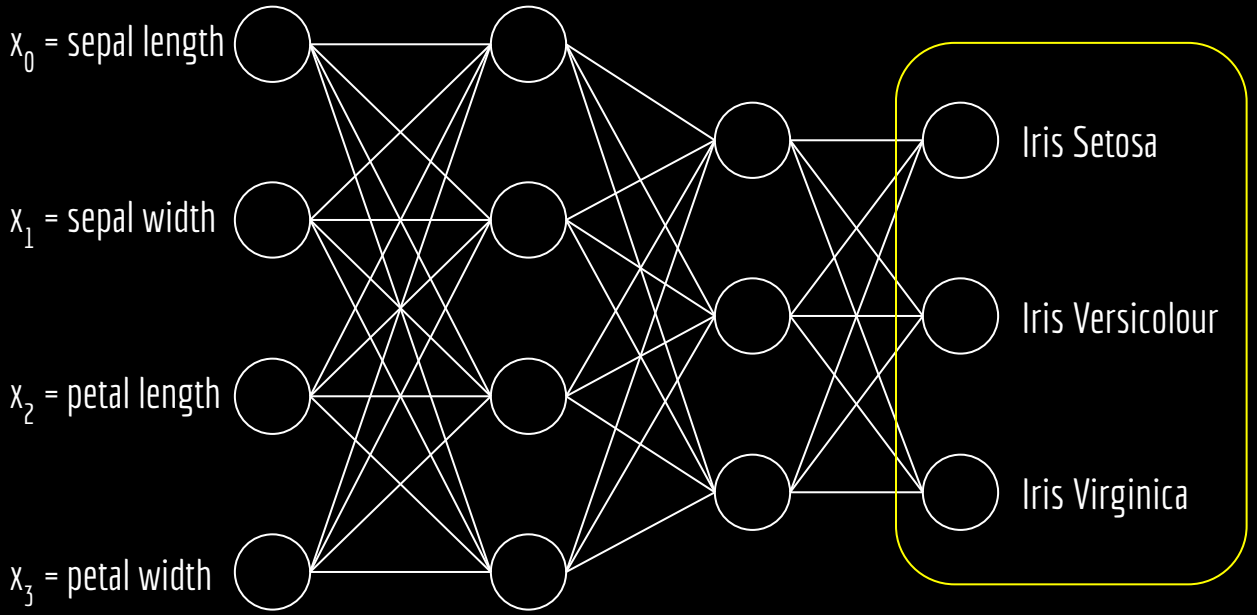
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(4, 4)
        self.fc2 = nn.Linear(4, 3)
        self.fc3 = nn.Linear(3, 3)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x
```

Sigmóides como funções de ativação.



Exemplo

Rede para resolver o problema Iris Setosa, considerando todas 4 características e 3 classes de saída.

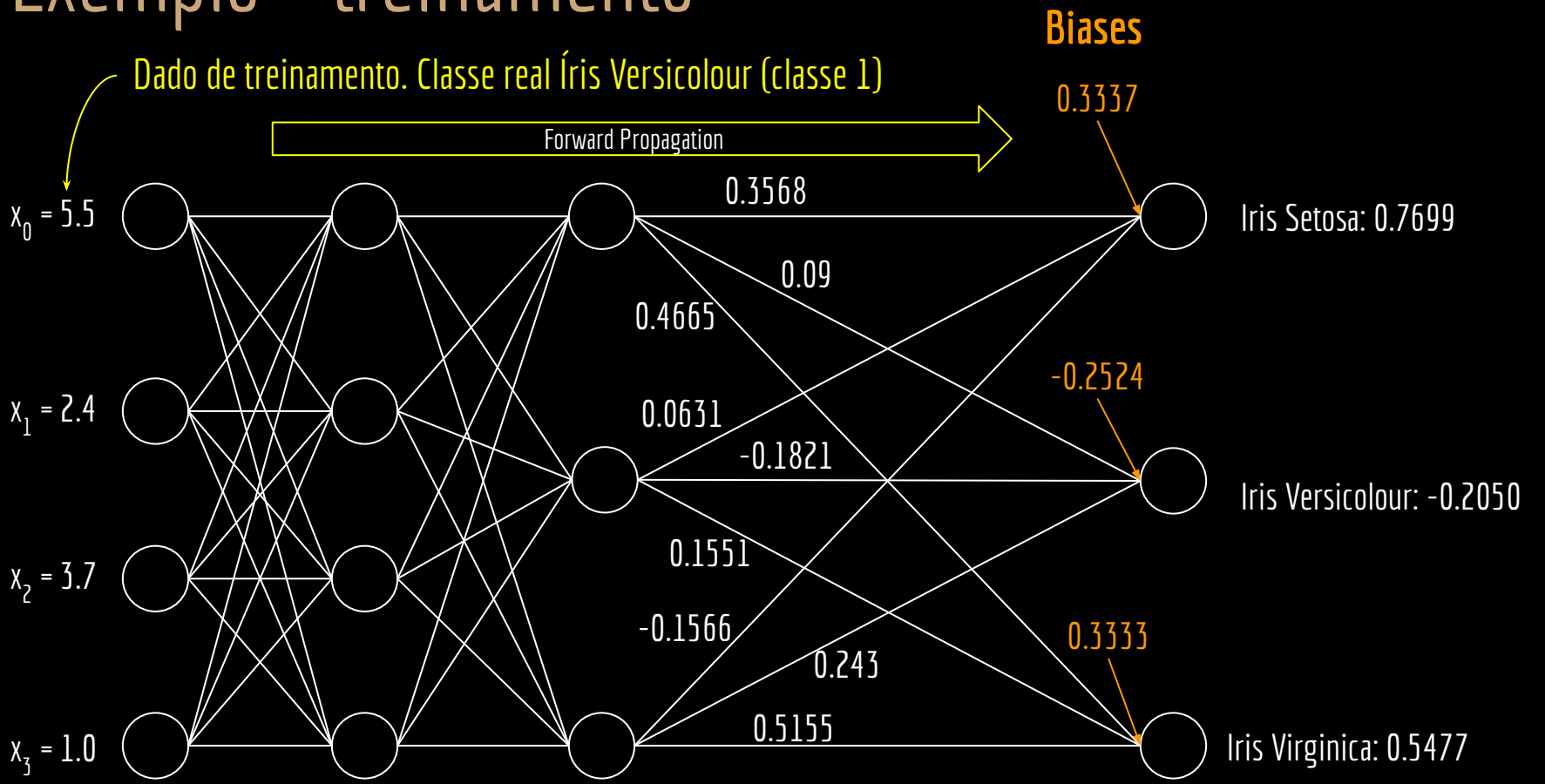


Problema multiclasse.

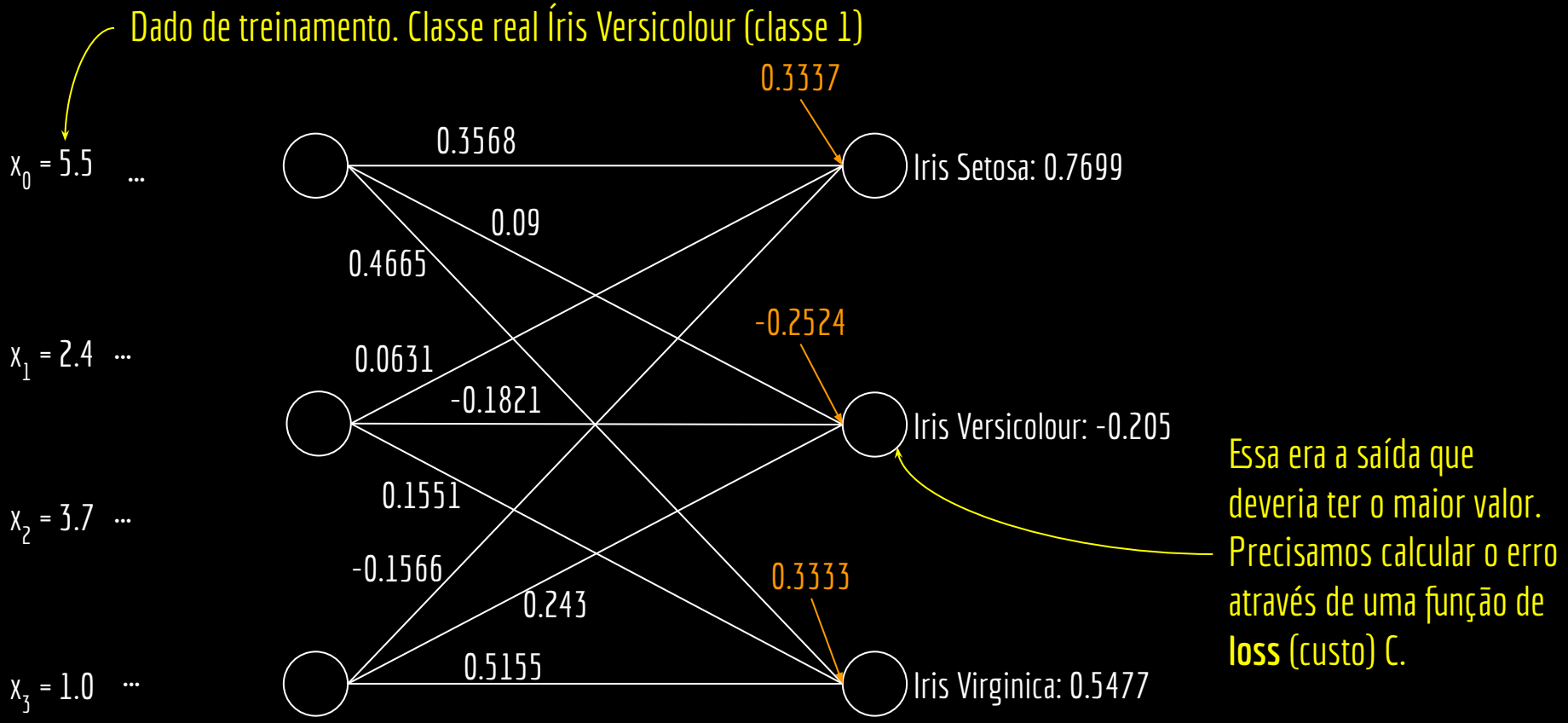
Um neurônio para cada classe na camada de saída.

A classe predita é a que possui maior valor.

Exemplo - treinamento



Exemplo - treinamento



Função de Loss

A função de Loss é responsável por calcular a “distância” entre a saída obtida e a esperada.

Ponto chave no design da rede.

Em um problema binário, por exemplo, poderíamos simplesmente calcular a diferença entre a saída obtida e a saída esperada.

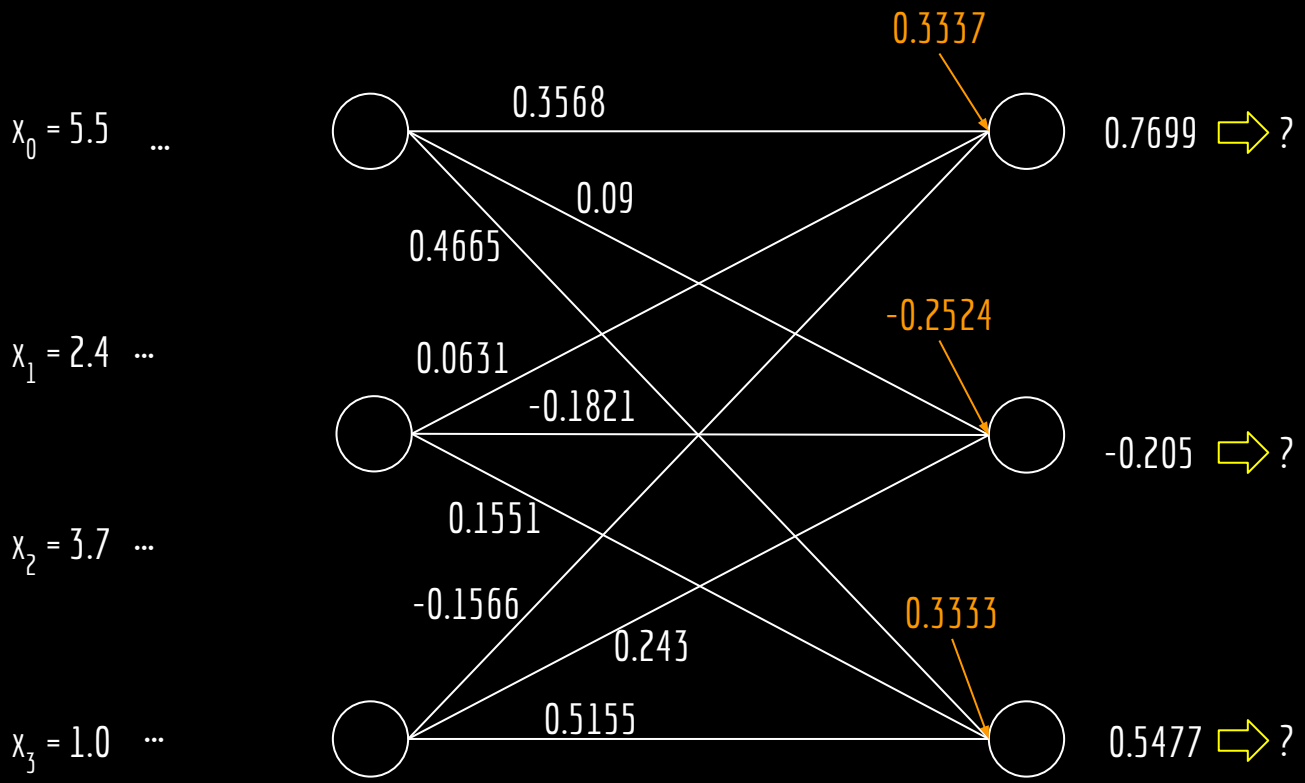
Talvez seja ineficiente, mas funcionaria.

Para o problema multiclasses, vamos precisar de algo mais softisticado.

Veja uma lista de funções prontas no Pytorch em https://pytorch.org/docs/stable/nn.html

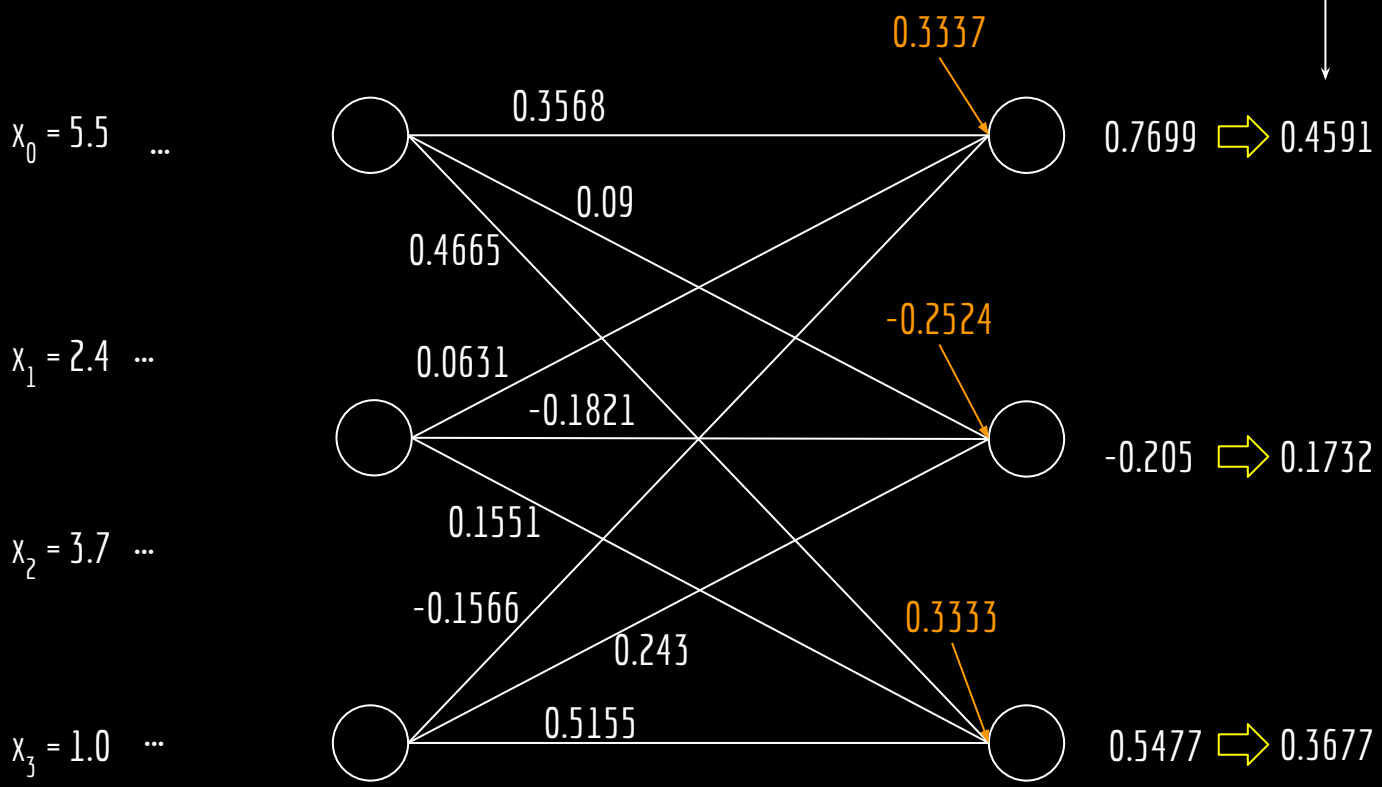
Exemplo - treinamento

$$softmax(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_j e^{\hat{y}_j}}$$



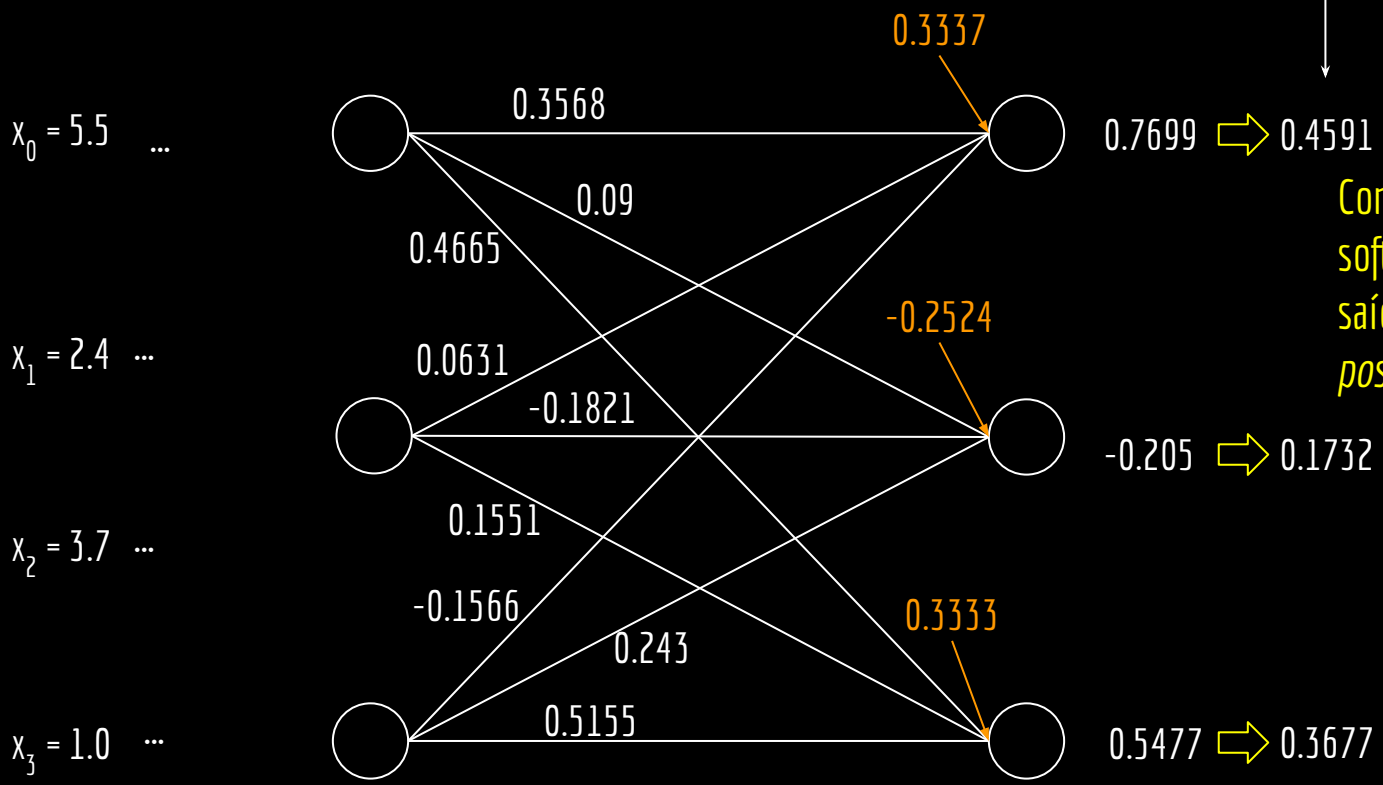
Exemplo - treinamento

$$softmax(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_j e^{\hat{y}_j}}$$



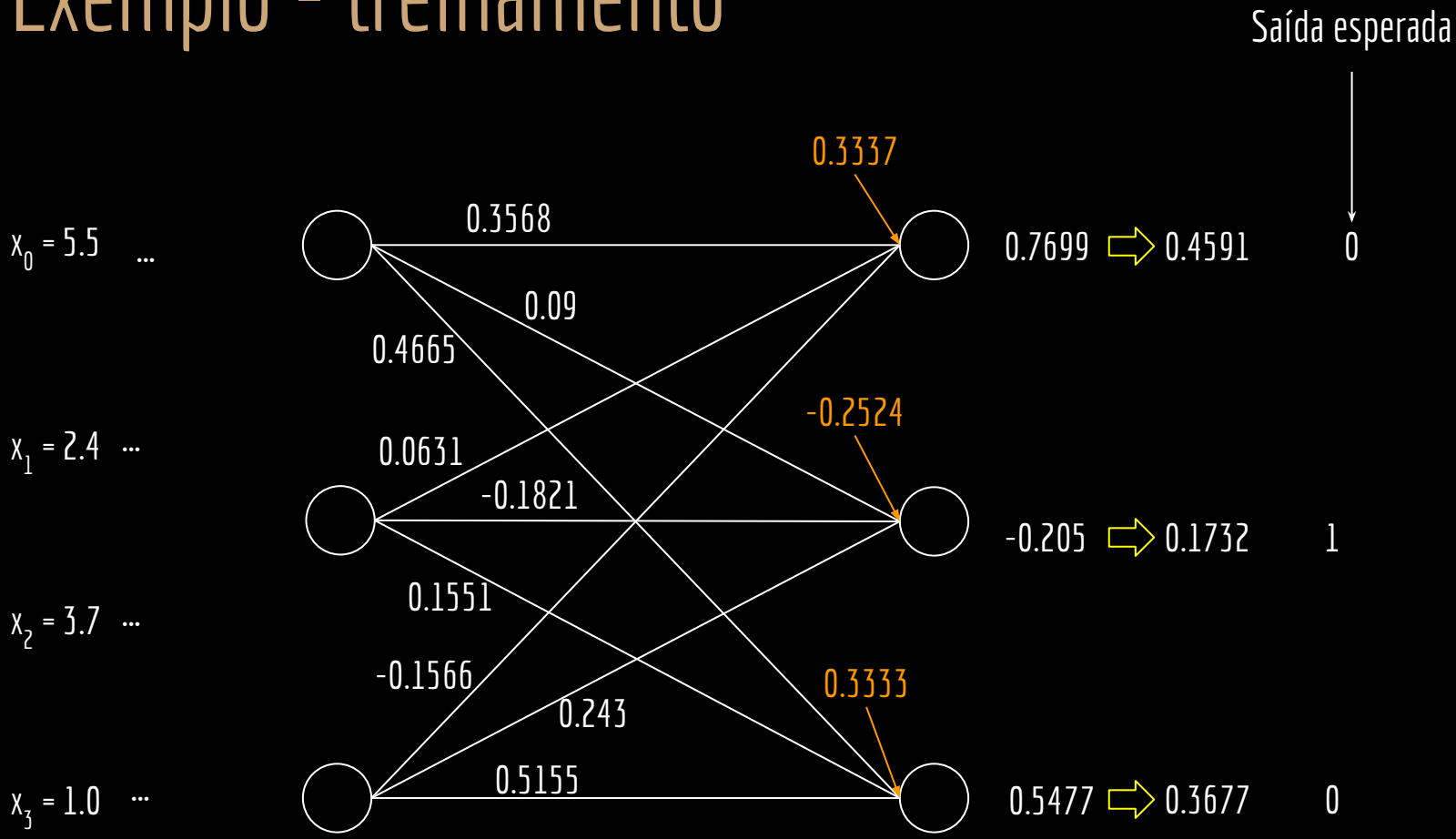
Exemplo - treinamento

$$softmax(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_j e^{\hat{y}_j}}$$



Com a aplicação da função softmax "normalizamos" a saída. Temos a probabilidade *a posteriori* das classes. Pesquise!

Exemplo - treinamento



Cross Entropy Loss

Como temos um problema de múltiplas classes, vamos usar uma função de *loss* que considera esse tipo de cenário.

A função vai nos dizer “o quão longe” estamos da resposta correta.

Uma opção comum é o Cross Entropy Loss:
https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html#torch.nn.CrossEntropyLoss

Cross Entropy Loss

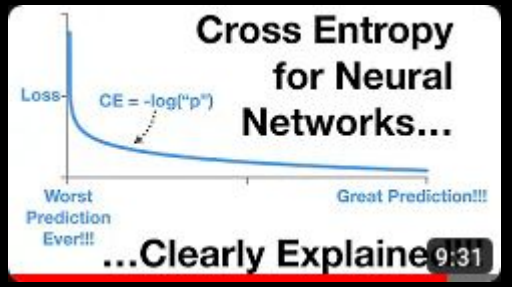
Dado que temos saídas com probabilidades a posteriori (através da função softmax), a equação é:

$$L = -ln(\hat{y}_r)$$

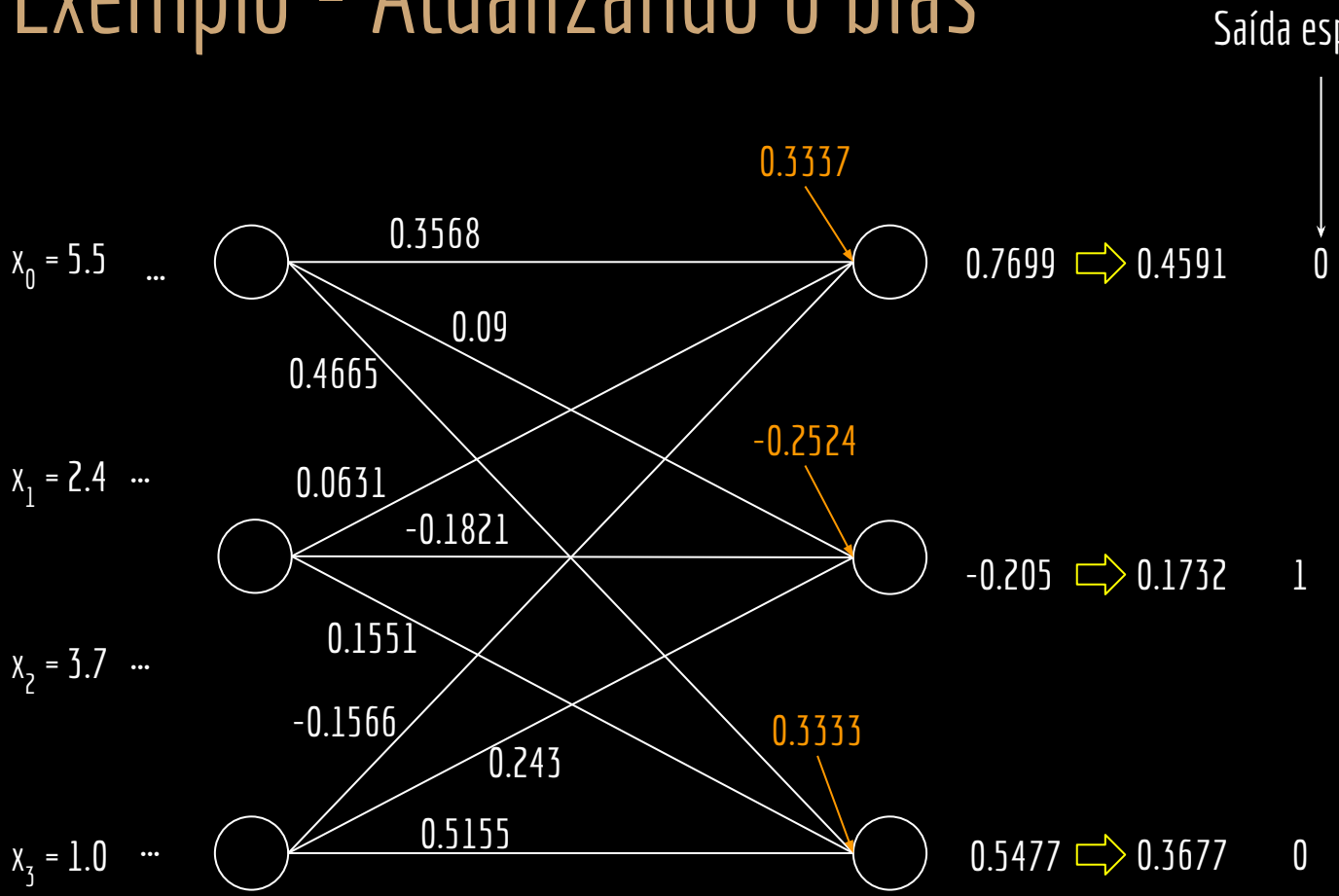
Onde r é a classe da instância de treinamento, e $\hat{y}_r$ é a probabilidade a posteriori (saída normalizada por softmax) do neurônio responsável por prever essa classe.

Assista

Assista a esse vídeo sobre o Cross Entropy Loss: https://www.youtube.com/watch?v=6ArSys5qHAU&t=510s

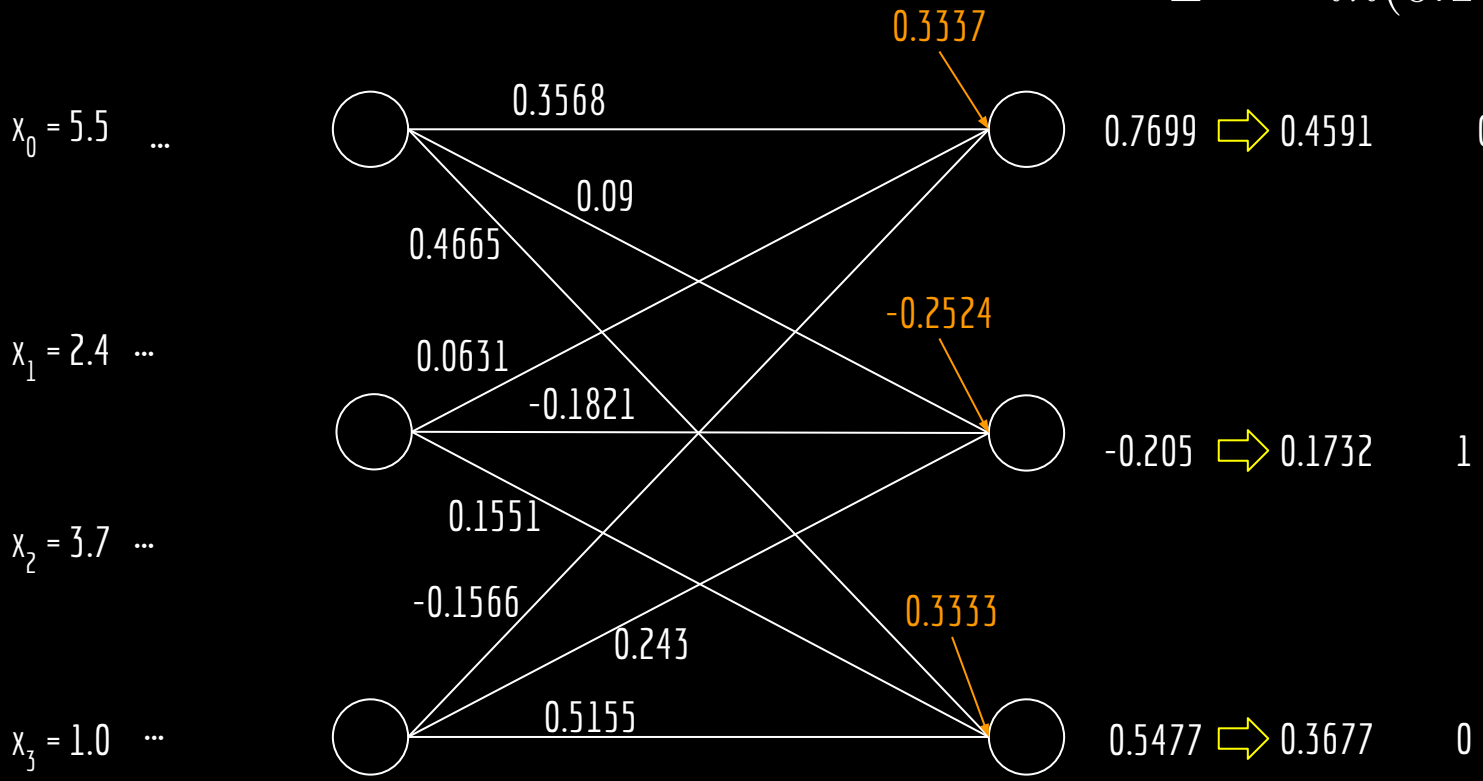


Exemplo - Atualizando o bias



Exemplo - Cross Entropy Loss

$$L = -ln(0.1732) = 1.7533$$

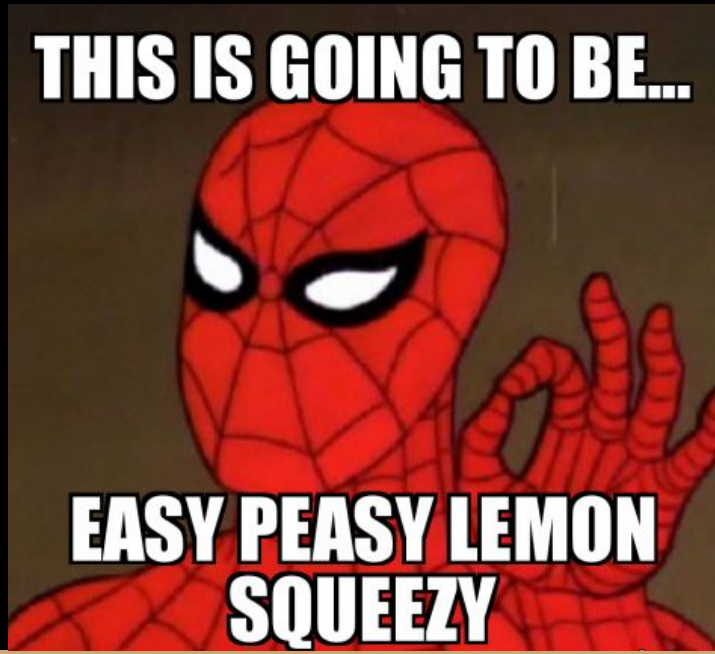


Treinamento

Baseado no *loss*, precisamos reajustar **todos** os pesos e biases.

Como exemplo, vamos ajustar apenas os biases da última camada.

- Camada mais simples de ajustar.
- Depende diretamente da saída.



Equações

O gradiente do erro em relação aos biases na última camada é dado pela derivada parcial:

$$\nabla E(b_i) = \frac{\partial L}{\partial b_i}$$

Bias do i-ésimo neurônio de saída.

Equações

O gradiente do erro em relação aos biases na última camada é dado pela derivada parcial:

$$\nabla E(b_i) = \frac{\partial L}{\partial b_i}$$

O Cross Entropy Loss que usamos não depende diretamente de b, mas sim das probabilidades *a posteriori*.

Usando a regra da cadeia:

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} . \frac{\partial p_j}{\partial output_i}$$

Equações

O gradiente do erro em relação aos biases na última camada é dado pela derivada parcial:

$$\nabla E(b_i) = \frac{\partial L}{\partial b_i}$$

O Cross Entropy Loss que usamos não depende diretamente de b, mas sim das probabilidades *a posteriori*.

Usando a regra da cadeia:

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\boxed{\partial p}} \cdot \frac{\partial p_j}{\partial output_i}$$

Probabilidade *a posteriori* da classe real.

Equações

Mais uma vez, a probabilidade a posteriori não depende diretamente de **b**, mas sim da saída. Aplicando a regra da cadeia mais uma vez:

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial b_i}$$

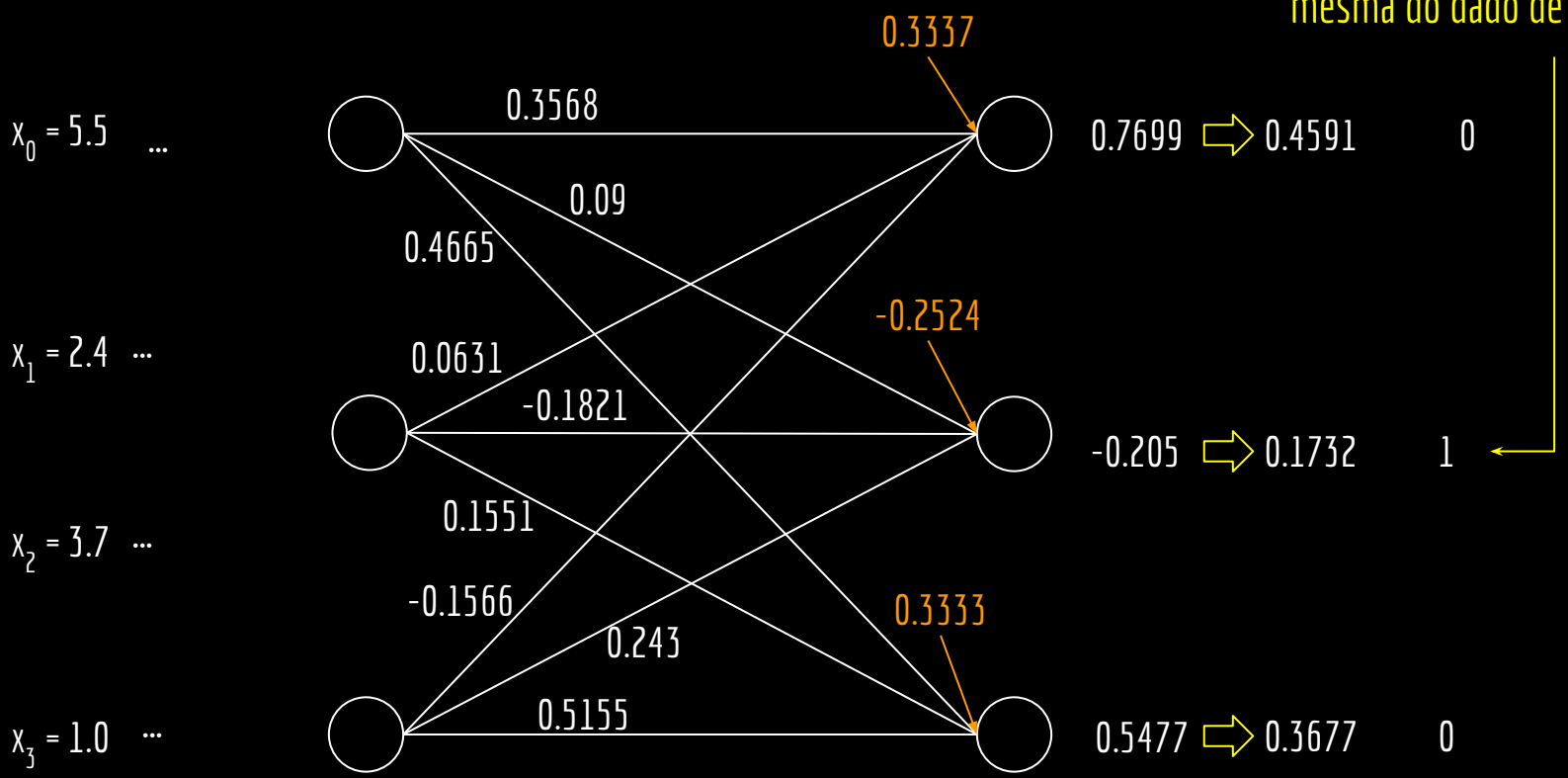
Equações

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} . \frac{\partial p_j}{\partial output_i} . \frac{\partial output_i}{\partial b_i}$$

$$L = -ln(p_{neuronio\_classe\_real})$$
$$p_i = softmax(output_i)$$
$$output_i = \mathbf{w_i}.\mathbf{outputs}^{layer-1} + b_i$$

Exemplo - Cross Entropy Loss

Supondo que estamos ajustando o bias do neurônio responsável por gerar a saída da classe que é a mesma do dado de treinamento.



Equações

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} . \frac{\partial p_j}{\partial output_i} . \frac{\partial output_i}{\partial b_i}$$

$\frac{\partial output_i}{\partial b_i} = 1$

$\frac{\partial L}{\partial p} = \frac{-1}{p_{real}}$

$\frac{\partial p_j}{\partial output_i} = p_j.(\delta_{ji} - p_i)$

Veja o passo a passo da derivada do softmax em
https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative

$$\delta_{ji} = \begin{cases} 1, & \text{se } j = i \\ 0, & \text{se } j \neq i \end{cases}$$

Sendo que:

$$L = -ln(p_{neuronio\_classe\_real})$$

$$p_i = softmax(output_i)$$

$$output_i = \mathbf{w_i}.\mathbf{outputs}^{layer-1} + b_i$$

Equações

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} . \frac{\partial p_j}{\partial output_i} . \frac{\partial output_i}{\partial b_i} = p_{real} - 1$$

Faça você mesmo

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial b_i}$$

$\frac{\partial output_i}{\partial b_i} = 1$

$\frac{\partial L}{\partial p} = \frac{-1}{p_{real}}$

$\frac{\partial p_j}{\partial output_i} = p_j.(\delta_{ji} - p_i)$

$$\delta_{ji} = \begin{cases} 1, & \text{se } j = i \\ 0, & \text{se } j \neq i \end{cases}$$

Sendo que:

$$L = -ln(p_{neuronio\_classe\_real})$$

$$p_i = softmax(output_i)$$

$$output_i = \mathbf{w_i}.\mathbf{outputs}^{layer-1} + b_i$$

Suponha que estamos ajustando o bias de um neurônio responsável por gerar a saída de uma classe **diferente** da do dado de treinamento. **Como fica a equação do gradiente?**

Faça você mesmo

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial b_i} = p_i$$

Onde p_i é a probabilidade *a posteriori* (resultado do softmax) do neurônio.

Atualizando os pesos

A cada iteração, os biases e pesos são atualizados de acordo com:

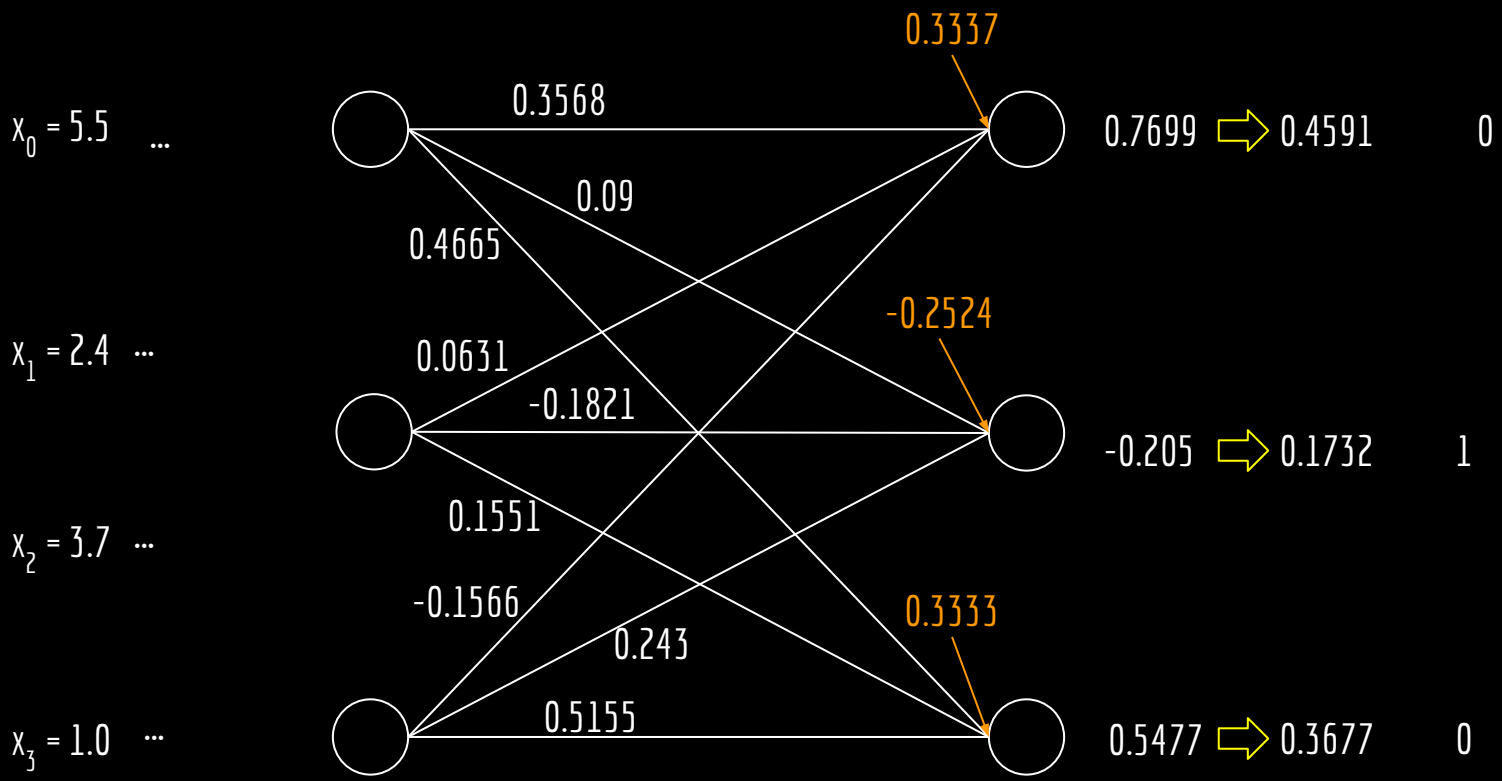
$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta \nabla E(\mathbf{w}^{\tau})$$

$$\mathbf{b}^{\tau+1} = \mathbf{b}^{\tau} + \eta \nabla E(\mathbf{b}^{\tau})$$

Onde τ é o índice da iteração, e η é o fator de aprendizagem.

Considerando $\eta = 0.05$

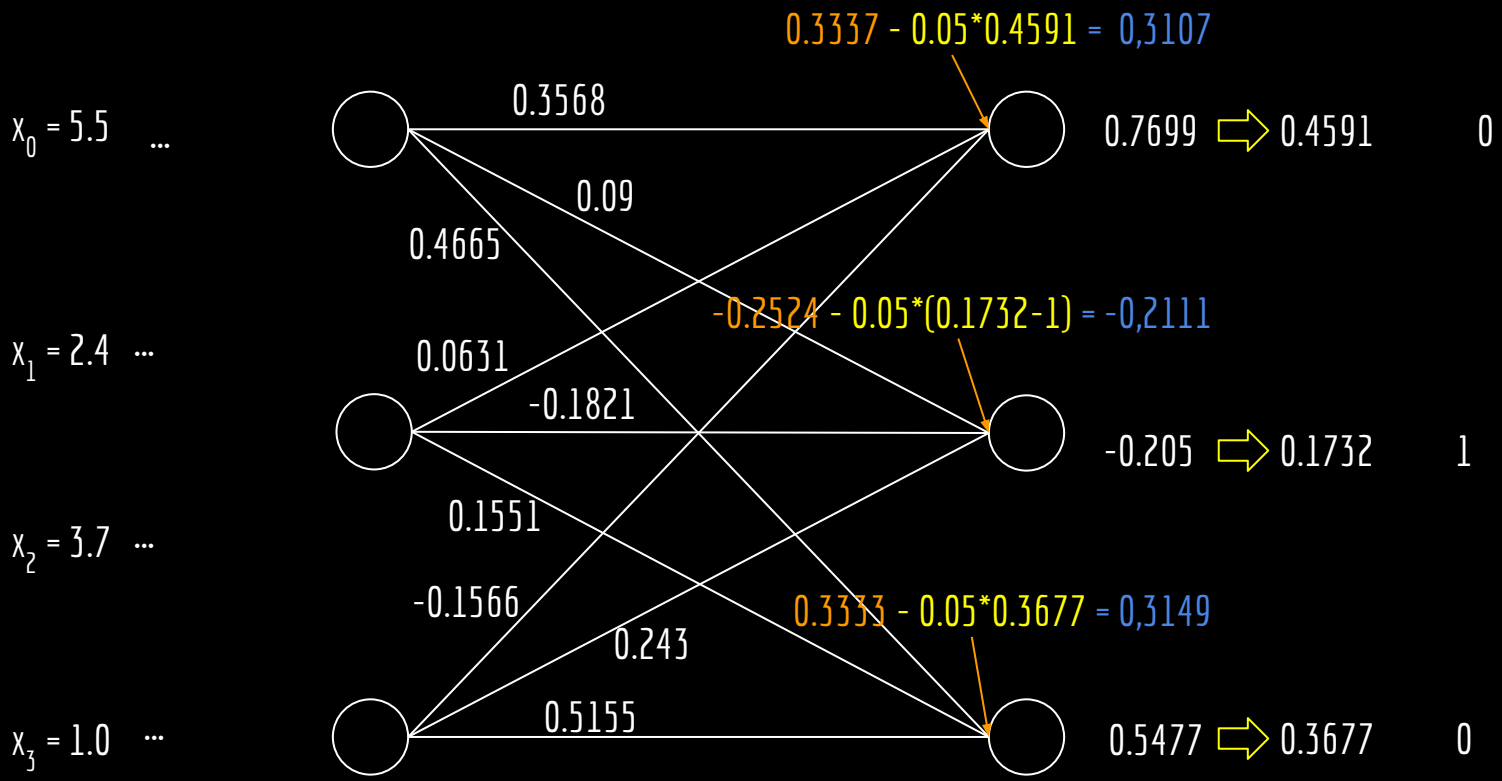
Aplicando no Exemplo



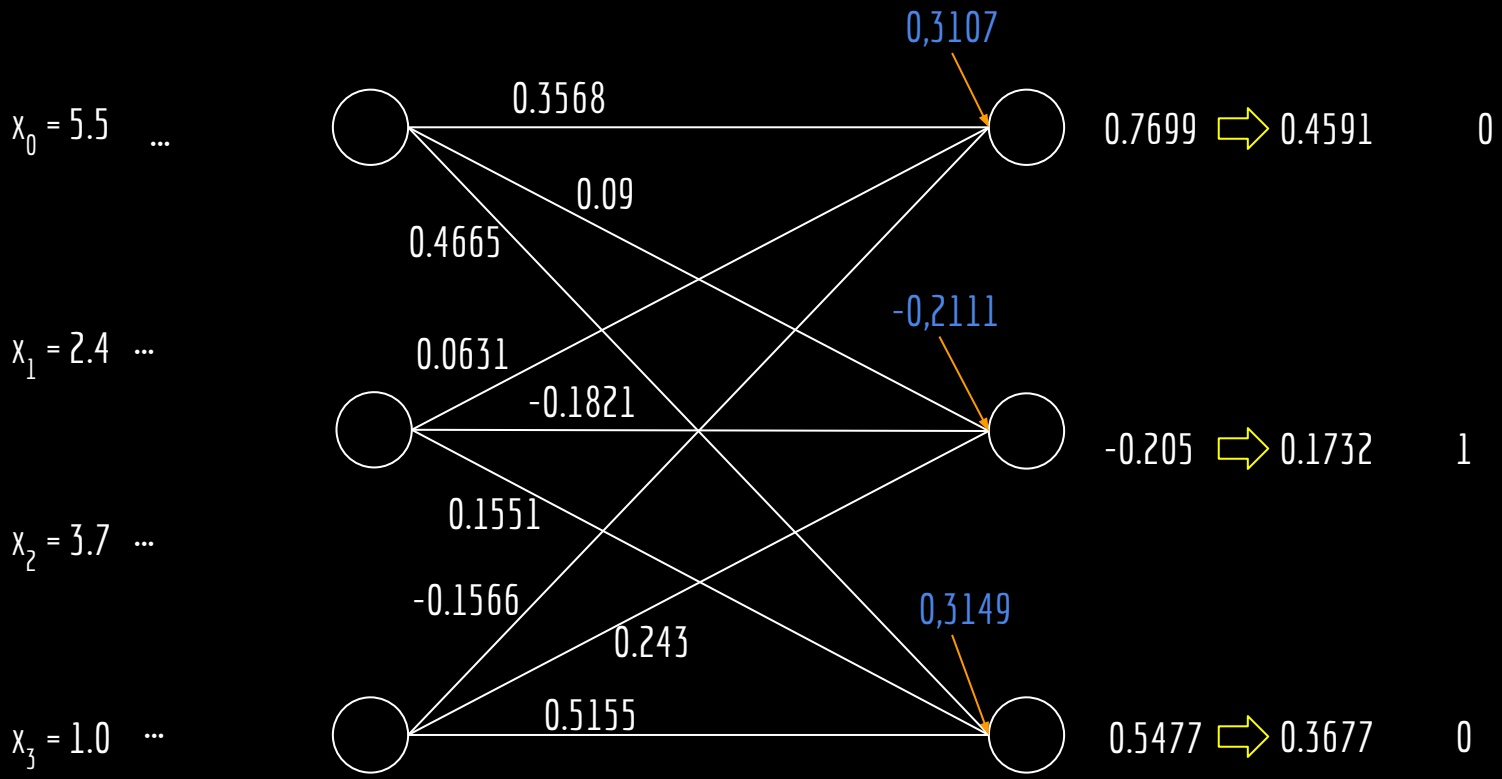
Aplicando no Exemplo

Considerando $\eta = 0.05$

Valor atualizado

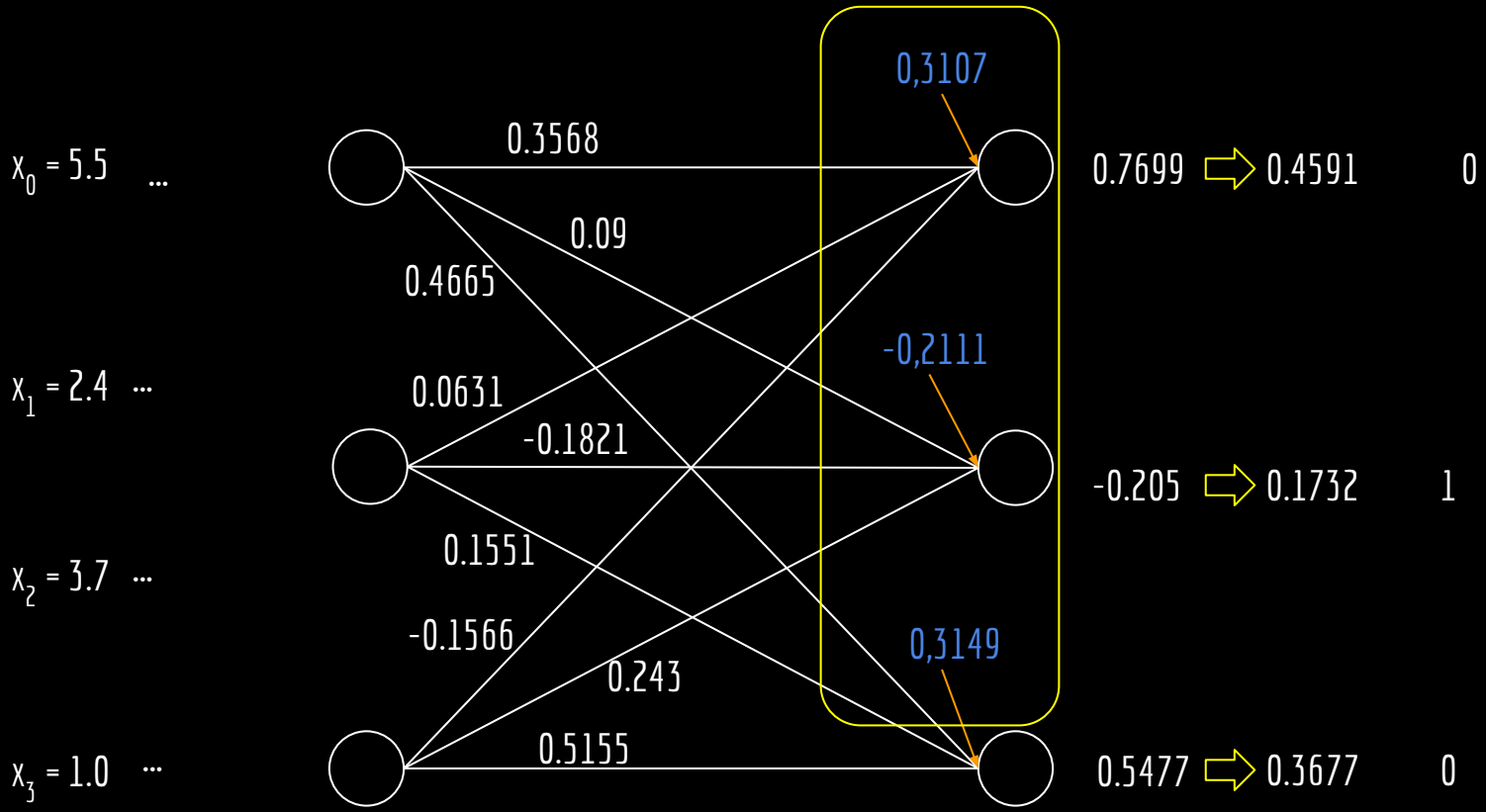


Aplicando no Exemplo



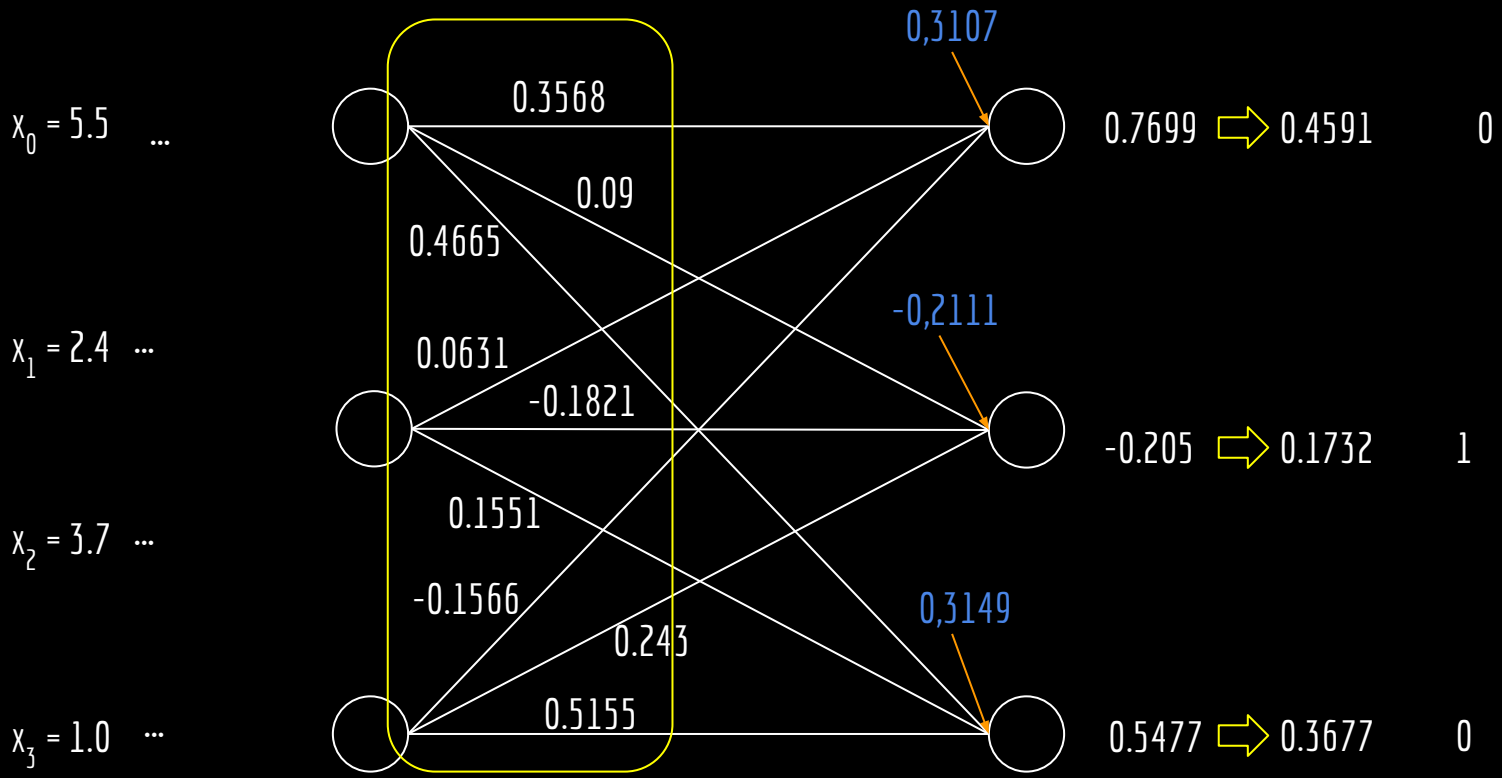
Atenção

Atenção: atualizamos **apenas o bias da última camada.**



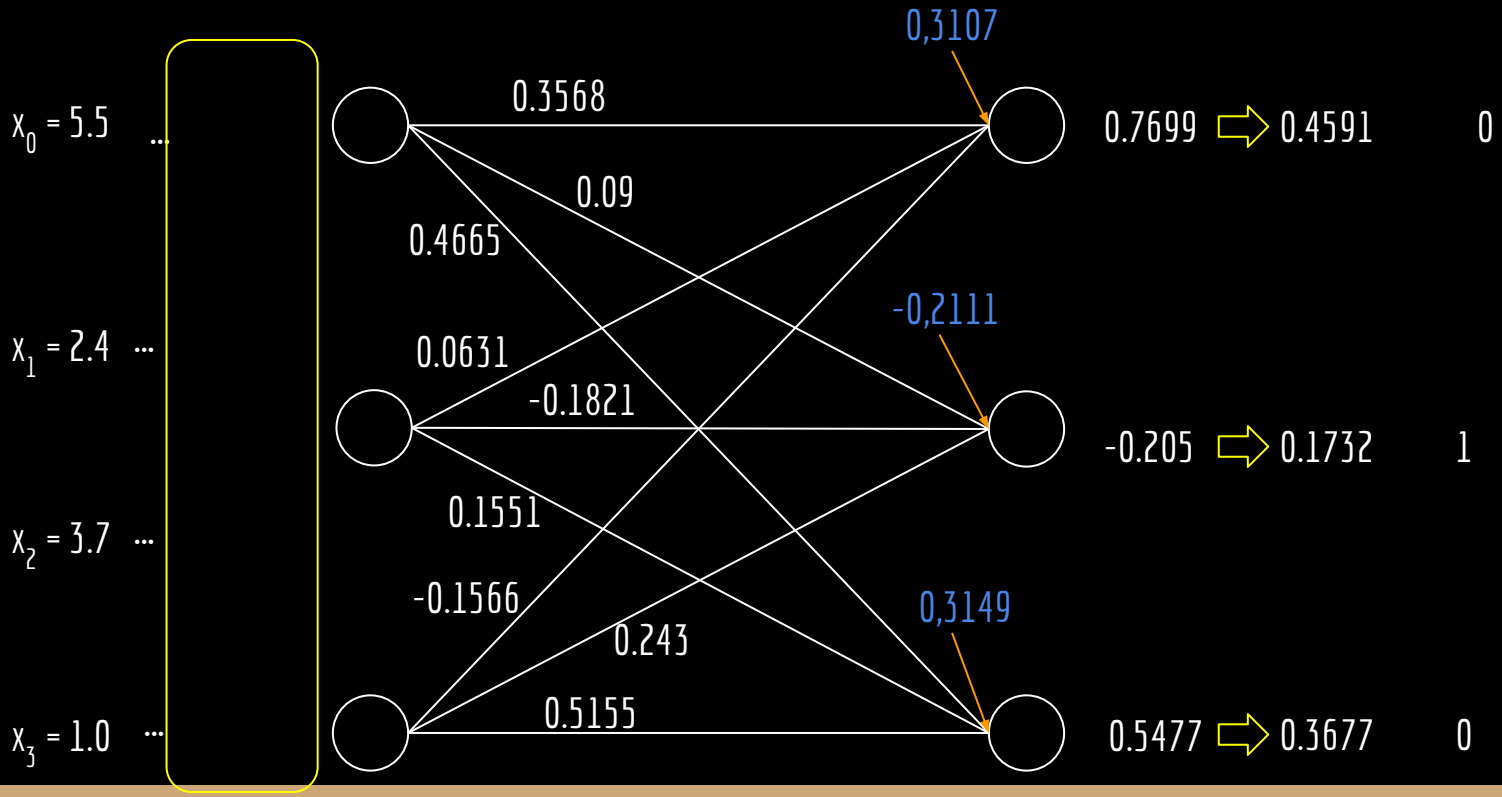
Atenção

Atenção: por um raciocínio semelhante, precisamos derivar as equações para atualizar os pesos.

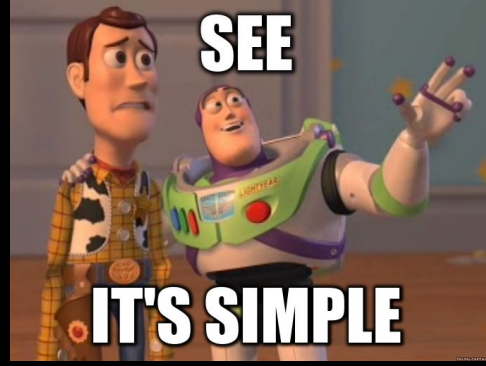


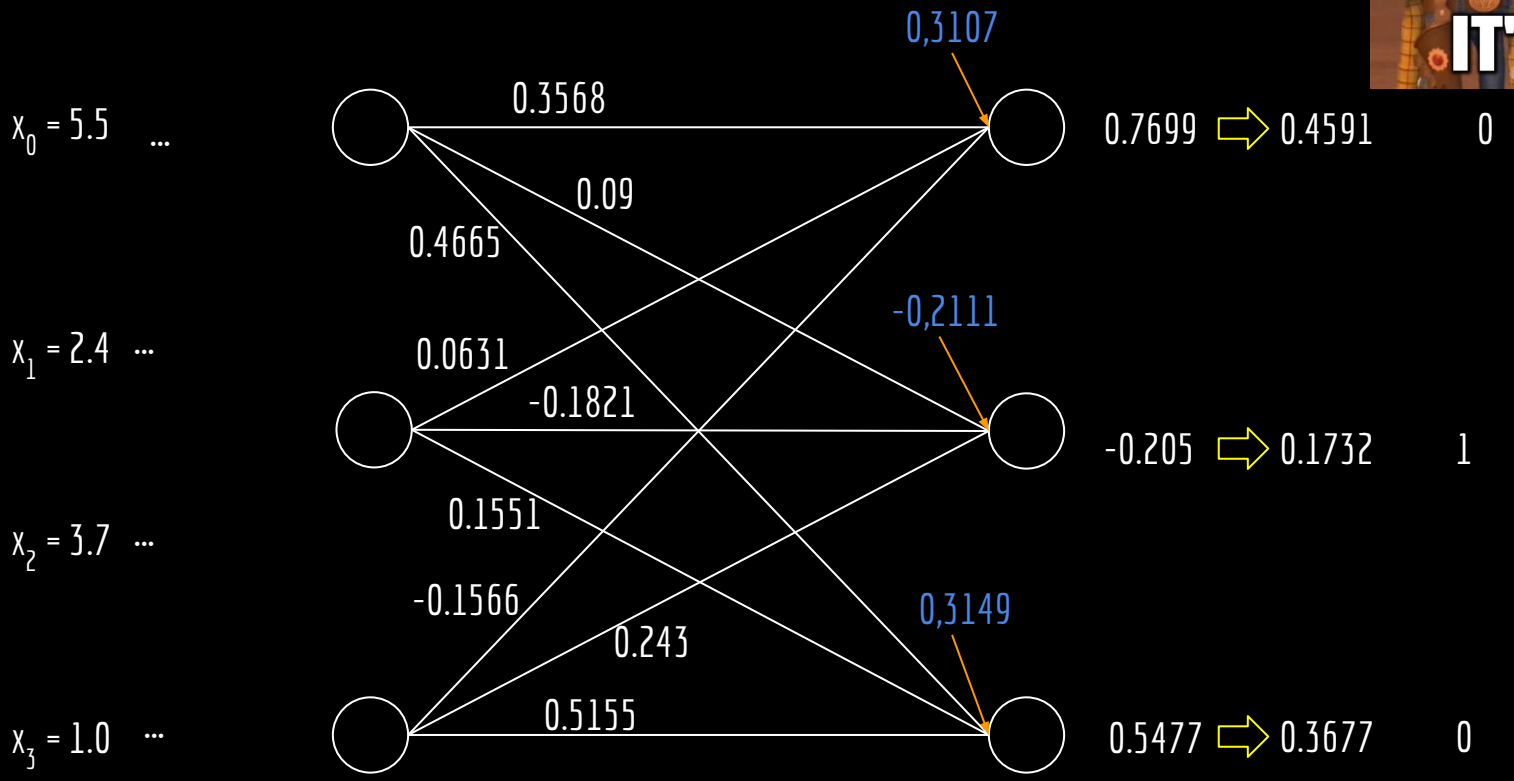
Atenção

Atenção: e precisamos derivar as expressões para atualizar as camadas anteriores. Veja nos vídeos de exercício que as equações são similares. **Todos os pesos e biases precisam ser atualizados!**



Atenção


SEE
IT'S SIMPLE


x_0 = 5.5 ...
x_1 = 2.4 ...
x_2 = 3.7 ...
x_3 = 1.0 ...
0.3568
0.09
0.4665
0.0631
-0.1821
0.1551
-0.1566
0.243
0.5155
0,3107
-0,2111
0,3149
0.7699
0.4591
0
-0.205
0.1732
1
0.5477
0.3677
0

Exercícios

1. Leia esse tutorial mostrando como montar redes simples com Pytorch.
   https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html
2. Assista a esse vídeo que mostra de uma forma um pouco diferente a atualização do último bias:
   https://youtu.be/xBEh66V9gZo?si=4Mo9-Gbwqaaqfz_7
3. Assista esse vídeo que mostra como as equações podem ser derivadas para as demais camadas.
   Cuidado. No vídeo, uma função de loss mais simples é usada, o que modifica um pouco as equações.
   https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4
4. Execute novamente o código do Colab. Salve o valor de bias da última camada antes de cada iteração de treino. Monte uma planilha e atualize esse valor manualmente após uma iteração de treino. Os seus resultados batem com os do Pytorch (dica: o Pytorch usa exatamente as mesmas equações que vimos hoje)?
5. Derive as equações para atualizar os pesos **w** da última camada da rede (resposta nos próximos slides).

Atualização dos pesos da última camada

Essa é a única parte da equação que muda quando comparado aos biases.

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial w_i}$$

$$\frac{-1}{p_{real}} \qquad p_j.(\delta_{ji} - p_i) \qquad \delta_{ji} = \begin{cases} 1, & \text{se } j = i \\ 0, & \text{se } j \neq i \end{cases}$$

Sendo que:

$$L = -ln(p_{neuronio\_classe\_real})$$

$$p_i = softmax(output_i)$$

$$output_i = \mathbf{w_i}.\mathbf{outputs}^{layer-1} + b_i$$

Atualização dos pesos da última camada

A equação da saída é $w_0.entrada_0 + w_1.entrada_1 + ... + b$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial w_i}$$

$$\frac{-1}{p_{real}} \qquad p_j.(\delta_{ji} - p_i) \qquad \delta_{ji} = \begin{cases} 1, & \text{se } j = i \\ 0, & \text{se } j \neq i \end{cases}$$

Sendo que:

$$L = -ln(p_{neuronio\_classe\_real})$$

$$p_i = softmax(output_i)$$

$$output_i = \mathbf{w_i.outputs}^{layer-1} + b_i$$

Atualização dos pesos da última camada

A equação da saída é $w_0.entrada_0 + w_1.entrada_1 + ... + b$
Derivando em função de w_i, sobra apenas o termo $entrada_i$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial w_i}$$

$$\frac{-1}{p_{real}} \qquad p_j.(\delta_{ji} - p_i) \qquad \delta_{ji} = \begin{cases} 1, & \text{se } j = i \\ 0, & \text{se } j \neq i \end{cases}$$

Sendo que:

$$L = -ln(p_{neuronio\_classe\_real})$$

$$p_i = softmax(output_i)$$

$$output_i = \mathbf{w_i.outputs}^{layer-1} + b_i$$

Atualização dos pesos da última camada

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial p} \cdot \frac{\partial p_j}{\partial output_i} \cdot \frac{\partial output_i}{\partial w_i}$$

A equação da saída é w_0.entrada$_0$ + w_1.entrada$_1$ + ... + b
Derivando em função de w_i, sobra apenas o termo entrada$_i$
Como a entrada da camada L é a saída de L-1, o termo é a saída do neurônio da camada L-1. Vamos chamar de $out_i^{(L-1)}$.

$$\frac{-1}{p_{real}}$$

$$p_j.(\delta_{ji} - p_i)$$

$$\delta_{ji} = \begin{cases} 1, & \text{se } j = i \\ 0, & \text{se } j \neq i \end{cases}$$

Sendo que:

$$L = -ln(p_{neuronio\_classe\_real})$$

$$p_i = softmax(output_i)$$

$$output_i = \mathbf{w_i}.\mathbf{outputs}^{layer-1} + b_i$$

Atualização dos pesos da última camada

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial p} . \frac{\partial p_j}{\partial output_i} . \frac{\partial output_i}{\partial w_i} = (p_{real} - 1) . w_i^{l-1}$$

Se o neurônio é responsável pela mesma classe do dado de treinamento.

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial p} . \frac{\partial p_j}{\partial output_i} . \frac{\partial output_i}{\partial w_i} = p_i . w_i^{l-1}$$

Caso contrário.

Referências

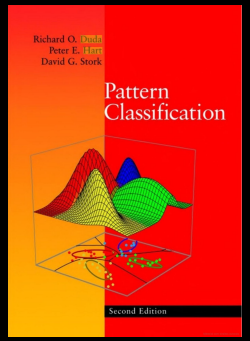
Duda, R. O., Hart, P. E., Stork, D. G. Pattern Classification. 2012.

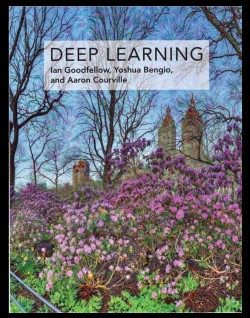
Goodfellow, I., Bengio, Y., Courville, A. Deep Learning. 2016.

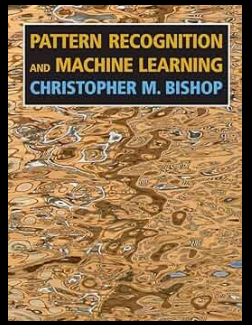
Bishop, C. M. Pattern Recognition and Machine Learning. 2006.

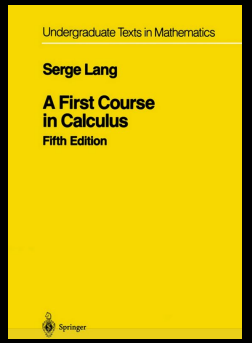
Lang, S. A First Course in Calculus. Suíça: Springer New York. 2012.

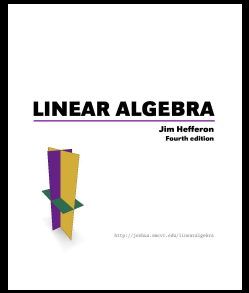
Hefferon, J. Linear Algebra. 2015.











Licença

Esta obra está licenciada com uma Licença Creative Commons Atribuição 4.0 Internacional.