Universidade Federal de Ciências da Saúde de Porto Alegre

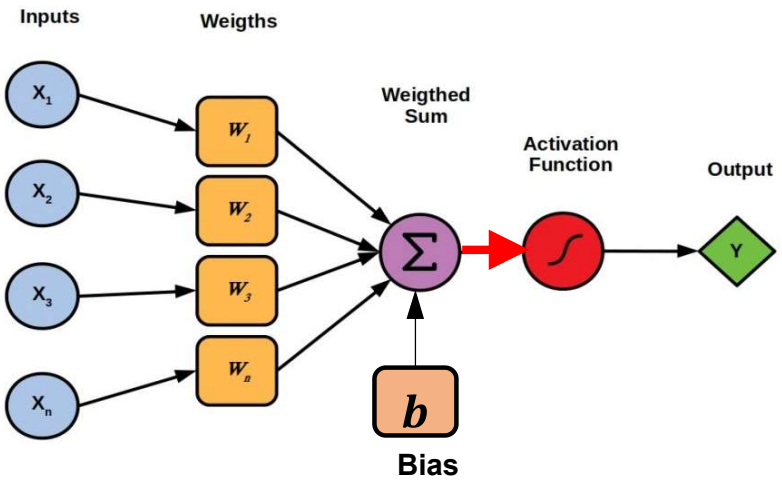
Aprendizado da rede

-- Otimização de parâmetros em redes neurais --

Prof. Viviane Botelho
vivianerb@ufcspa.edu.br

Objetivo da aula: Compreender os hiperparâmetros.

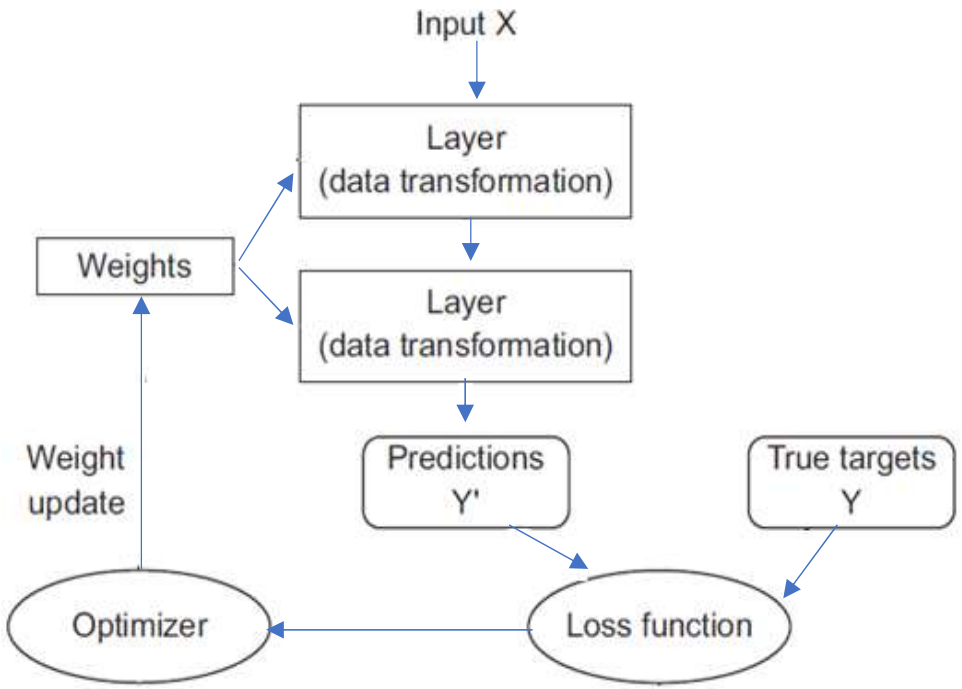
Lembrando...



Treinar a rede significa estimar os parâmetros (otimização)

Combinações lineares: $x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b$

Processo de treinamento: Obter os valores de pesos que gere menor diferença entre a saída predita pelo modelo e a saída dos dados de treino.



Função objetivo (*Loss*): Mede a discrepância entre os dados medidos e os dados preditos pelo modelo. Objetivo do treinamento é a minimização da *loss*.

Cuidado com a escolha da Loss. A rede tomará qualquer atalho que puder para minimizar a perda. Então esta perda tem que estar vinculada com o objetivo do problema. !

Keras

Probabilistic losses

- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

Regression losses

- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function

Mais comuns:

Predição: MSE ou MAE

Classificação binária: binary_crossentropy

Classificação multiclasses: categorical_crossentropy

*** Keras permite criação de Loss personalizada**

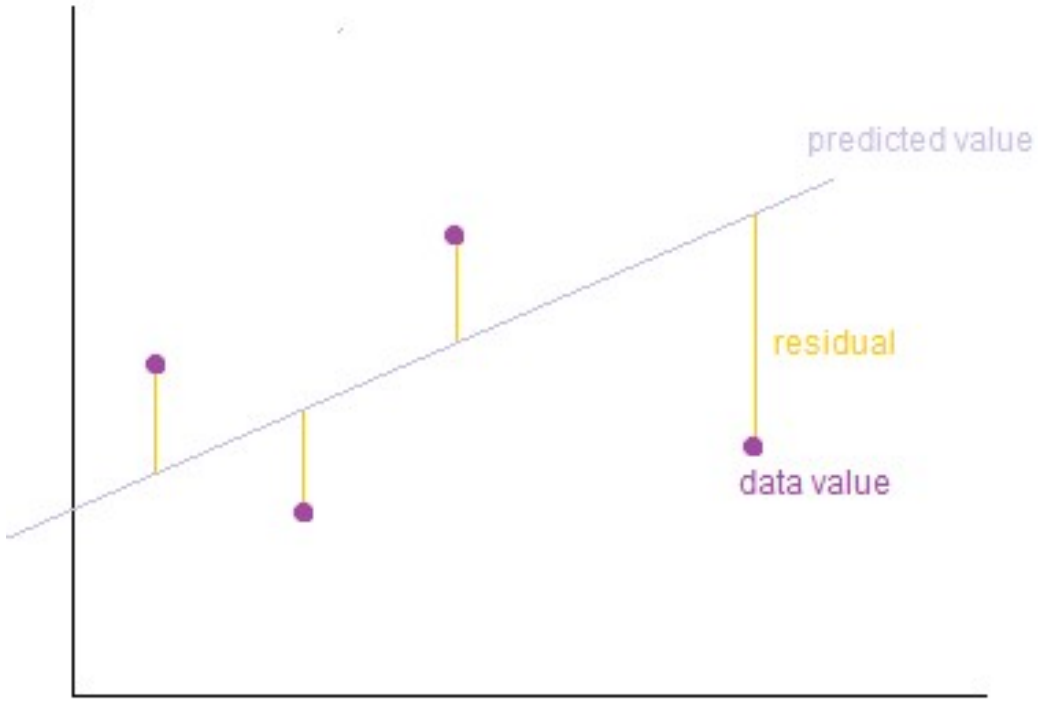
Função objetivo (*Loss*):

Predição: Mean Squared Error (MSE) or Mean absolute error (MAE)

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

$$MAE = \frac{\sum_{i=1}^{n} | y_i - \hat{y}_i |}{n}$$

y_i: saída medida da amostra i
$\hat{y}_i$: saída predita da amostra i
n: número de amostras

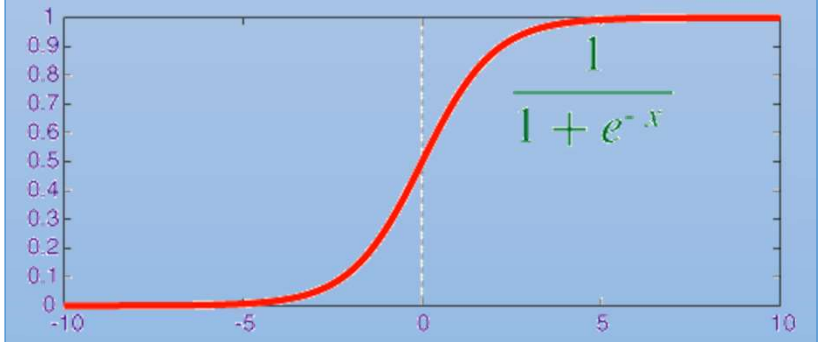


Função objetivo (*Loss*):

Classificação binária:
binary_crossentropy (log loss)

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Lembrando: Saída da rede a função de ativação é uma sigmoid (ou similar): Probabilidade de pertencer a classe 1.



- y_i: saída medida: 0 ou 1
- $p(y_i)$: saída predita (probabilidade de pertencer a classe 1)
- N: número de amostras

Por exemplo:

Real (y_i)	Probablidade de ser 1 $p(y_i)$	Loss
0	0,05	0,02
0	0,95	1,30
1	0,95	0,02
1	0,05	1,30
1	0,52	0,28
0	0,52	0,32

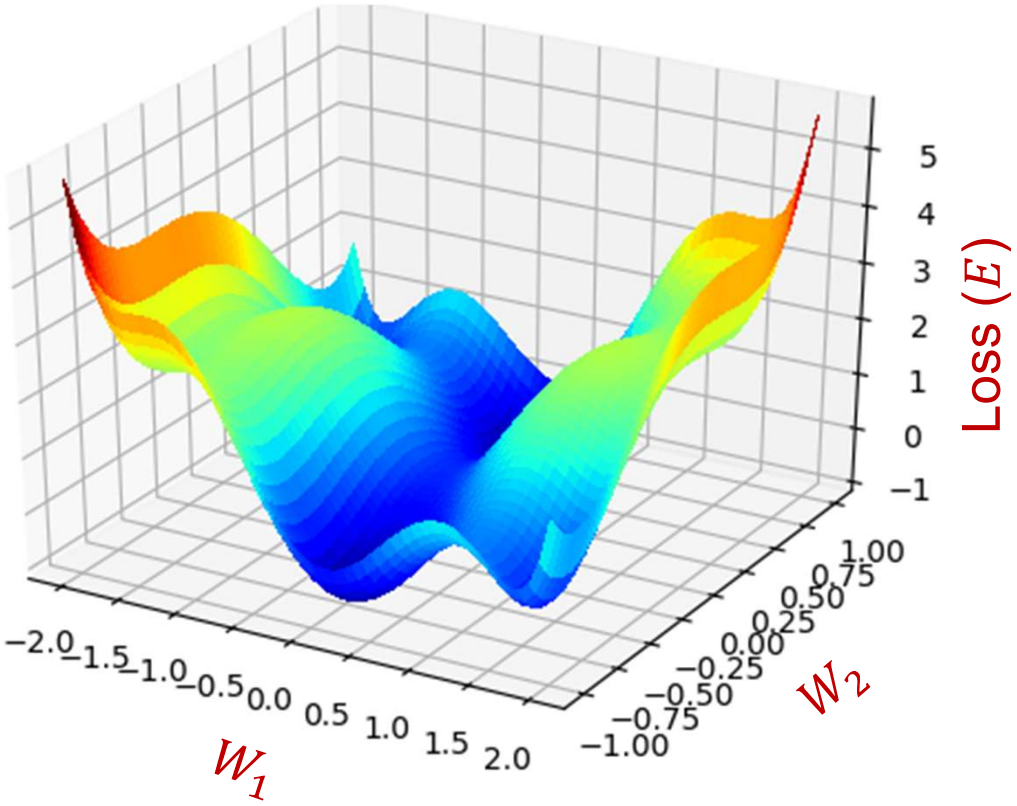
Resumindo: Quantifica se o modelo “errou feio” ou ficou indeciso indeciso?

Otimizador: Gradiente descendente

Como que a rede decide qual será o próximo conjunto de pesos a ser testado

?

Exemplo: Rede neural com dois pesos para ajustar

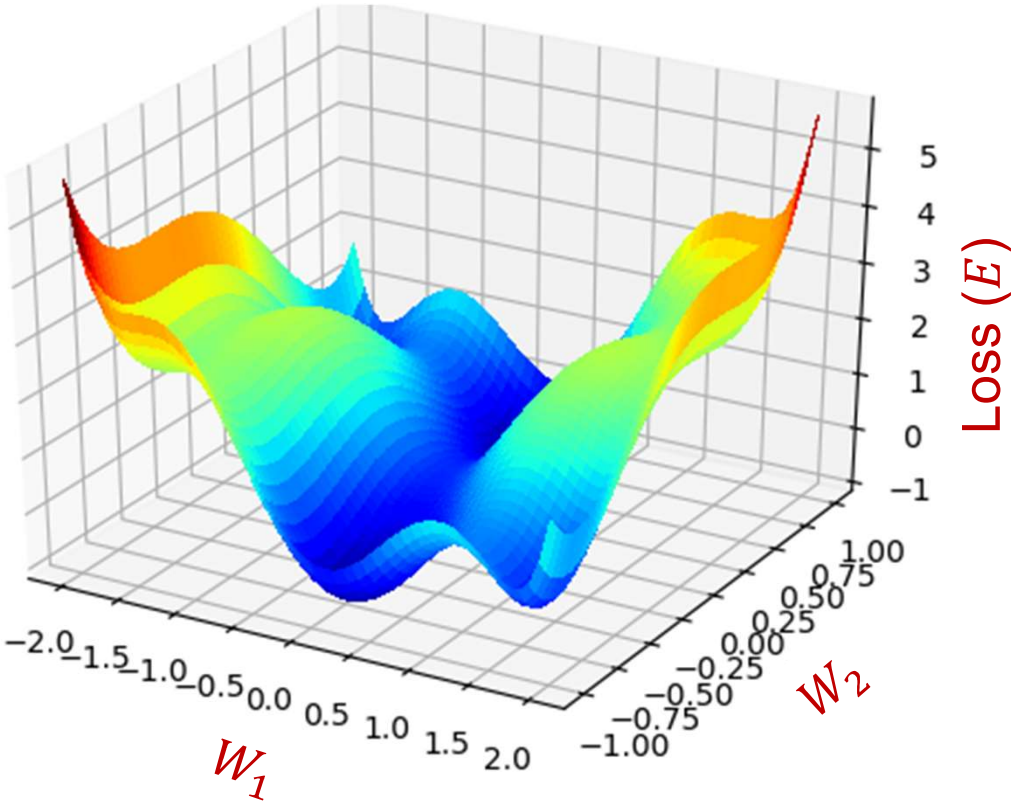


Otimizador: Gradiente descendente

Como que a rede decide qual será o próximo conjunto de pesos a ser testado

?

Exemplo: Rede neural com dois pesos para ajustar



Vetor Gradiente: Direção de máxima variação

$$\nabla f = \langle f_x, f_y, f_z \rangle = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}$$

Por essa razão a função de ativação deve ser diferenciável

!

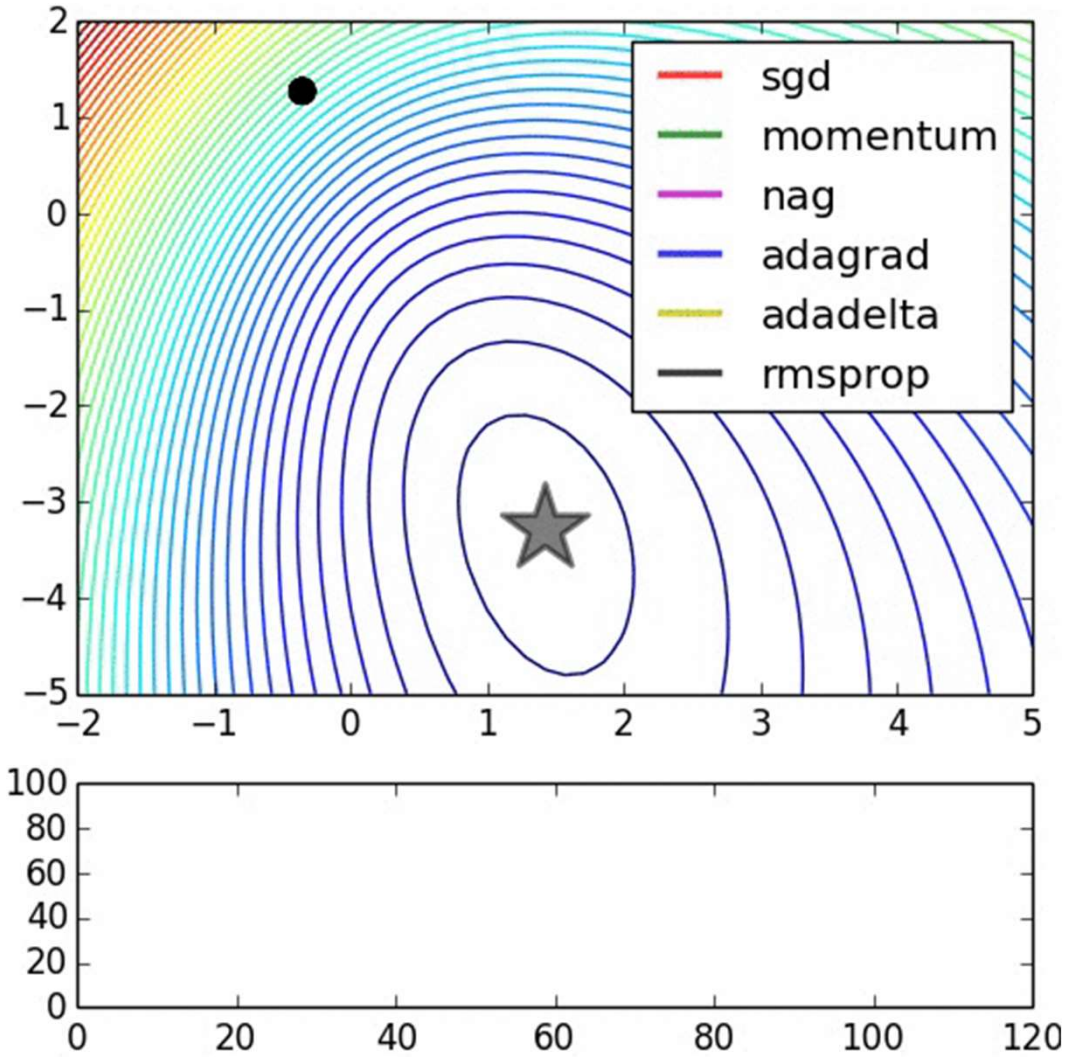
Otimizador:

Variações do Gradiente descendente

Keras

Available optimizers

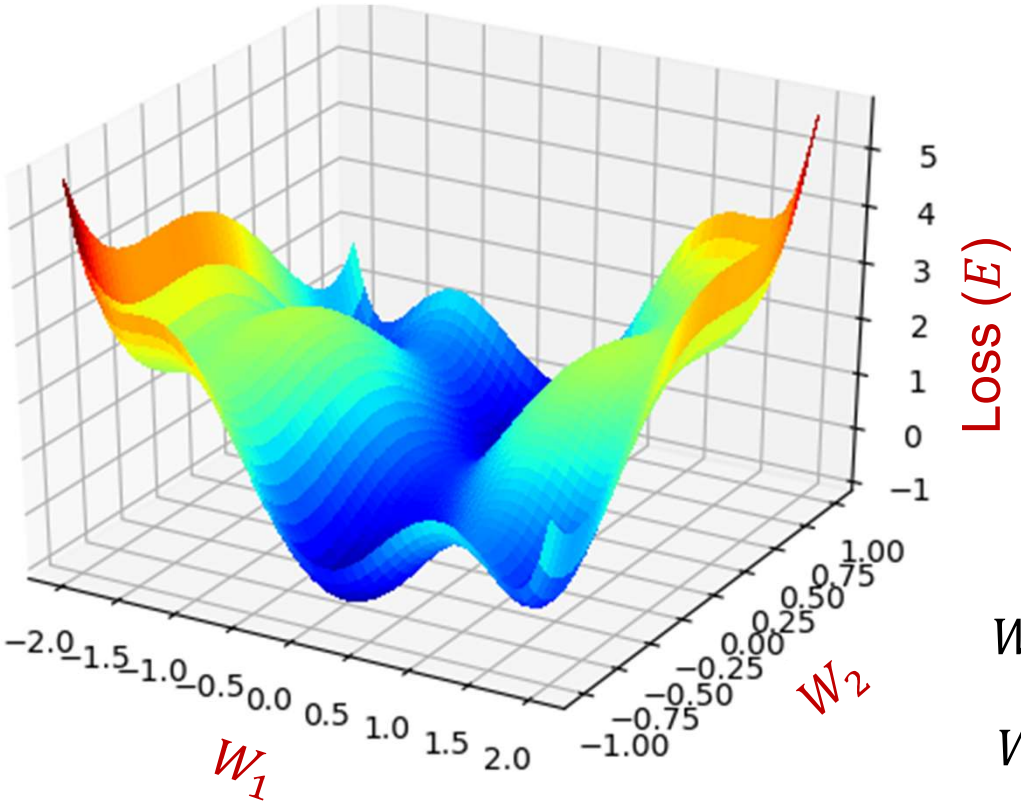
- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl



Otimizador: Gradiente descendente

Como que a rede decide qual será o próximo conjunto de pesos a ser testado ?

Exemplo: Rede neural com dois pesos para ajustar



Vetor Gradiente: Direção de máxima variação

$$\nabla f = \langle f_x, f_y, f_z \rangle = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}$$

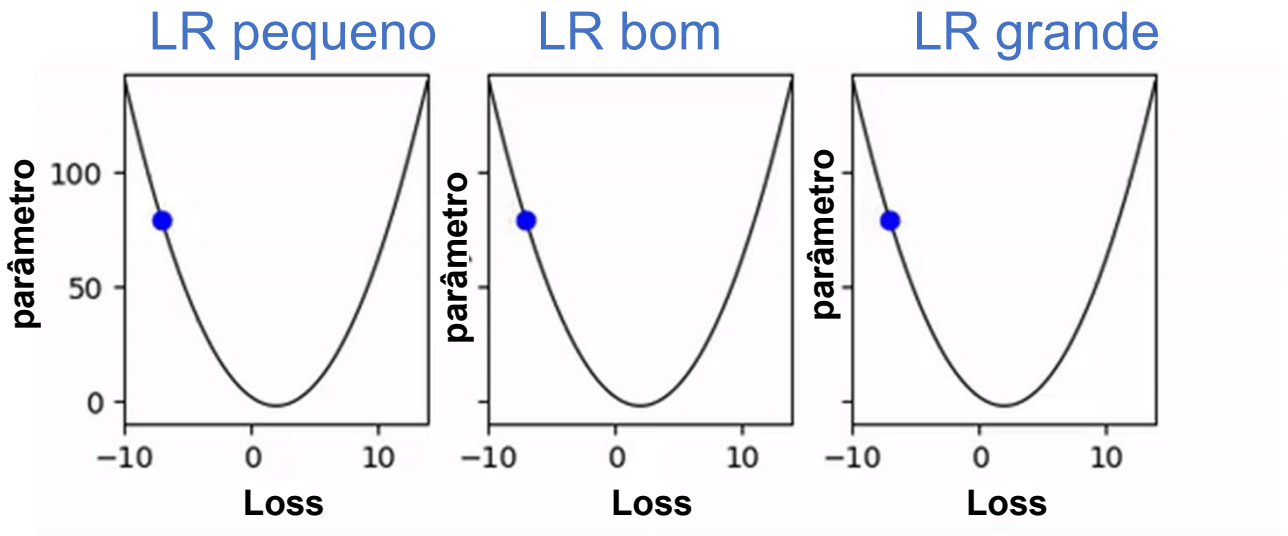
Por essa razão a função de ativação deve ser diferenciável !

$$W_1^{new} = W_1^{old} - \propto \nabla E$$

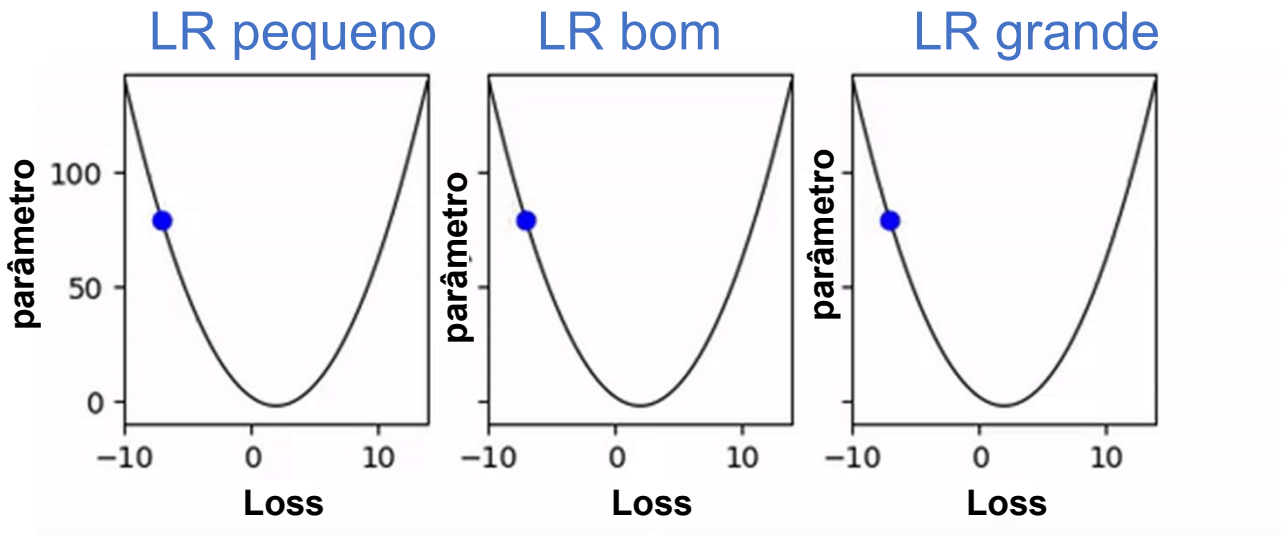
$$W_2^{new} = W_2^{old} - \propto \nabla E$$

$\propto$ é *a taxa de aprendizado*

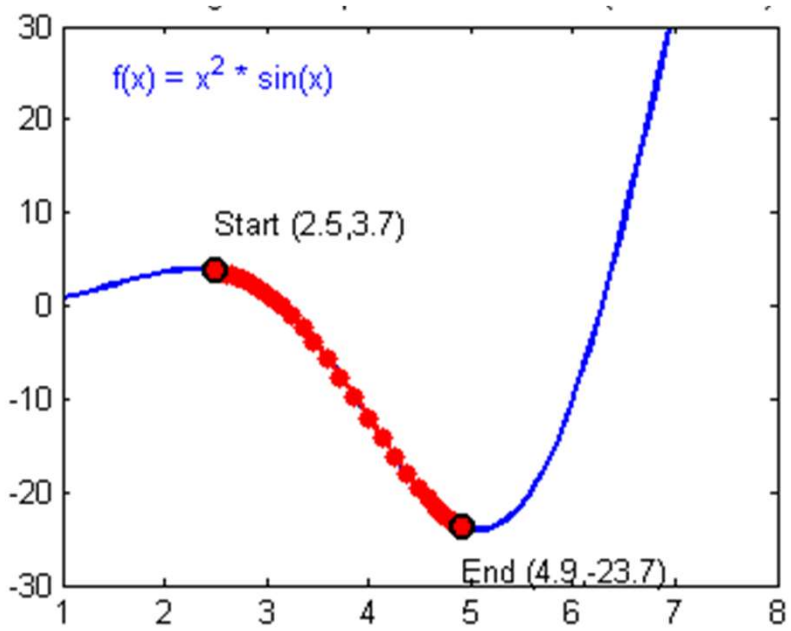
Otimizador: Como a *learnig rate* impacta no treinamento da rede?



Otimizador: Como a *learnig rate* impacta no treinamento da rede?



Alternativa: *learnig rate* adaptativo



Adam (default Keras) utiliza LR adaptativo. Ainda assim definir o influencia muito na qualidade e velocidade do treinamento!

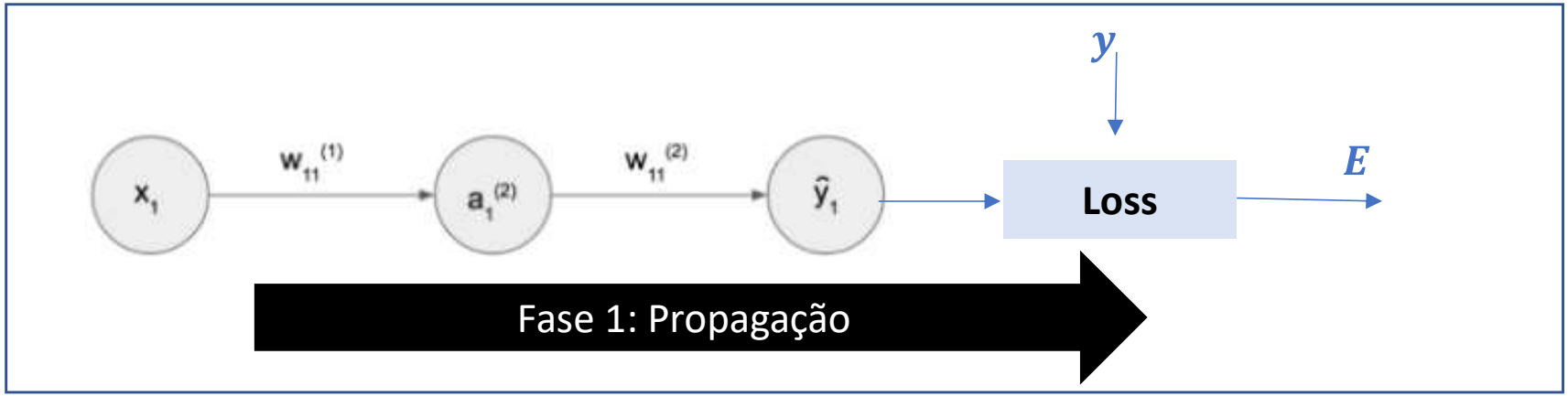
!

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
```

UFCSPA

Algoritmo de backpropagation

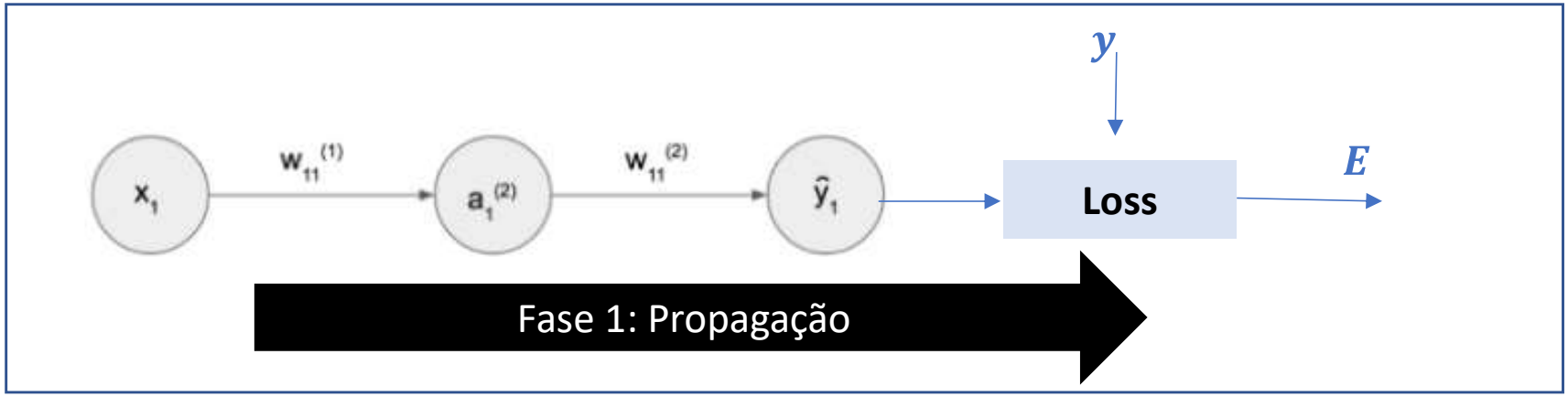
- Partindo de um valor aleatório para os parâmetros ($w_{11}^{(2)}{}_x$ e $w_{11}^{(w)}$)



$$E = f(\widehat{y_1}) = f(f(w_{11}^{(2)} a_1^{(2)})) = f(f(w_{11}^{(2)} f(w_{11}^{(1)} x_1)))$$ ➔ **Função composta**

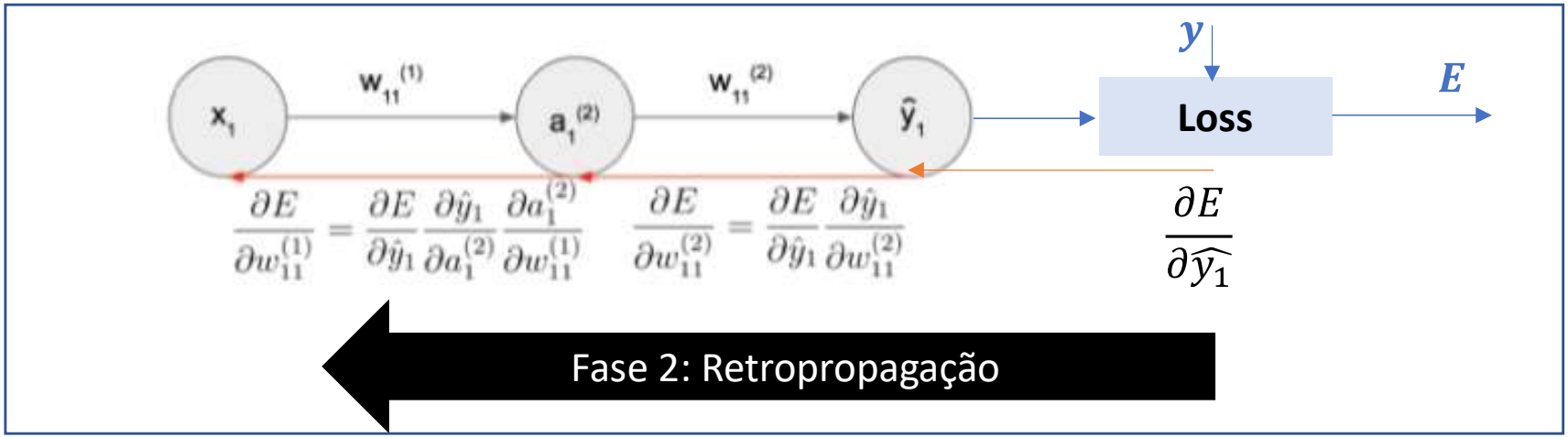
Algoritmo de backpropagation

- Partindo de um valor aleatório para os parâmetros ($w_{11}^{(2)}{}_x$ e $w_{11}^{(w)}$)



$$E = f(\hat{y_1}) = f(f(w_{11}^{(2)} a_1^{(2)})) = f(f(w_{11}^{(2)} f(w_{11}^{(1)} x_1)))$$ ➔ **Função composta**

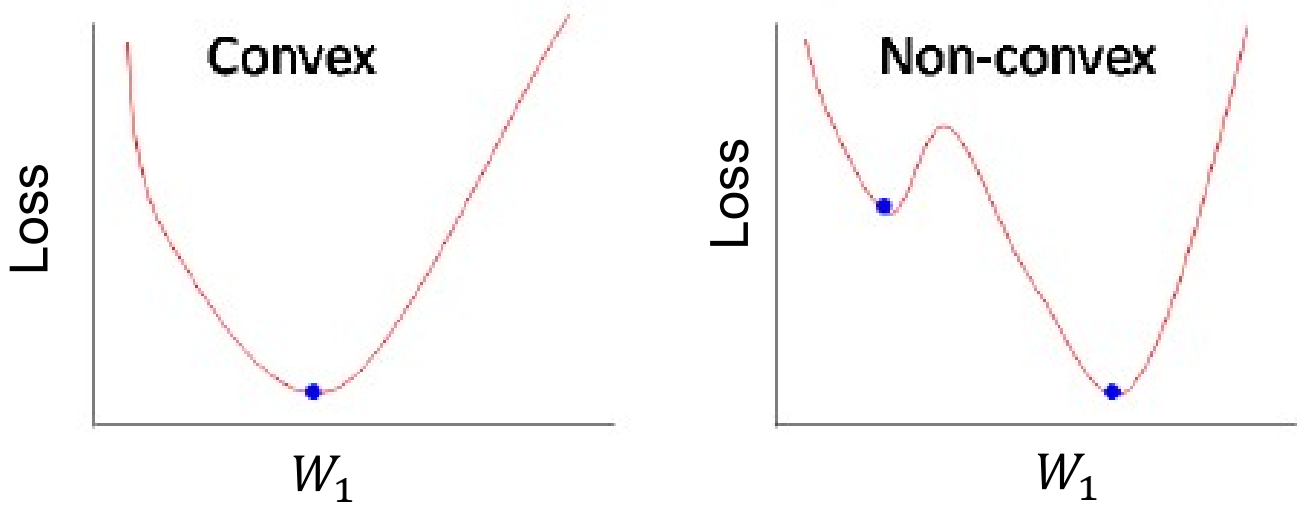
- Atualizar os pesos: Computar derivadas parciais da $Loss$ (L) em relação a todos os parâmetros $\left(\frac{\partial E}{\partial w_{11}^{(2)}} \; e \; \frac{\partial L}{\partial w_{11}^{(1)}} \right)$ ➔ Regra da cadeia



Otimizador: Inicialização dos parâmetros

Usualmente os parâmetros são inicializados de um valor aleatório qualquer (estado aleatório definido pela máquina)

Exemplo: Rede com 1 peso (W_1)

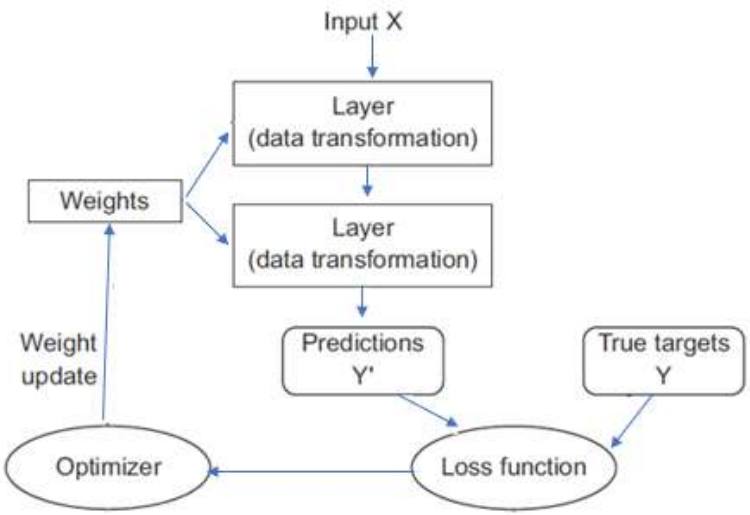


Fixar o “chute inicial” dos parâmetros facilita a reprodutibilidade dos resultados.

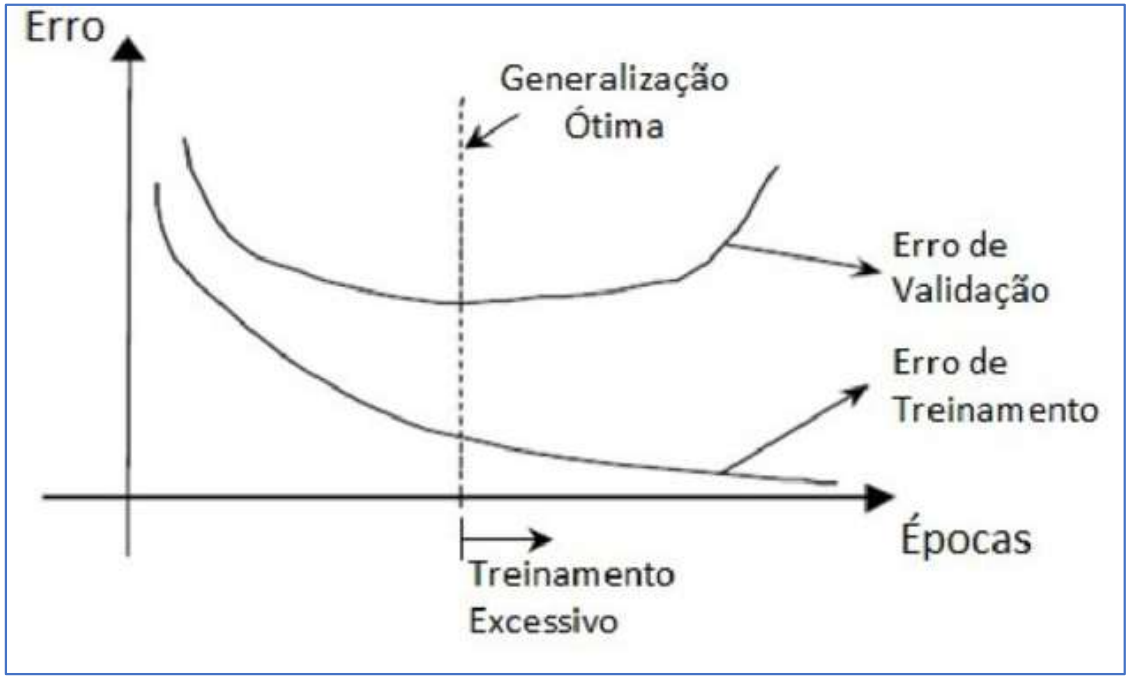
Fixar o “chute inicial” não garante resultados de boa qualidade.

Mesmo fixando os parâmetros iniciais, os resultados podem não ser exatamente iguais, ideal é repetir mesmo experimento várias vezes!

Número de épocas: Número de vezes que o processo iterativo vai se repetir. Uma época significa treinar a rede neural com todos os dados **de treinamento para um ciclo**.



Cuidado com o número excessivo de épocas.



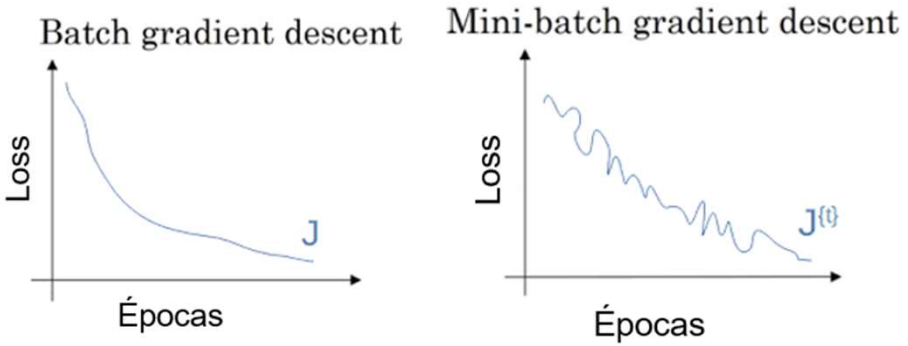
Alternativa no Keras: Critério de parada a partir do monitoramento de uma métrica

```
tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0,
    patience=0,
    verbose=0,
    mode="auto",
    baseline=None,
    restore_best_weights=False,
)
```

Tamanho do lote (batch size)

Número de amostras de treinamento utilizadas em cada iteração (isto é, a cada atualização dos pesos).

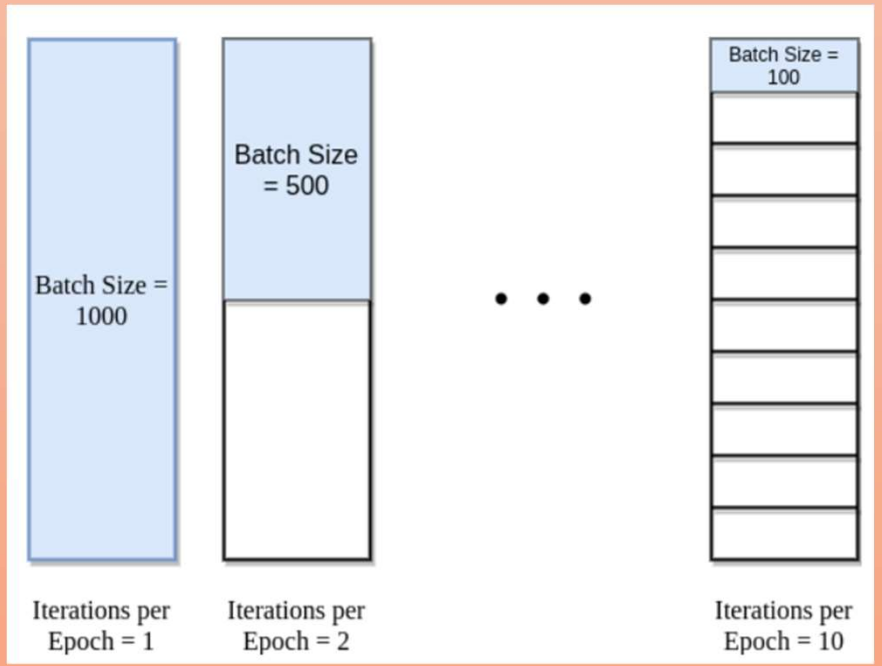
- **Batch Gradient Descent**. Batch Size = Size of Training Set
- **Stochastic Gradient Descent**. Batch Size = 1
- **Mini-Batch Gradient Descent**. 1 < Batch Size < Size of Training Set



Obs: Batch size default do Keras é 32!

A cada época, todo o conjunto de dados é avaliado, independente do tamanho do lote

Exemplo: Conjunto com 1000 amostras de treino

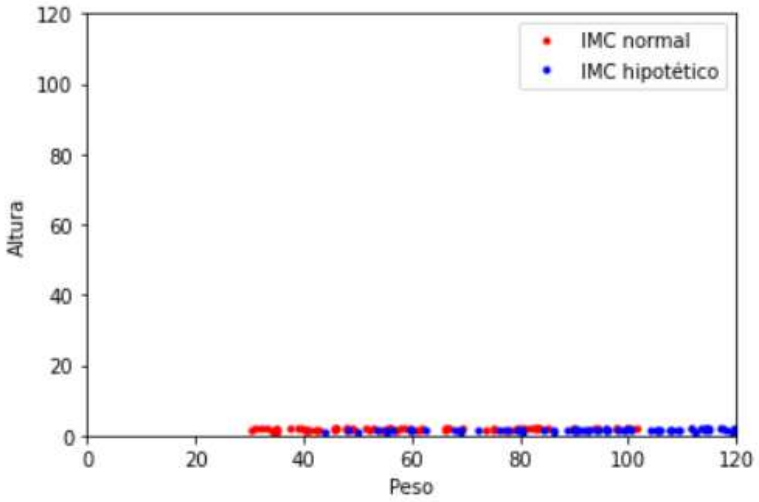


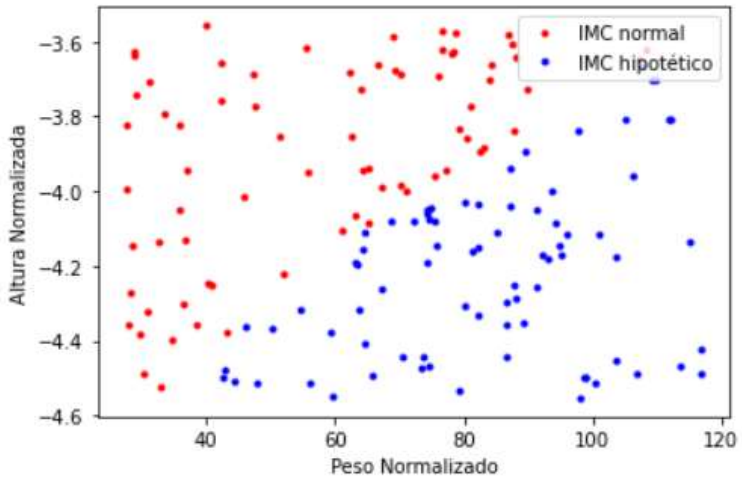
Normalização dos dados: ***Batch normalization***

Deixar os dados de entrada na mesma ordem de grandeza

Normalização das entradas

Exemplo hipotético: Entradas são peso e altura e saída é a classe IMC normal e IMC elevado

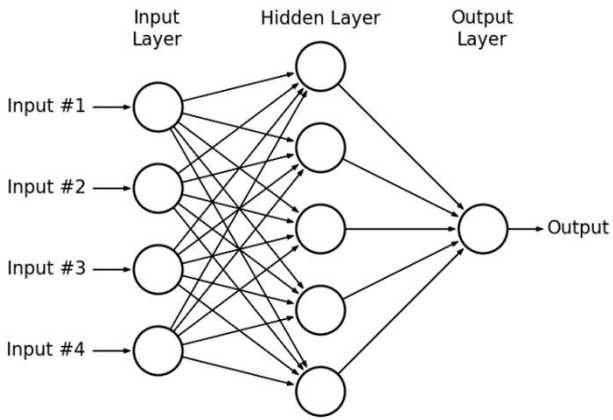




$$X_{i_{norm}} = \frac{X_i - \overline{X_i}}{\sigma_{X_i}}$$

Normalizando os dados apenas antes de sua entrada na rede:

- Pode ser que a normalização de cada *batch* não seja representativa do dataset inteiro.
- Entradas das camadas ocultas não estão normalizadas.



Solução é utilizar *Batch normalization* !

Normalização dos dados: ***Batch normalization***

Batch Normalization é um tipo de camada introduzida entre as camadas densa que pode normalizar de forma adaptativa os dados.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

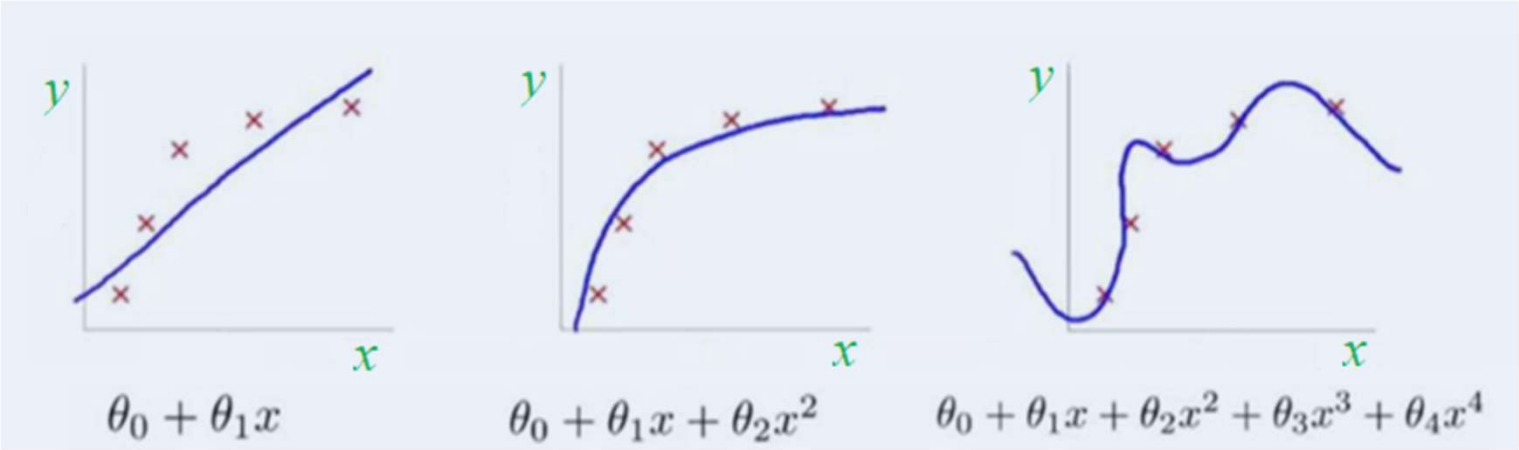
Batch normalization melhora a propagação do gradiente durante a otimização:

- Acelera o treinamento.
- Permite *learning rates* maiores.
- Melhora o processo de treinamento, como um todo.

Regularização: Técnicas para reduzir *overfitting*

Ideia:

- Quanto mais simples o modelo (com menor número de parâmetros), menor é a chance de *overfitting*.



- Forma sistemática "para zerar" os parâmetros desnecessários.

1) Inclusão de penalização na *Loss* (por exemplo, se aplicado a MSE)

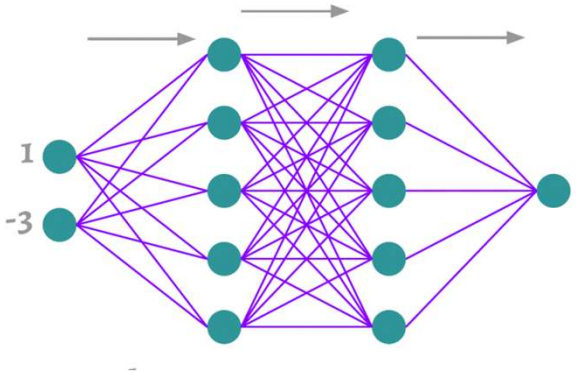
$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} + \begin{cases} \text{L1: } \lambda \sum_{j=0}^{M} |W_j| \\ \text{L2: } \lambda \sum_{j=0}^{M} W_j^2 \end{cases}$$

$\lambda > 0$: parâmetro de regularização!

Regularização: Técnicas para reduzir *overfitting*

2) Camada de *Dropout*

Consiste em zerar **aleatoriamente** (e temporariamente) alguns neurônios ocultos em cada iteração.



Em cada etapa do treinamento, uma rede diferente é gerada. Assim, conceitualmente, o procedimento é semelhante ao uso de um conjunto de muitas redes diferentes. No momento do teste, toda a rede é usada (todas as unidades), mas com pesos reduzidos.

Por que funciona: O *dropout* evita que as unidades se "coadaptem", isto é, as unidades aprendem a não depender muito umas das outras.

Efeitos:

- Força a rede a aprender quais são os neurônios mais robustos.
- O *dropout* aumenta o número de iterações necessárias para convergir.
- O tempo de cada iteração é menor pois a rede é mais simples.

Referências

https://towardsdatascience.com/multi-class-metrics-made-simple-part-i-precision-and-recall-9250280bddc2

https://scikit-learn.org/0.15/auto_examples/plot_roc.html

https://towardsdatascience.com/on-optimization-of-deep-neural-networks-21de9e83e1

https://www.untitledkingdom.com/blog/let-me-introduce-you-to-neural-networks-feaa7aa2afe2

https://www.researchgate.net/publication/341817070_Artificial_neural_networks_for_neuroscientists_A_primer/figures?lo=1

https://pt.linkedin.com/pulse/m%C3%A9todos-de-otimiza%C3%A7%C3%A3o-do-gradientdescent-ruan-costa

https://www.baeldung.com/cs/epoch-neural-networks