

XI Encontro de Física e Astronomia da UFSC

Introdução às redes neurais com Python e Keras

Prof. Dr. Robson da Silva Oliboni



Introdução às redes neurais com Python e Keras

Aspectos técnicos de redes profundas

Sumário

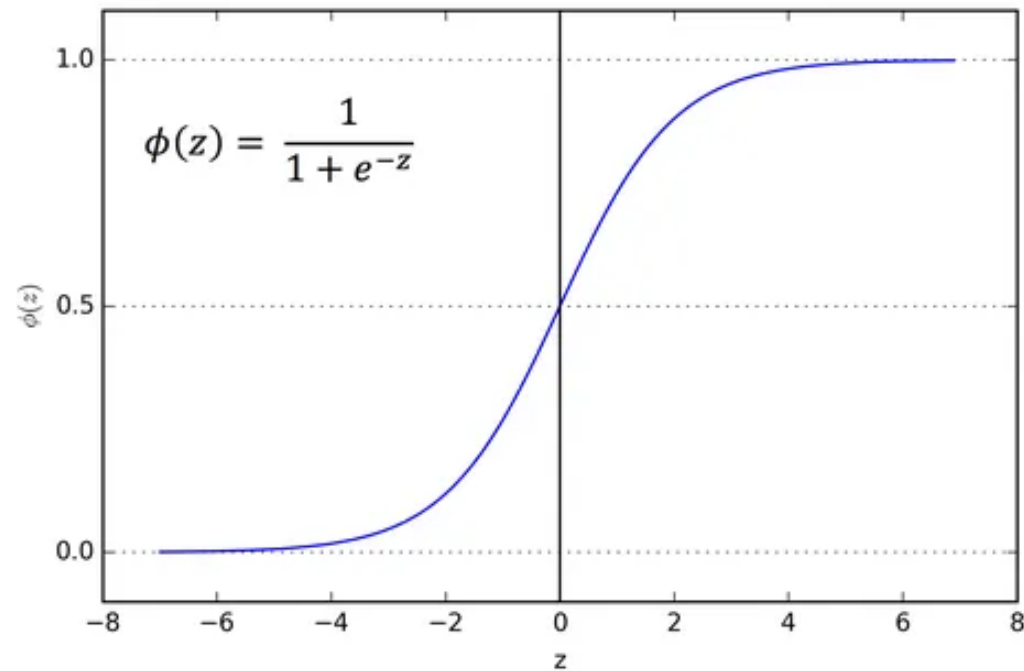
1. Funções de ativação
2. Problemas com gradientes
3. Outras funções de ativação
4. Inicialização de pesos
5. Aperfeiçoamento dos hiperparâmetros
6. Otimizadores mais rápidos
7. Aprendizado por transferência

Possíveis problemas com o treinamento de redes profundas

- Gradientes de fuga, dissipação de gradientes ou explosão de gradientes → Funções de ativação + algoritmos de inicialização;
- Tempo de treinamento → Cronogramas de aprendizado, melhores otimizadores;
- Quantidade de dados → Aprendizado por transferência; pré-treinamento não-supervisionado.

Funções de ativação

Característica importante: contradomínio ou limites da função.

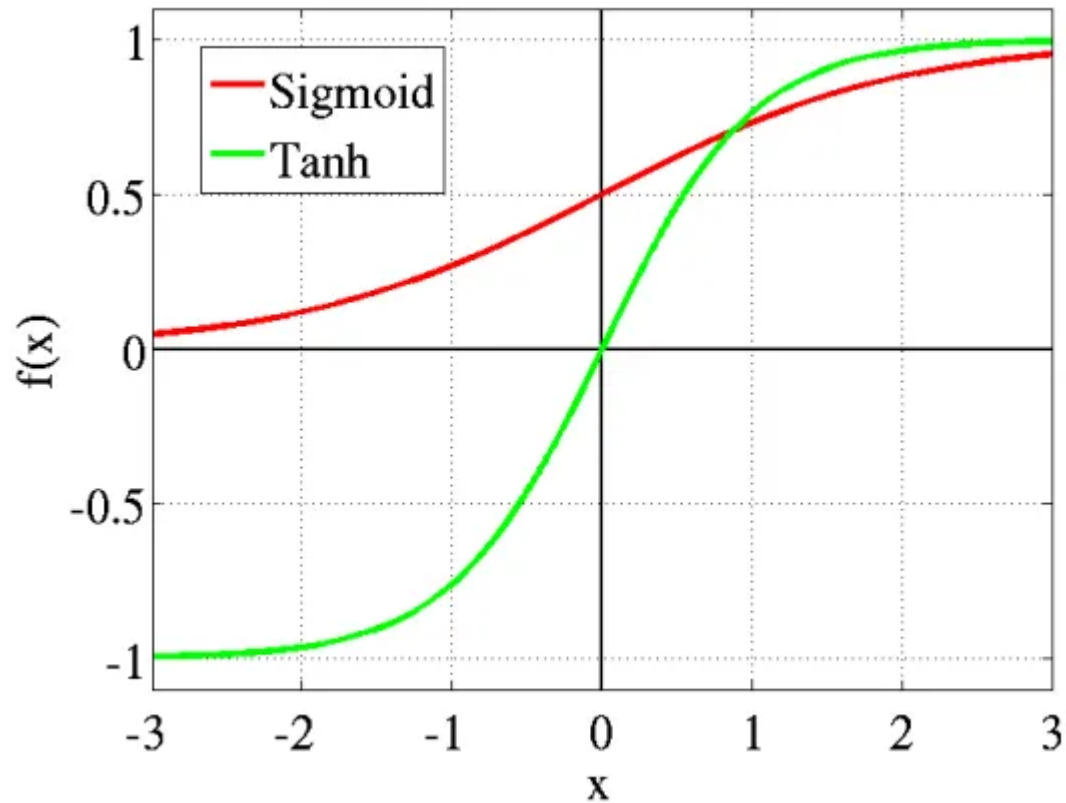


Função logística ou sigmóide: outputs entre 0 e 1 → útil para modelar probabilidades;

Função contínua e diferenciável.

Funções de ativação

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Função tangente hiperbólica: saídas (*outputs*) entre -1 e $+1$;

Assimétrica em relação a origem (melhor para treinamento).

Problemas de gradientes de fuga e explosão de gradientes

Xavier Glorot e Yoshua Bengio identificaram que a combinação da função de ativação sigmoide logística e a técnica de inicialização de peso comumente utilizada (distribuição normal de 0 e desvio-padrão 1) levava a problemas no treinamento de redes profundas [<https://homl.info/47>].

Com essa função de ativação e esse esquema de inicialização, a variância das saídas de cada camada é bem maior que a variância de suas entradas. Ao avançar pela rede, a variância continua aumentando camada após camada, até que a função de ativação sature nas camadas superiores. Essa saturação ainda se agrava mais pelo fato de a função logística ter uma média de 0,5 e não 0 (como a *tanh*, que se comporta um tanto melhor)

Problemas de gradientes de fuga e explosão de gradientes

Certas funções de ativação, como a sigmoide, espremem uma grande parte do espaço de inputs em uma pequena região entre 0 e 1. Portanto, uma grande mudança no input da sigmoide causará uma pequena mudança no output. Logo, a derivada torna-se pequena.

Função está saturada se $\lim_{|v| \rightarrow \infty} |\nabla f(v)| = 0$

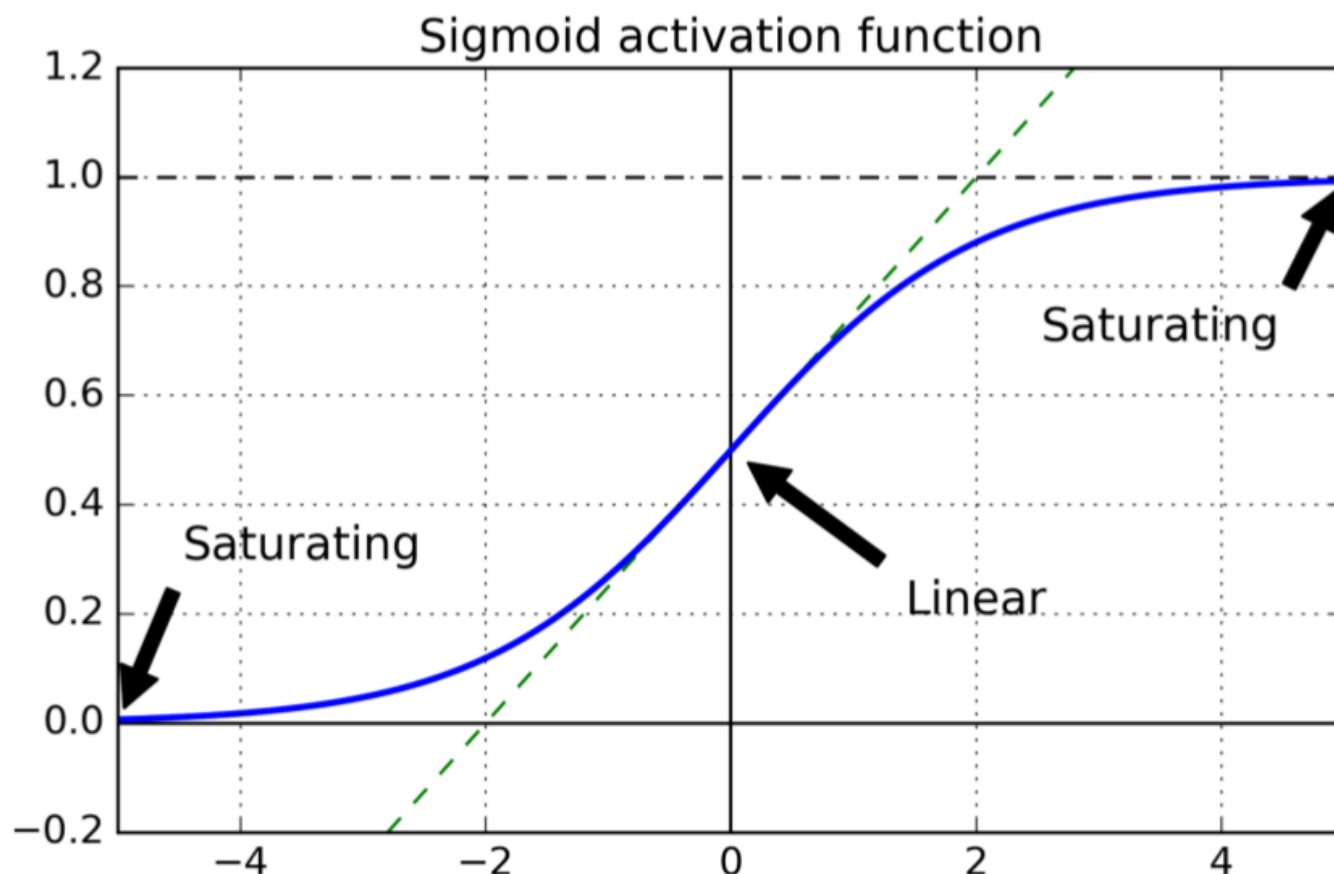
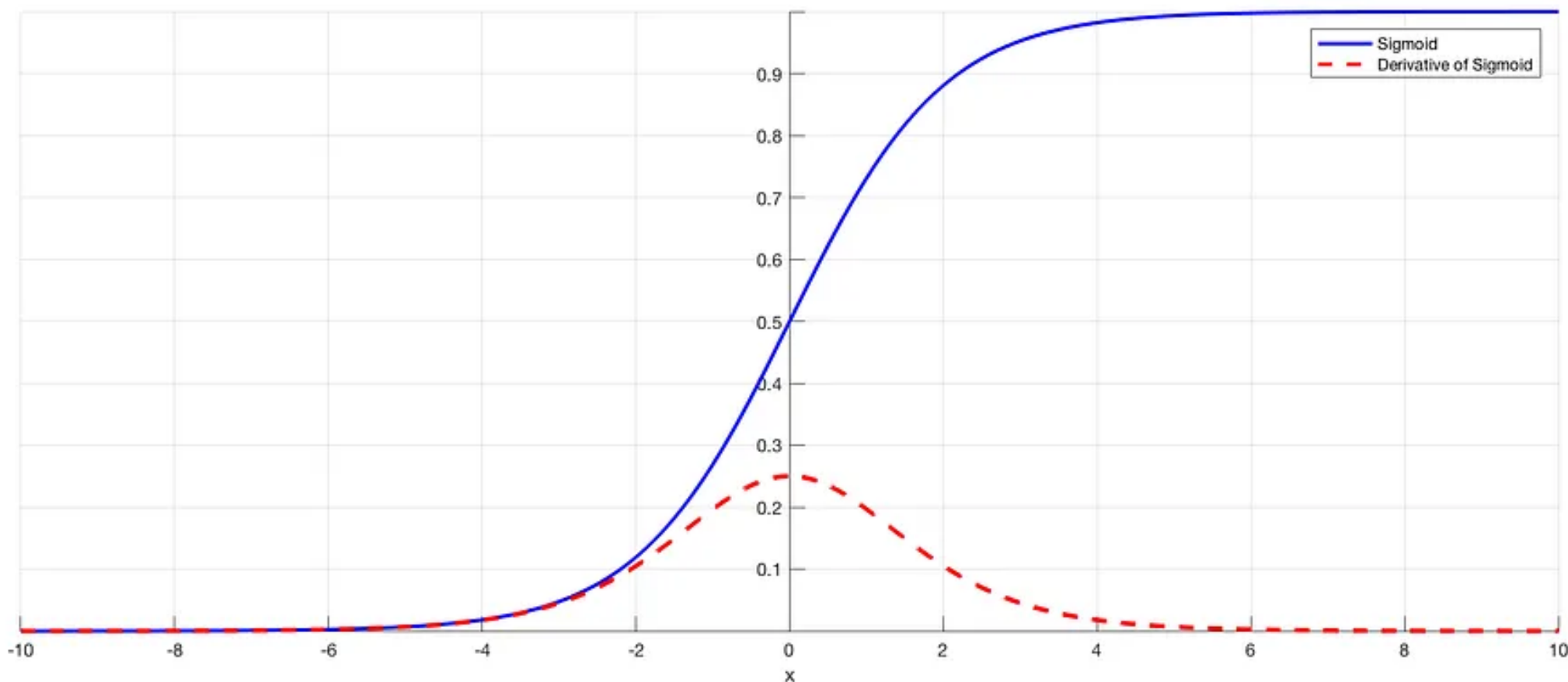


Figure 11-1. Logistic activation function saturation

Problemas de gradientes de fuga e explosão de gradientes

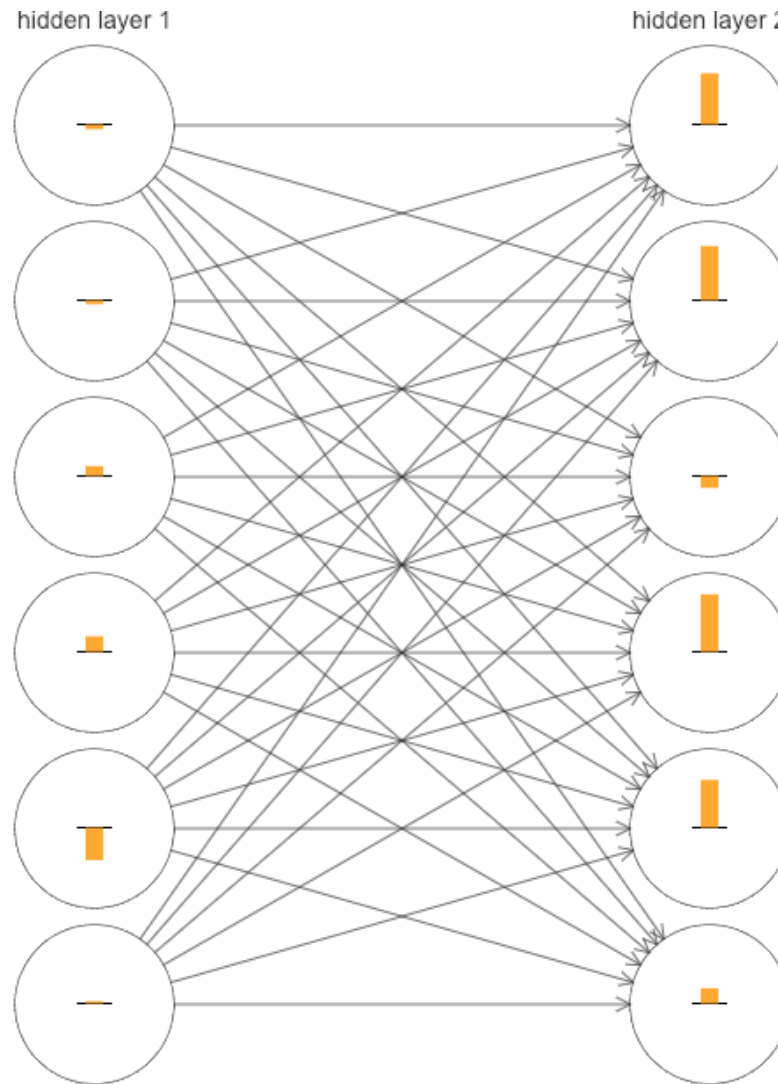
<https://towardsdatascience.com/the-vanishing-exploding-gradient-problem-in-deep-neural-networks-191358470c11>



Função achata os valores de x ; derivada zero quando $|x|$ é grande.

Problemas de gradientes de fuga e explosão de gradientes

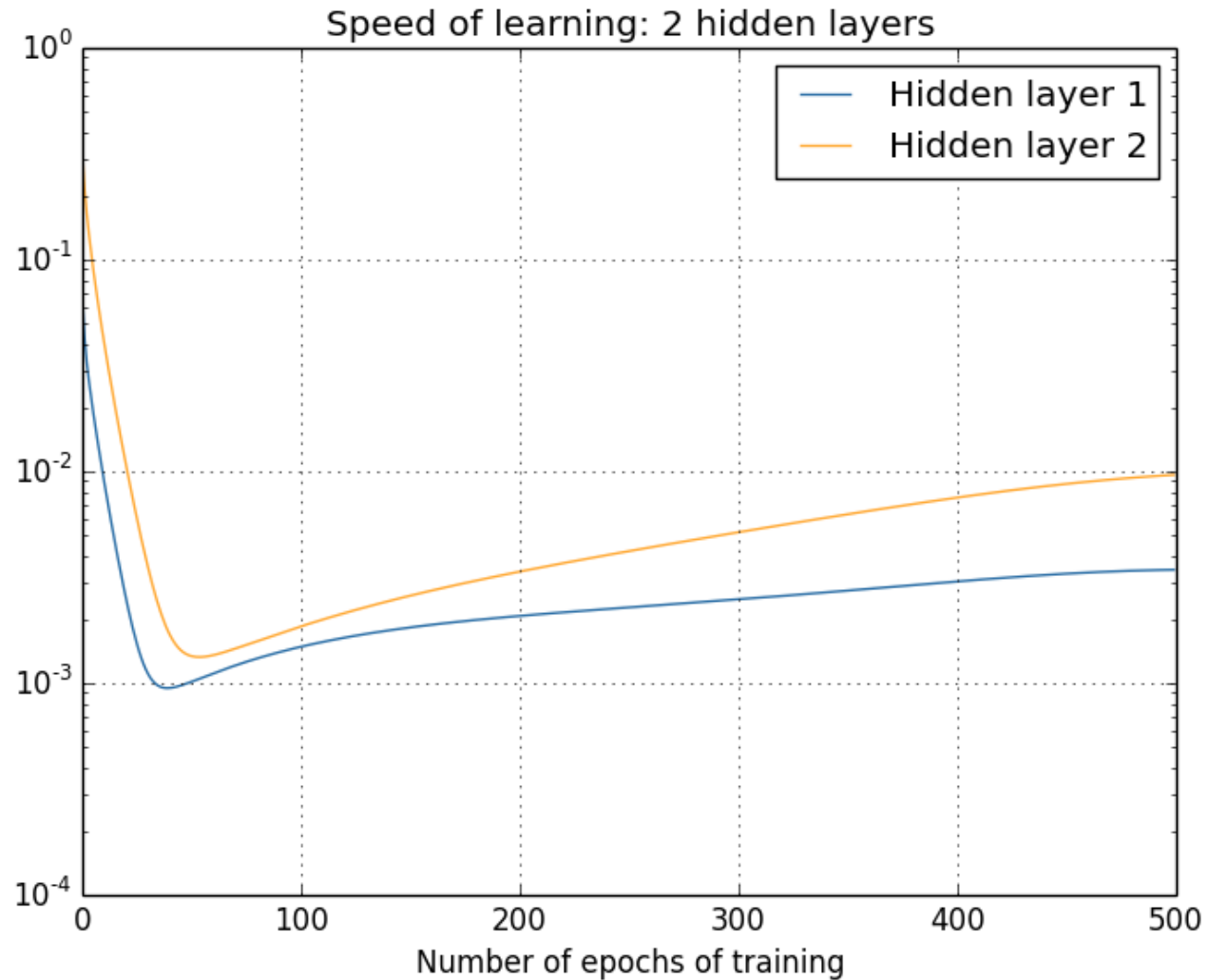
Derivada da função custo = “velocidade” de mudança dos pesos na rede.



Em redes neurais profundas, camadas inferiores mudam os pesos mais ‘lentamente’.

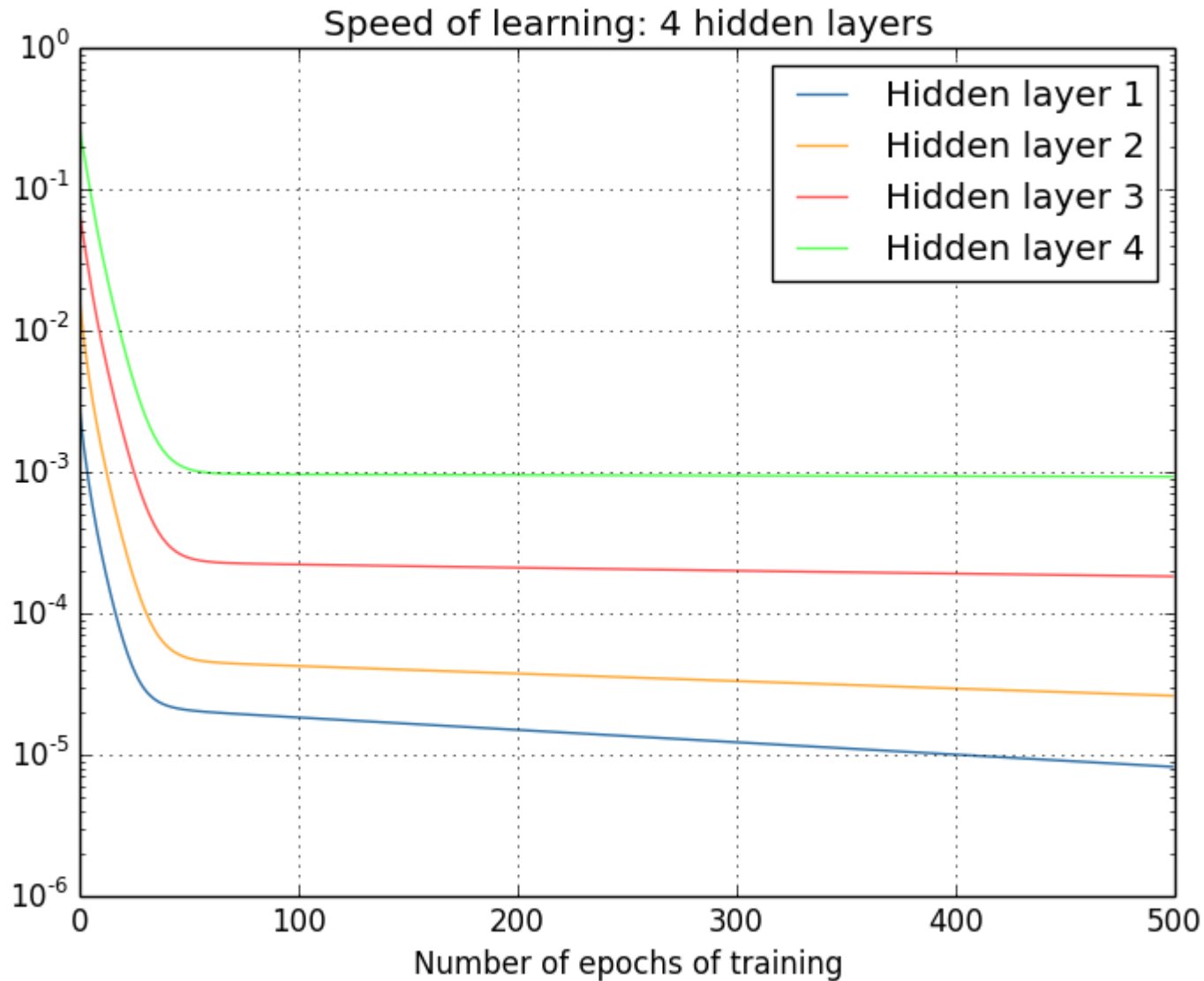
Problemas de gradientes de fuga e explosão de gradientes

Diferença na 'velocidade' de aprendizado (frequência de mudança dos pesos) em duas camadas:



Problemas de gradientes de fuga e explosão de gradientes

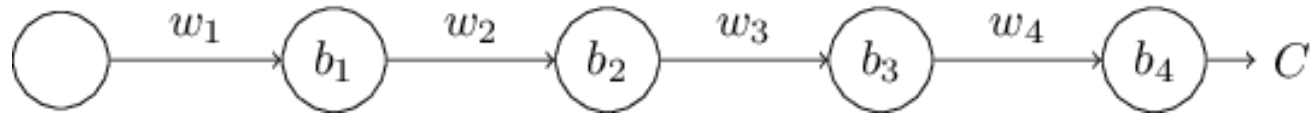
Diferença na 'velocidade' de aprendizado (frequência de mudança dos pesos) em quatro camadas:



Problemas de gradientes de fuga e explosão de gradientes

Exemplo: 3 camadas ocultas, apenas um neurônio por camada:

Cada neurônio com peso w e bias b ; custo = $C = a_4$.



Estimativa da derivada em relação ao primeiro peso: *Produto dos pesos \times ativações:*

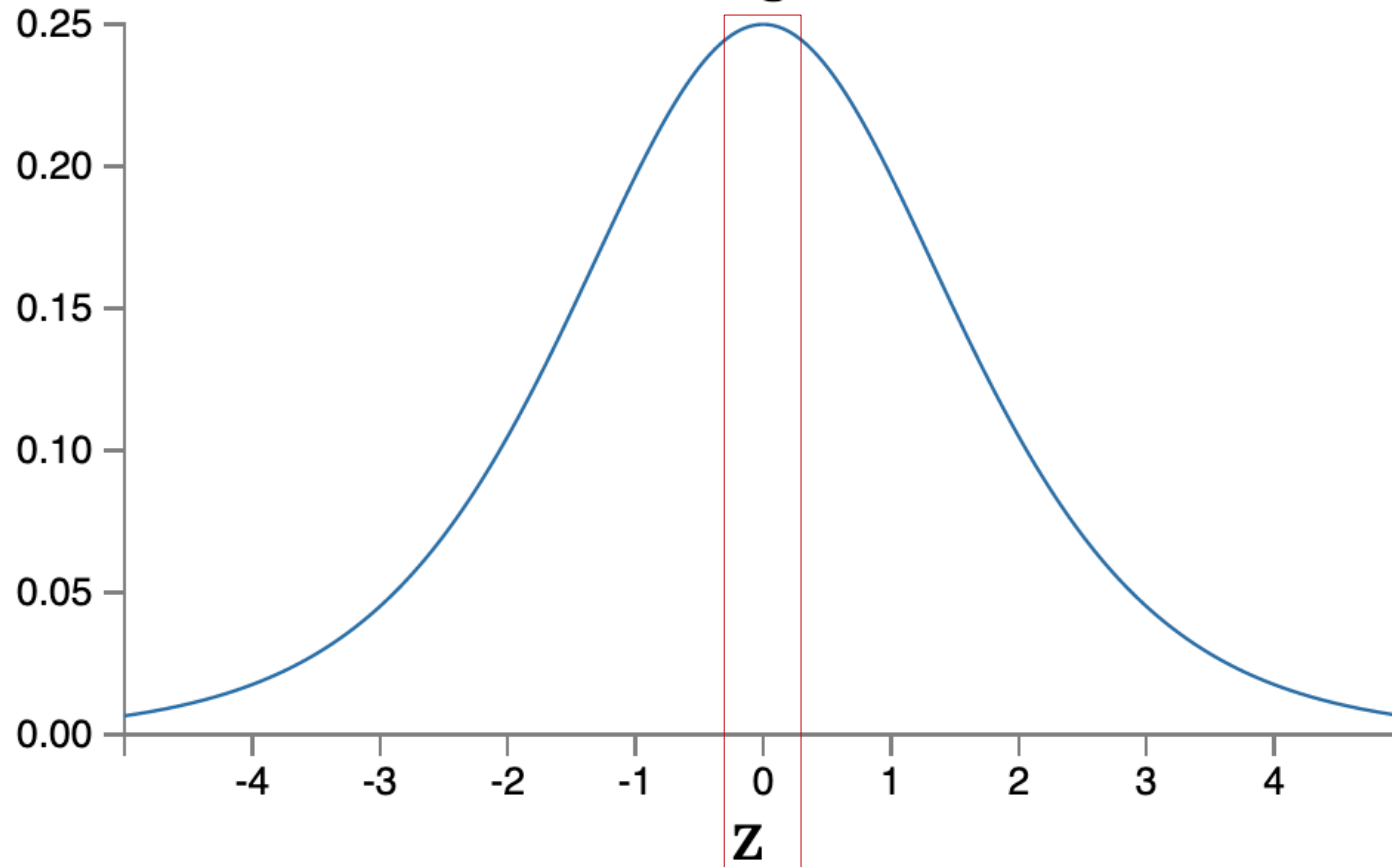
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



Pesos normalmente inicializados com média 0 e desvio-padrão 1 $\rightarrow |w_j| < 1$.

Problemas de gradientes de fuga e explosão de gradientes

Derivative of sigmoid function



$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \underbrace{w_2 \sigma'(z_2)}_{< \frac{1}{4}} \underbrace{w_3 \sigma'(z_3)}_{< \frac{1}{4}} \underbrace{w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}}_{\text{common terms}}$$

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3) \underbrace{w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}}_{\text{common terms}}$$

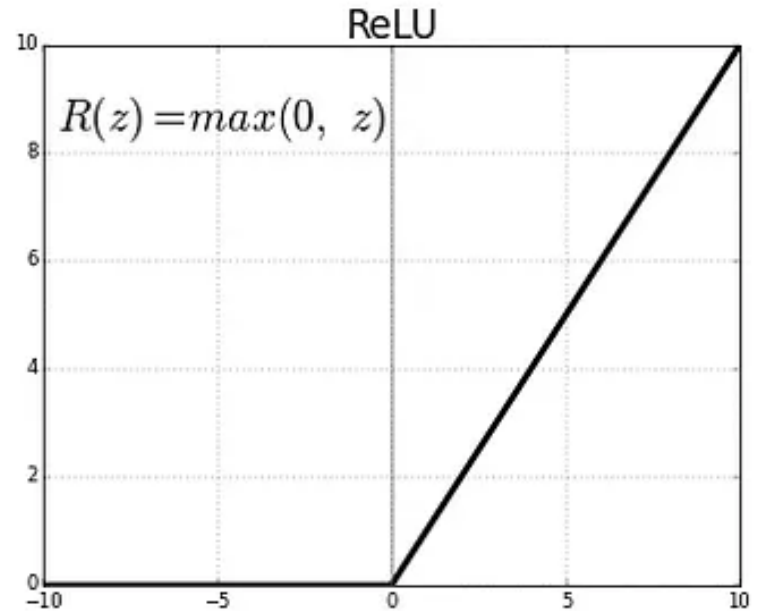
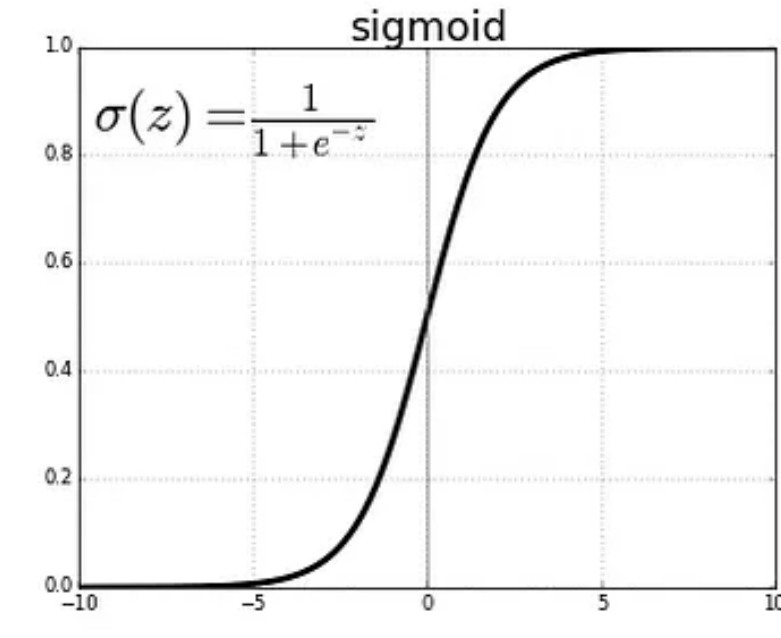
Derivada atinge um máximo quando $\sigma'(0) = \frac{1}{4}$.

Os produtos são menores que $\frac{1}{4}$...

Funções de ativação de não saturação

Função ReLU (Rectified linear unit):

Mais usada atualmente para redes convolucionais e redes profundas.



Ativação esparsa; computacionalmente eficiente; melhor propagação de gradiente.

Contradomínio de 0 a infinito.

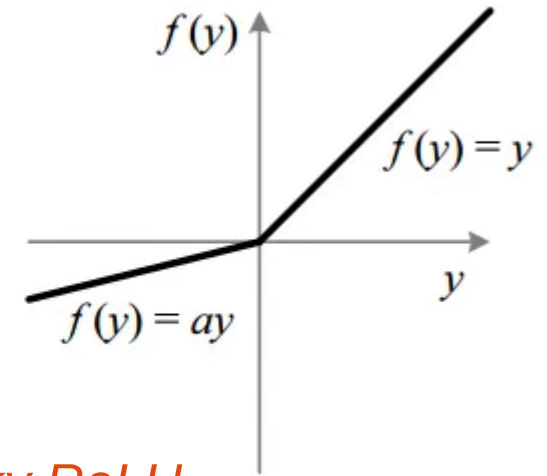
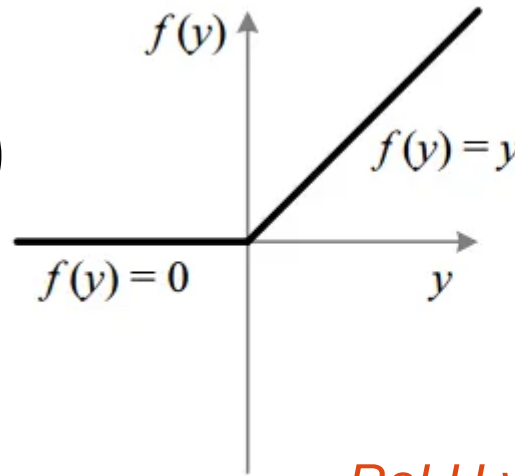
Dying ReLU... (especialmente para altos η)

Funções de ativação

$$\text{leaky ReLU}(z) = \max(\alpha z, z)$$

Comum: $\alpha = 0.01$.

Contradomínio de $-\infty$ a ∞



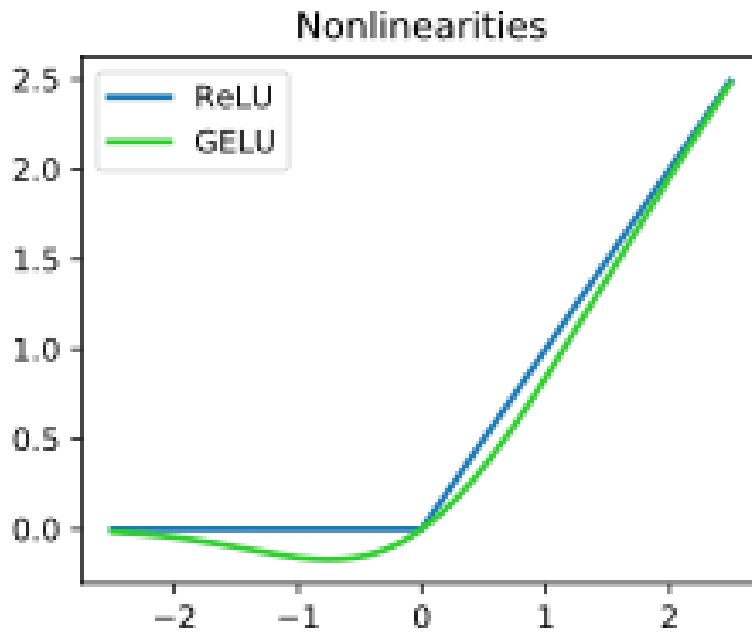
ReLU vs Leaky ReLU

```
Model = keras.models.Sequential([  
    [...]  
    Keras.layers.Dense(10),  
    Keras.layers.LeakyReLU(alpha=0.2),  
    [...]  
])
```

$$\text{ELU}_{\alpha} = \alpha (\exp(z) - 1), z < 1$$


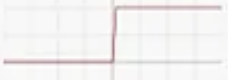

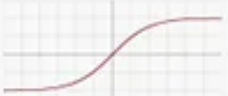
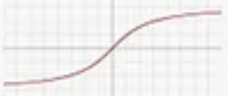



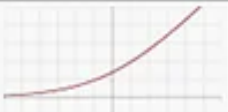
$$\text{ELU}_{\alpha} = z, z \geq 1$$

```
Model = keras.models.Sequential([  
    [...]  
    Keras.layers.Dense(10, activation="selu",  
        kernel_initializer="lecun_normal")  
    ....  
])
```



Funções de ativação de não saturação

SELU > ELU > leaky ReLU > ReLU > tanh > logística

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Inicialização de pesos em redes profundas

Prática comum: Pesos com média zero, desvio-padrão 1.

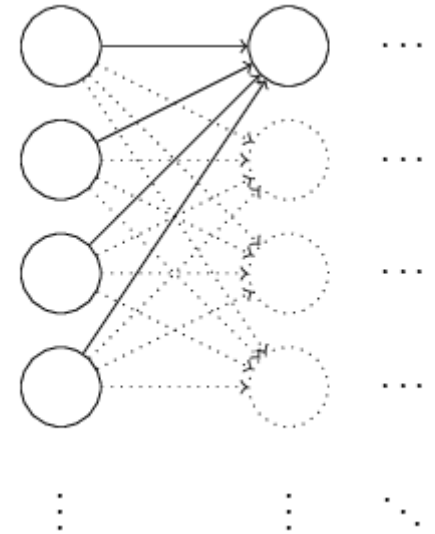
$$z = \sum_j w_j X_j$$

$$X \approx N(\mu_X, \sigma_X^2)$$

$$Y \approx N(\mu_Y, \sigma_Y^2)$$

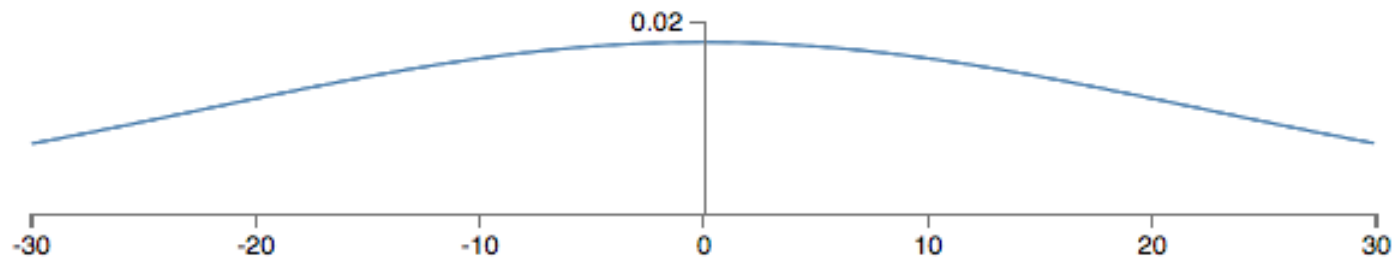
$$Z = X + Y$$

$$Z \approx N(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$$



Supondo 1000 neurônios, 500 ativados (=1), com bias: desvio-padrão da saída $\sqrt{(501)} \sim 22.4$:

Logo, a saída z também é uma gaussiana com desvio padrão bastante amplo (Gaussiana achatada); provavelmente $|z|$ alto; como os outputs são 0 ou 1, a derivada/gradiente é baixo!

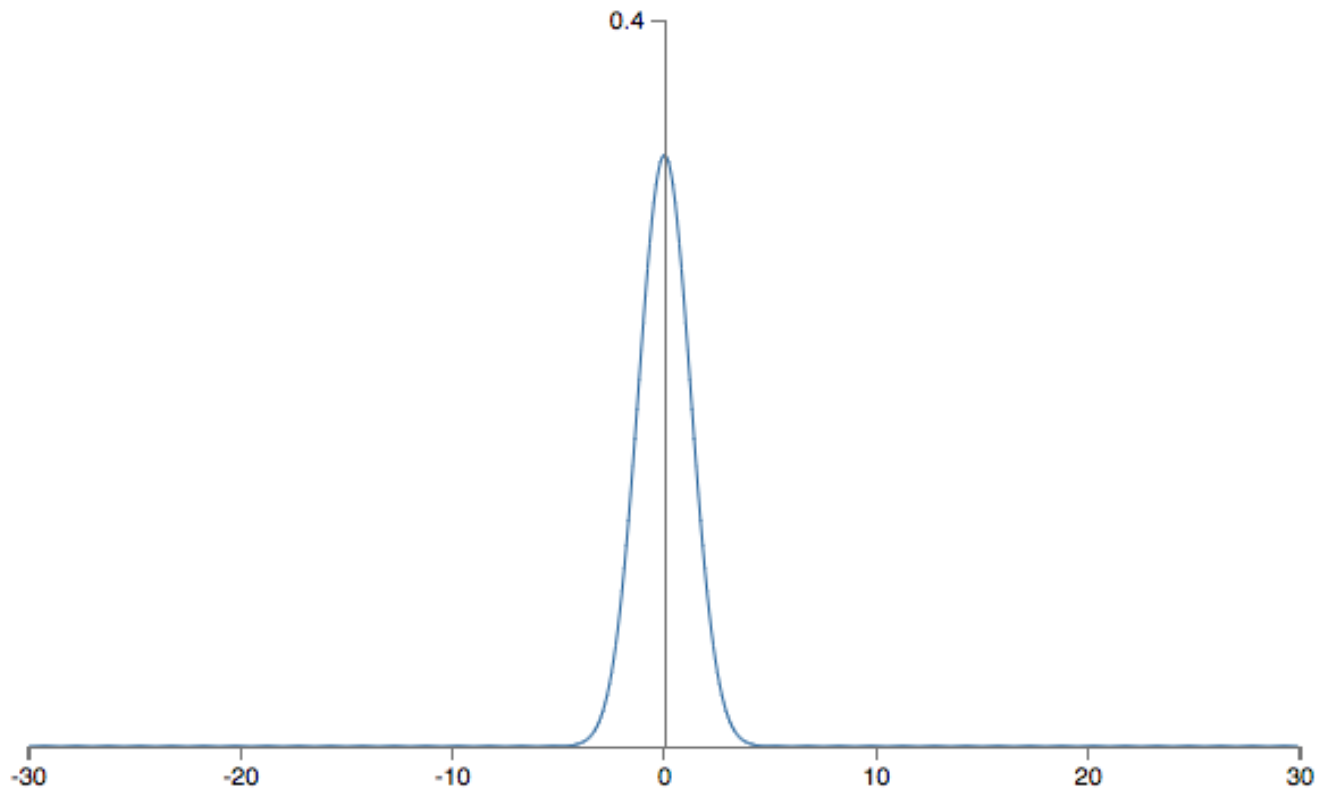


Alternativa: Iniciar os pesos com desvio padrão $\sqrt{(n_{in})}$, sendo n_{in} os neurônios de pesos de entrada.

Inicialização de pesos em redes profundas

Alternativa: Pesos com média zero, desvio-padrão $\sqrt{n_{in}}$.

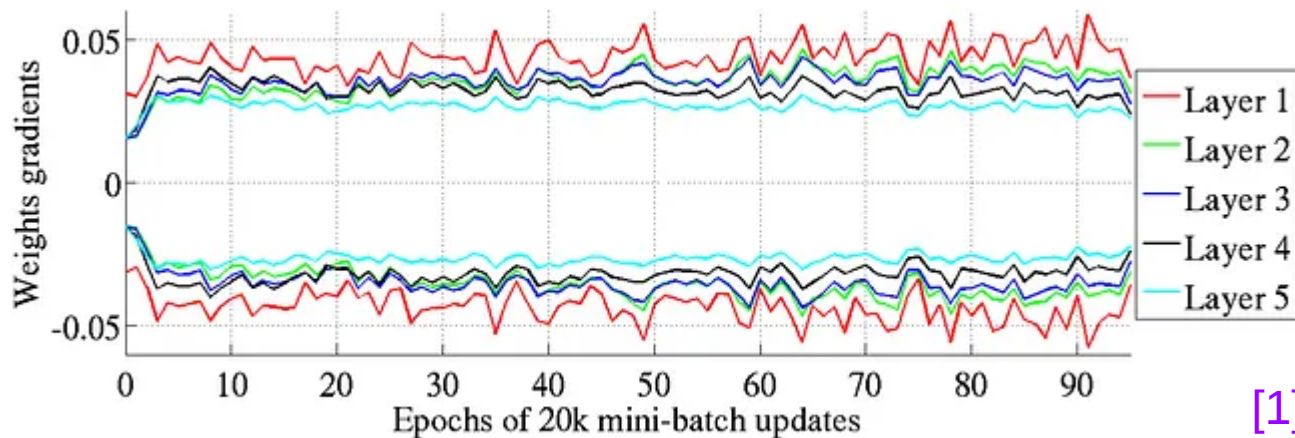
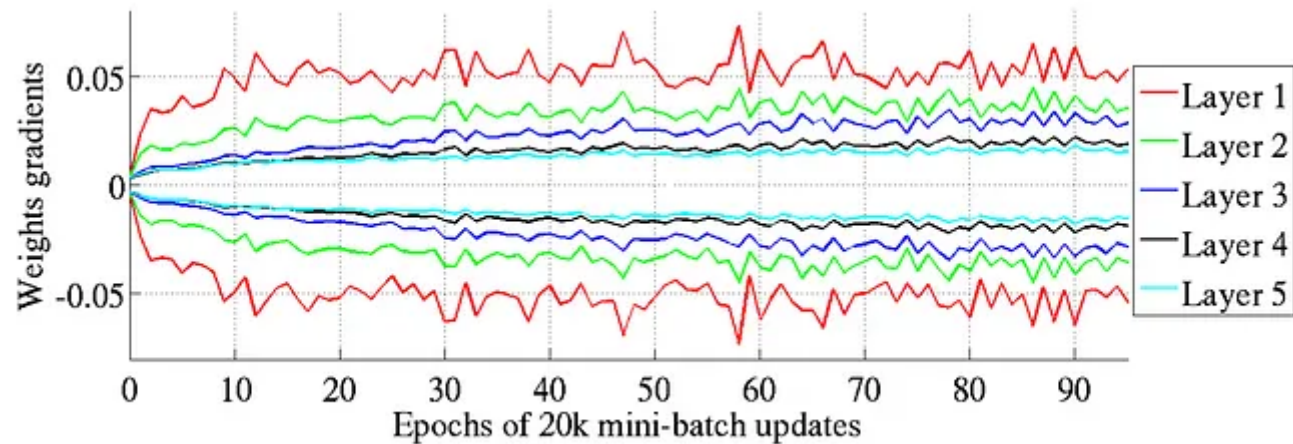
Supondo 1000 neurônios, 500 ativados ($=1$), com bias: desvio padrão da saída $\sqrt{3/2}$:
Agora a saída corresponde a uma Gaussiana com um desvio padrão bem menor.



Inicialização de pesos em redes profundas

Inicialização de Glorot ou de Xavier (*padrão no *Keras*)

Distribuição uniforme entre $-r$ e $+r$ com: $r = \pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$



[1]

Acima, os gradientes utilizando a inicialização normal; abaixo, com a inicialização de Glorot.

Inicialização de pesos em redes profundas

Exemplo em:

<https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79>

Inicialização de He ou de Kaiming

`Keras.layers.Dense(10, activation='relu', kernel_initializer='he_normal')`

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

[1]

$fan_{in} = n_{in}$ = número de entradas; $fan_{out} = n_{i+1}$ = número de saídas do neurônio.

Dying gradients – Normalização em batch (BN)

Embora o uso da inicialização He junto com a ELU (ou qualquer variante da ReLU) possa reduzir significativamente o risco de problemas com gradiente no início do treinamento, isso não garante que eles não retornem *durante* o treinamento das redes profundas.

A técnica BN consiste em adicionar uma operação no modelo logo antes ou depois da função de ativação de cada camada oculta.

Essa operação simplesmente centraliza em zero e normaliza cada entrada, em seguida escalona e modifica o resultado usando dois novos vetores de parâmetros por camada: um para o escalonamento e outro para o deslocamento.

[<https://homl.info/51>]

Estimativa da média e do desvio-padrão da entrada é realizada em relação ao mini-batch atual B :

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)}$$

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2$$

$$z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta$$

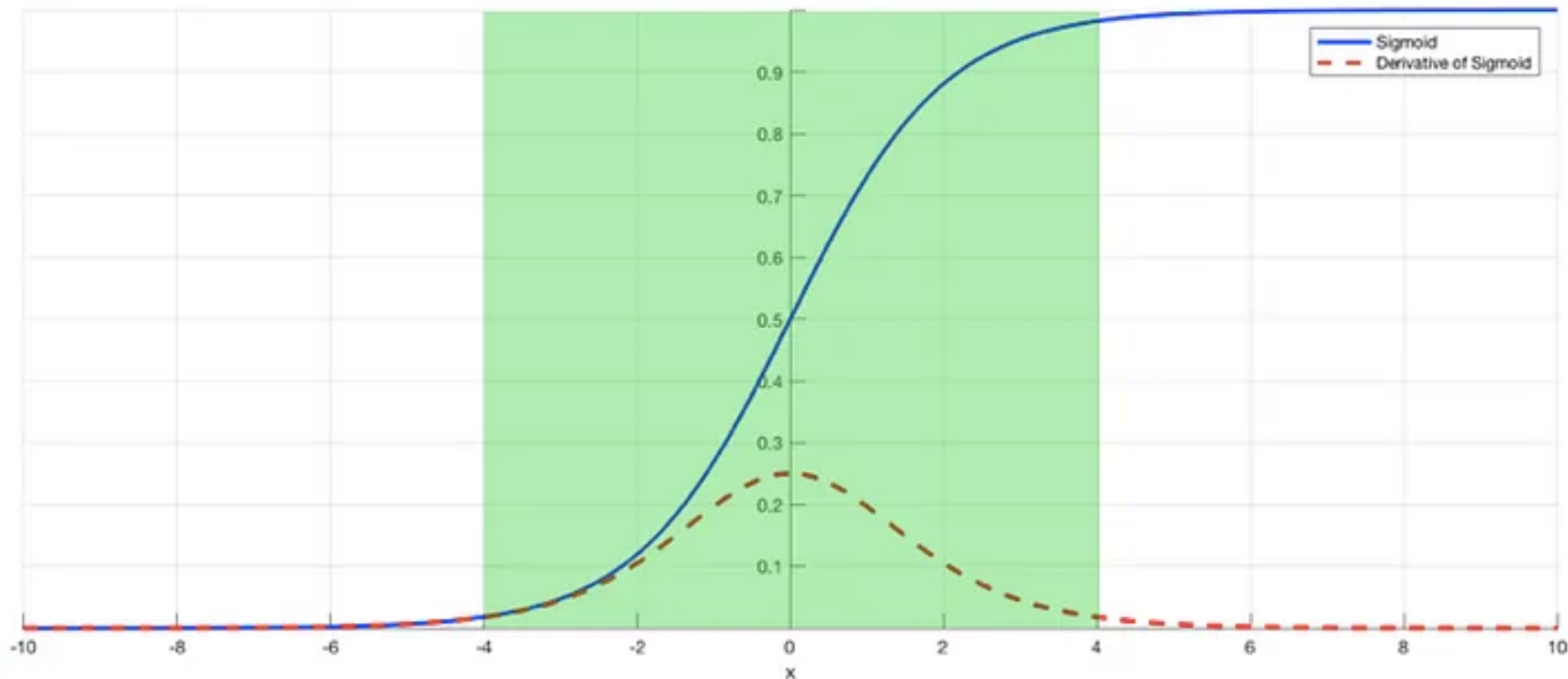
```
Model = keras.models.Sequential([  
    Keras.layers.Flatten(input_shape=[28,28]),  
    Keras.layers.BatchNormalization(),  
    Keras.layers.Dense(300, activation='elu'),  
    Keras.layers.BatchNormalization(),  
    [ ... ]  
])
```

Dying gradients – Normalização em batch (BN)

[\[https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739\]](https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739)

Parâmetros γ e β estimados durante o treinamento, e μ e σ ao final do treinamento; cada camada BN adiciona quatro parâmetros por entrada em cada camada.

Normalização do input $|x|$



Dying gradients – Clipping do gradiente

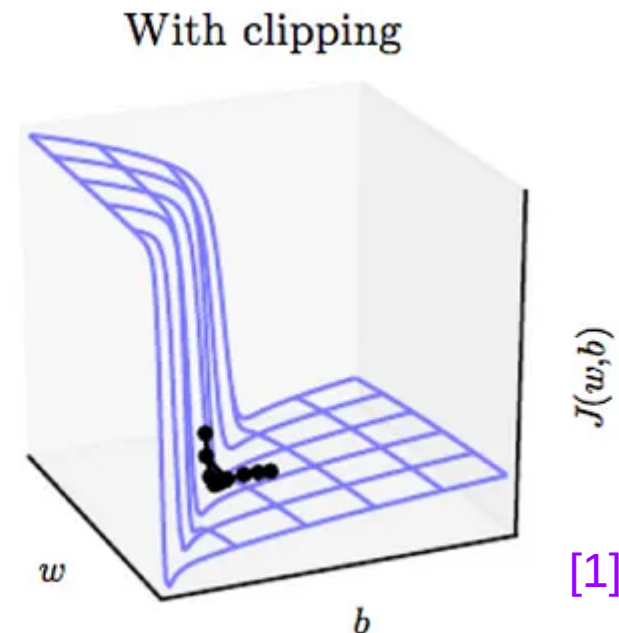
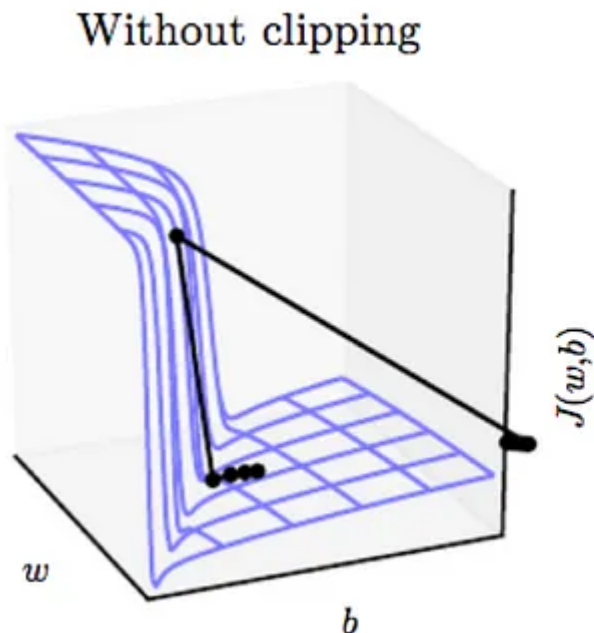
Garante que o vetor gradiente não tenha norma maior que c : $\mathbf{g} \leftarrow c \frac{\mathbf{g}}{\|\mathbf{g}\|}$

Técnica utilizada para 'recortar' os gradientes durante a retropropagação, de modo que eles nunca excedam algum limiar.

Usada mais em redes neurais recorrentes, onde a normalização em batch é mais difícil.

$[0.9, 1000] \rightarrow \text{clipvalue} \rightarrow [0.9, 1.0]$
 $[0.9, 1000] \rightarrow \text{clipnorm} \rightarrow [0.00899964, 0.9999595]$

```
Optimizer = keras.optimizers.SGD(clipvalue=1.0)  
Model.compile(loss='mse', optimizer=optimizer)
```



Aperfeiçoamento de hiperparâmetros das redes

O *hiperparâmetro* é um parâmetro de um algoritmo de aprendizado (não do modelo). Como tal, não é afetado pelo próprio algoritmo de aprendizado; deve ser definido antes do treinamento e permanecer constante ao longo dele.

Escolha “automática” de hiperparâmetros, através de algoritmos evolutivos:
[<https://homl.info/automlpost>]

– *Número de camadas ocultas*

Os dados do mundo real costumam ser estruturados de forma hierárquica e as redes profundas usam esse fato em seu proveito: as camadas ocultas inferiores modelam as estruturas de baixo nível; as camadas ocultas intermediárias combinam essas estruturas de baixo nível para modelar as estruturas de nível intermediário; e, por sua vez, as camadas ocultas superiores e a camada de saída combinam essas estruturas intermediárias para modelar estruturas de alto nível.

Essa arquitetura hierárquica não apenas ajuda as DNNs a convergirem mais rápido para uma boa solução, como também melhora a capacidade de generalização para conjuntos de dados novos.

Aperfeiçoamento de hiperparâmetros das redes

– *Número de neurônios por camada oculta*

Normalmente, é mais simples e eficiente escolher um modelo com mais camadas e neurônios do que você de fato precisa e, em seguida, usar técnicas de ajuste para impedir que o modelo se sobreajuste.

Em geral, você tem uma relação custo-benefício maior aumentando o número de camadas em vez do número de neurônios por camada.

– *Tamanho do batch*

Tamanho de batch preferível de até 32 instâncias [<https://homl.info/smallbatch>]; com outras técnicas mais refinadas, é possível treinar redes profundas com maiores valores de batch [<https://homl.info/largebatch>; <https://homl.info/largebatch2>]

Escolha de hiperparâmetros: Taxa de aprendizado, η

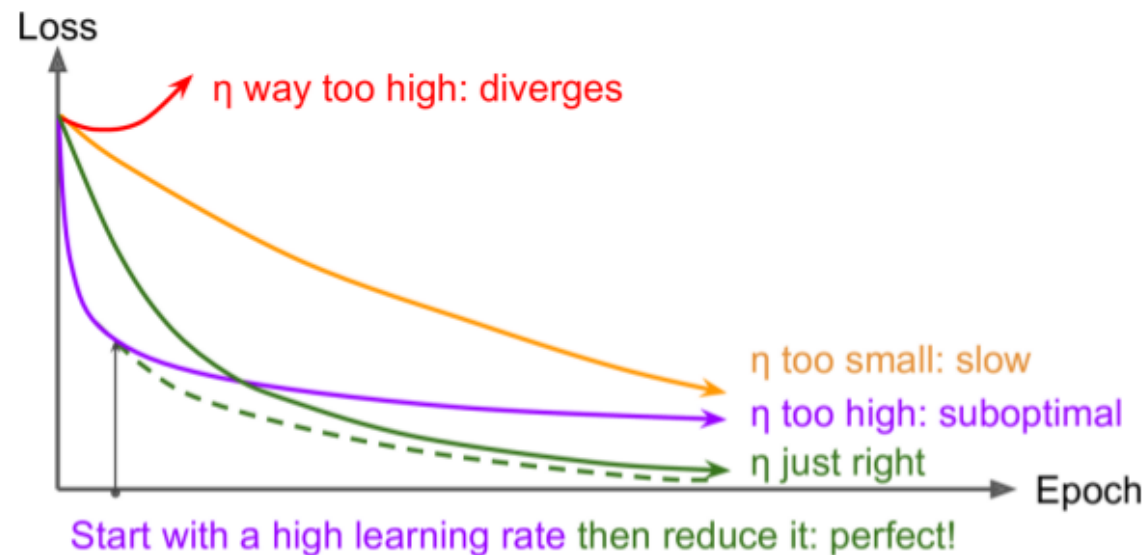


Figure 11-8. Learning curves for various learning rates η

Convergência: Função custo diminuir menos de 10^{-3} em uma iteração:
..., 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, ...

Cronogramas de aprendizado: *Power scheduling*

Optimizer = keras.optimizer.SGD(lr=0.01, decay=1e-4)

$$\eta(t) = \frac{\eta_0}{(1+t/s)^c}$$

Decay = 1/s; Keras: c=1.

Estratégia geral: [<https://www.deeplearningbook.com.br/capitulo-26-como-escolher-os-hiperparametros-de-uma-rede-neural/>]

Otimizadores mais rápidos

Otimizadores mais rápidos são necessários para treinamento de redes profundas: *Otimização momentum, gradiente acelerado de Nesterov, AdaGrad, RMSProp e otimização Adam e Nadam.*

Gradiente descendente: $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$

Atualização dos pesos depende do valor local (atual) do gradiente; caso esteja em uma região plana no hiperespaço de parâmetros, variação será pequena.

Otimizadores mais rápidos

– Otimização momentum

Na otimização momentum, o gradiente ganhará “momento” à medida que desce em direção a um mínimo no hiperespaço de parâmetros [<https://homl.info/54>].

A otimização momentum se importa bastante com os gradientes anteriores: a cada iteração, subtrai o gradiente local do vetor *momentum* \mathbf{m} e atualiza os pesos somando esse vetor *momentum*; o gradiente é usado para aceleração, não velocidade.

$$\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta + \mathbf{m}$$

A otimização momentum é importante para redes profundas em que não utilizam normalizam em batch, pois entradas superiores acabam tendo entradas com escalas muito diferentes. Esta otimização também ajuda a passar por mínimos locais.

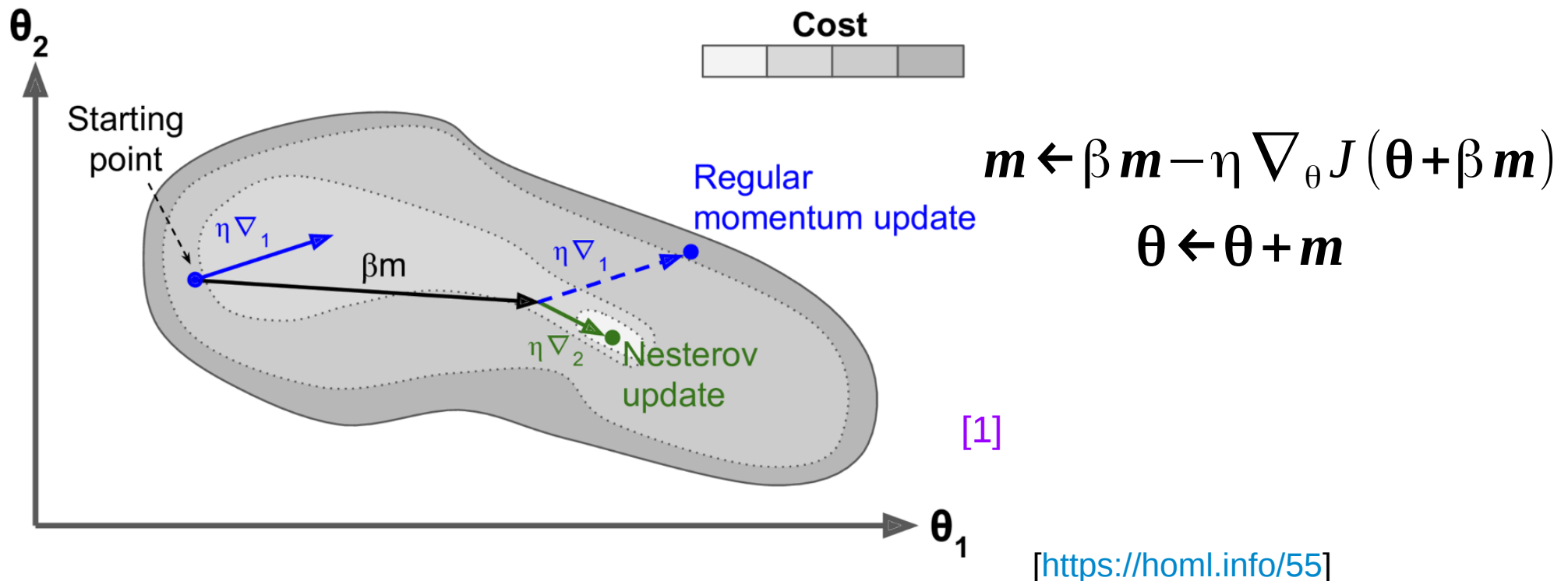
```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

<https://www.geeksforgeeks.org/ml-momentum-based-gradient-optimizer-introduction/>

Otimizadores mais rápidos

– Gradiente acelerado de Nesterov (NAG)

É uma pequena variação da otimização momentum. Utiliza o valor do gradiente não na posição local theta, mas um pouco à frente na direção do momentum, $\theta + \beta m$:



É mais rápido, possui menos oscilações durante a otimização.

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

Otimizadores mais rápidos

– AdaGrad

No AdaGrad, a taxa de aprendizado decai, porém é mais rápido em dimensões acentuadas do que em dimensões com declives mais suaves; a taxa de aprendizado é *adaptativa*.

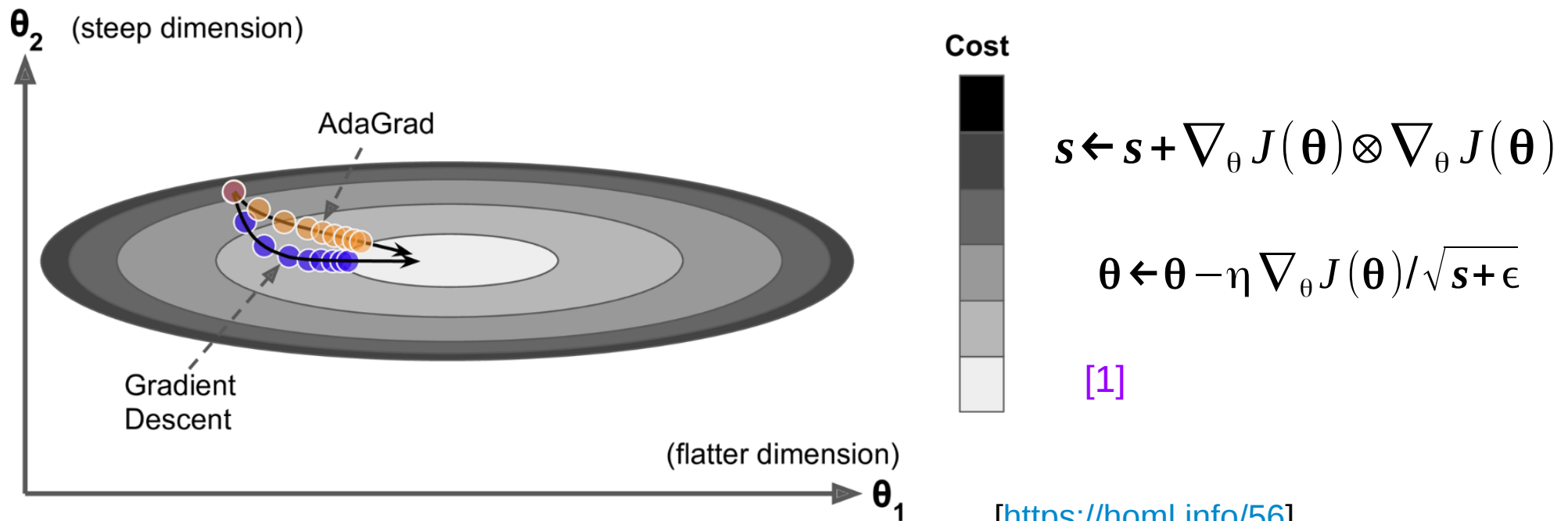


Figure 11-7. AdaGrad versus Gradient Descent: the former can correct its direction earlier to point to the optimum

Possui bom desempenho em problemas quadráticos simples, no entanto costuma parar muito cedo ao treinar redes neurais. A taxa de aprendizado é reduzida gradativamente, tanto que o algoritmo acaba parando completamente antes de atingir o mínimo global.

Otimizadores mais rápidos

– *RMSProp*

O algoritmo RMSProp corrige o AdaGrad acumulando somente os gradientes das iterações mais recentes (em vez de todos os gradientes desde o início do treinamento). Isso é feito usando decaimento exponencial na primeira etapa.

[<https://homl.info/57>; <https://homl.info/58>]

$$\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\boldsymbol{\theta}) \otimes \nabla_{\theta} J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\theta} J(\boldsymbol{\theta}) / \sqrt{\mathbf{s} + \epsilon}$$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Otimizadores mais rápidos

– Otimização Adam e Nadam

Adam, *adaptive moment estimation*, combina as ideias da otimização momentum e RMSProp: assim como a otimização momentum, registra uma média de decaimento exponencial dos gradientes anteriores; e, como o RMSProp, registra uma média de decaimento exponencial dos gradientes quadrados anteriores.

[<https://homl.info/59>]

$$1. \quad \mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$2. \quad \mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$3. \quad \hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T}$$

$$4. \quad \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^T}$$

*Keras: ϵ padrão: 10^{-7}

$$5. \quad \theta \leftarrow \theta + \eta \hat{\mathbf{m}} / \sqrt{\hat{\mathbf{s}} + \epsilon}$$

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

**Nadam*: Usa o truque de Nesterov [<https://homl.info/nadam>]

Otimizadores mais rápidos

Escolha padrão para redes profundas: Métodos de aprendizado adaptativo (*Adam*, *Nadam*); caso falhem, *NAG* [<https://hml.info/60>].

Table 11-2 compares all the optimizers we've discussed so far (* is bad, ** is average, and *** is good).

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

[1]

Who's Adam and What's He Optimizing?
<https://www.youtube.com/watch?v=MD2fYip6QsQ>

Quantidade de dados rotulados: Aprendizado por transferência

Também aplicável para pré-treinar redes profundas!

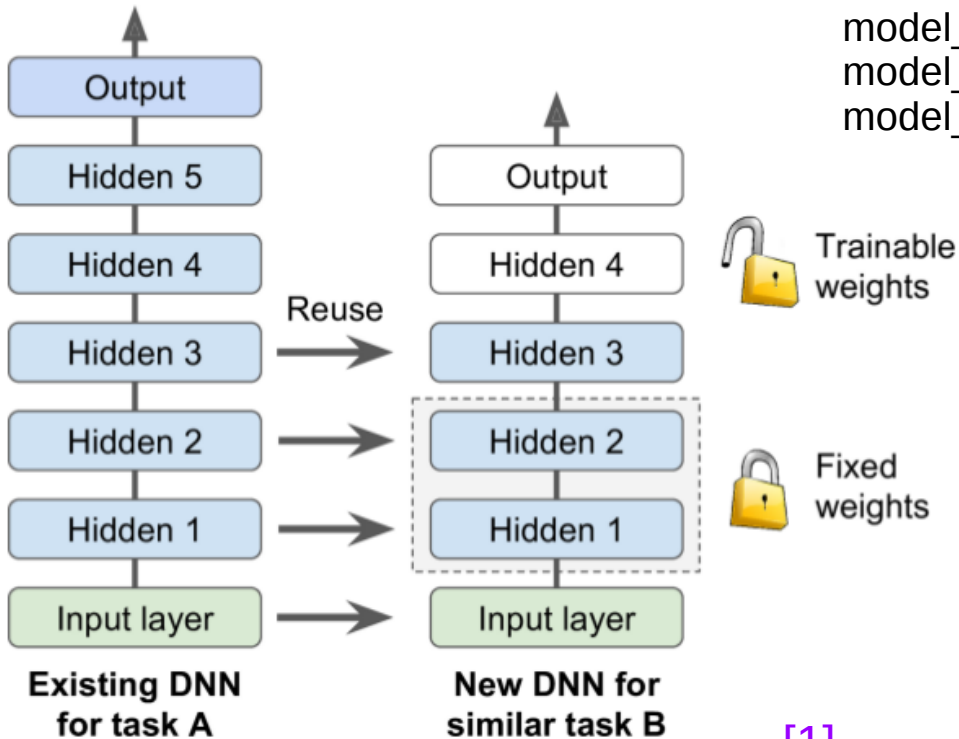


Figure 11-4. Reusing pretrained layers

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

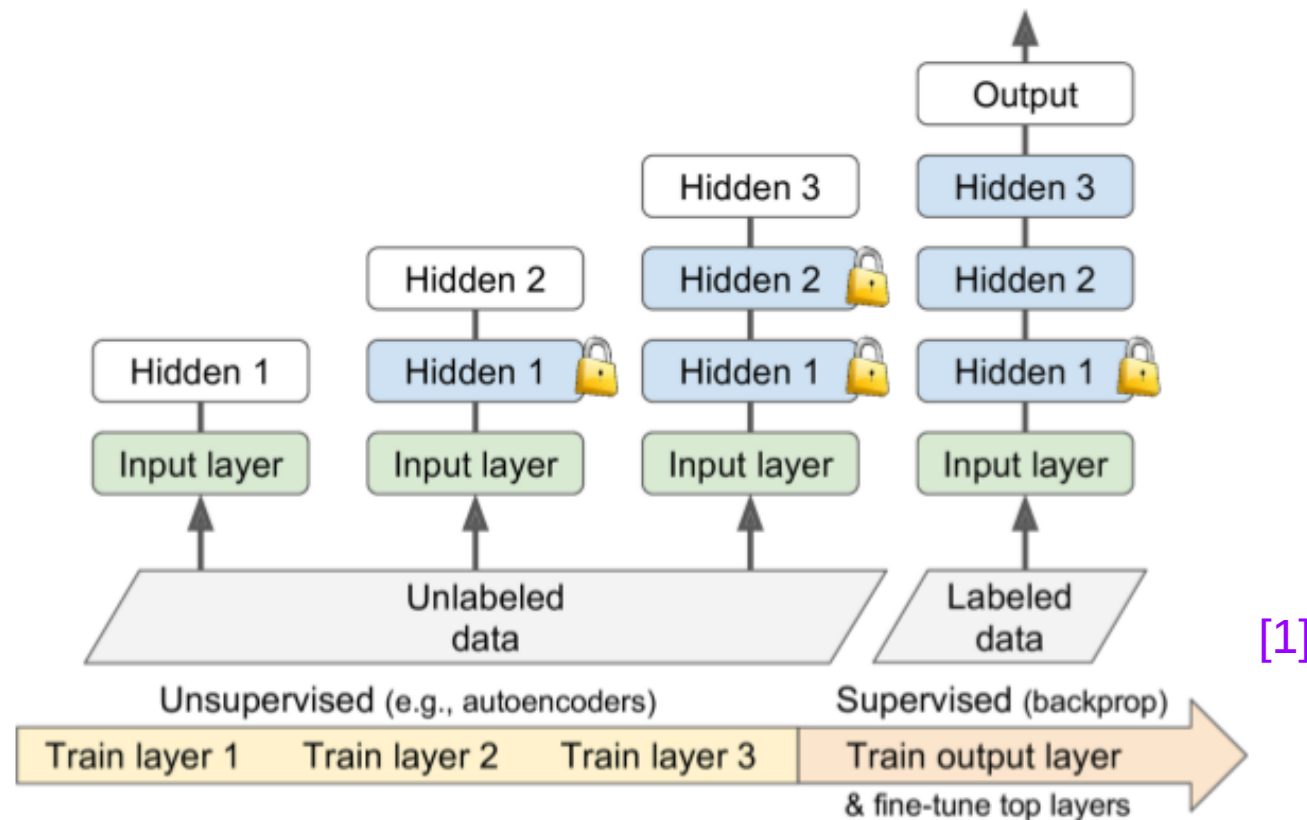
[1]

```
For layer in model_B_on_A.layers[:-1]:
    Layer.trainable = False
```

```
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd", metrics=['accuracy'])
```

https://keras.io/guides/transfer_learning/

Quantidade de dados rotulados: Pré-treinamento não-supervisionado



Exemplo da técnica *greedy layer-wise pretraining*. No treinamento não-supervisionado, um modelo é treinado nos dados não rotulados (ou em todos os dados) usando uma técnica de aprendizado não supervisionado, e depois é ajustado para a tarefa final nos dados rotulados utilizando uma técnica de aprendizado supervisionado; a parte não supervisionada pode treinar uma camada por vez, ou pode treinar o modelo completo diretamente.

Diretrizes práticas

Table 11-2. Default DNN configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

[1]

- Normalizar as características de entrada!
- Se precisar de um modelo esparsos, usar a regularização l1;
- Caso precise de um modelo de baixa latência: Usar menos camadas, dobrar as camadas de normalização em batch e usar uma função de ativação rápida, como a leaky ELU ou ReLU.