Entrenamiento de Redes Neuronales

Luis Norberto Zúñiga Morales

15 de mayo de 2022

Contenido

Introducción

- La clase pasado entrenaron una red neuronal. ¡Ya se pueden comer el mundo! ¿Cierto? ¡¿Cierto?!
- Cuando entrenen redes neuronales más profundas, se toparán con problemas que requieren su comprensión.

Introducción

Algunos problemas que pueden surgir:

1. El problema del desvanecimiento gradiente o problemas con el gradiente explosivo.

Introducción

Algunos problemas que pueden surgir:

1. El problema del desvanecimiento gradiente o problemas con el gradiente explosivo.
2. ¿Qué pasa si no tienen suficiente información para entrenar una red?

Introducción

Algunos problemas que pueden surgir:

1. El problema del desvanecimiento gradiente o problemas con el gradiente explosivo.
2. ¿Qué pasa si no tienen suficiente información para entrenar una red?
3. El entrenamiento es lento.

Introducción

Algunos problemas que pueden surgir:

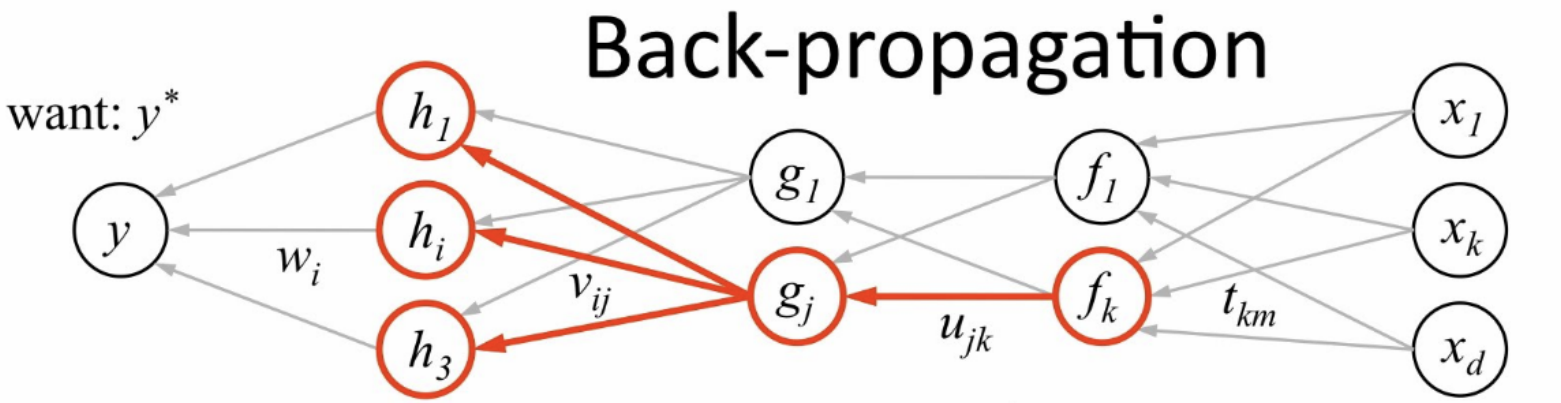
1. El problema del desvanecimiento gradiente o problemas con el gradiente explosivo.
2. ¿Qué pasa si no tienen suficiente información para entrenar una red?
3. El entrenamiento es lento.
4. Una red neuronal fácilmente puede tener millones de parámetros. ¿Cómo evitar el sobreajuste?

Problema: Desvanecimiento del Gradiente

Durante el entrenamiento o ajuste de los parámetros de una red neuronal utilizan backpropagation con gradiente descendiente.

¿Notan algún problema?

Problema: Desvanecimiento del Gradiente



1. receive new observation $\boldsymbol{x} = [x_1 \ldots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer 1…L
   compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y\text{-}y^*)$
4. **back-propagate error:** for each unit g_j in each layer L…1

(a) compute error on g_j

$$\frac{\partial E}{\partial g_j} = \sum_i \sigma'(h_i) v_{ij} \frac{\partial E}{\partial h_i}$$

- $\frac{\partial E}{\partial g_j}$: should g_j be higher or lower?
- $\sigma'(h_i) v_{ij}$: how h_i will change as g_j changes
- $\frac{\partial E}{\partial h_i}$: was h_i too high or too low?

(b) for each u_{jk} that affects g_j

(i) compute error on u_{jk}

$$\frac{\partial E}{\partial u_{jk}} = \frac{\partial E}{\partial g_j} \sigma'(g_j) f_k$$

- $\frac{\partial E}{\partial g_j}$: do we want g_j to be higher/lower
- $\sigma'(g_j) f_k$: how g_j will change if u_{jk} is higher/lower

(ii) update the weight

$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

Problema: Desvanecimiento del Gradiente

- Una vez que el algoritmo terminó de calcular el gradiente de la función de costo para cada parámetro en la red...

Problema: Desvanecimiento del Gradiente

- Una vez que el algoritmo terminó de calcular el gradiente de la función de costo para cada parámetro en la red...
- Utiliza estos gradientes para actualizar cada parámetro con un paso de gradiente descendiente.

Problema: Desvanecimiento del Gradiente

- Una vez que el algoritmo terminó de calcular el gradiente de la función de costo para cada parámetro en la red...
- Utiliza estos gradientes para actualizar cada parámetro con un paso de gradiente descendiente.
- ¿Lo malo? El gradiente se hace más y más pequeño conforme la retropropagación llega a las capas más profundas.

Problema: Desvanecimiento del Gradiente

- Una vez que el algoritmo terminó de calcular el gradiente de la función de costo para cada parámetro en la red...
- Utiliza estos gradientes para actualizar cada parámetro con un paso de gradiente descendiente.
- ¿Lo malo? El gradiente se hace más y más pequeño conforme la retropropagación llega a las capas más profundas.
- Consecuencia: no se actualizan las conexiones de las capas más bajas.

Problema: Desvanecimiento del Gradiente

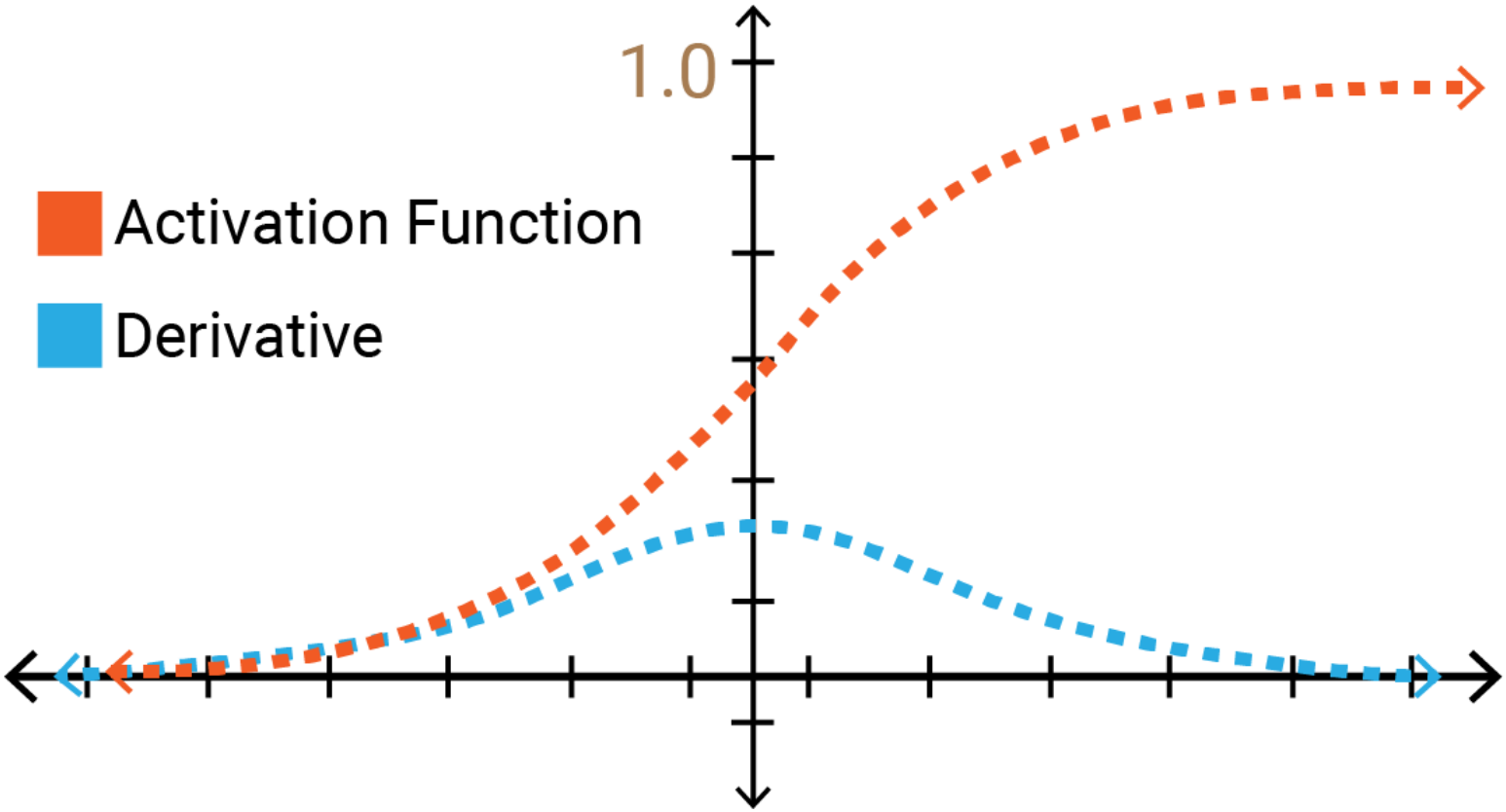
- Una vez que el algoritmo terminó de calcular el gradiente de la función de costo para cada parámetro en la red...
- Utiliza estos gradientes para actualizar cada parámetro con un paso de gradiente descendiente.
- ¿Lo malo? El gradiente se hace **más y más pequeño** conforme la retropropagación llega a las capas más profundas.
- Consecuencia: **no se actualizan las conexiones** de las capas más bajas.
- El entrenamiento **nunca converge** a una solución "buena".

Problema: Desvanecimiento del Gradiente

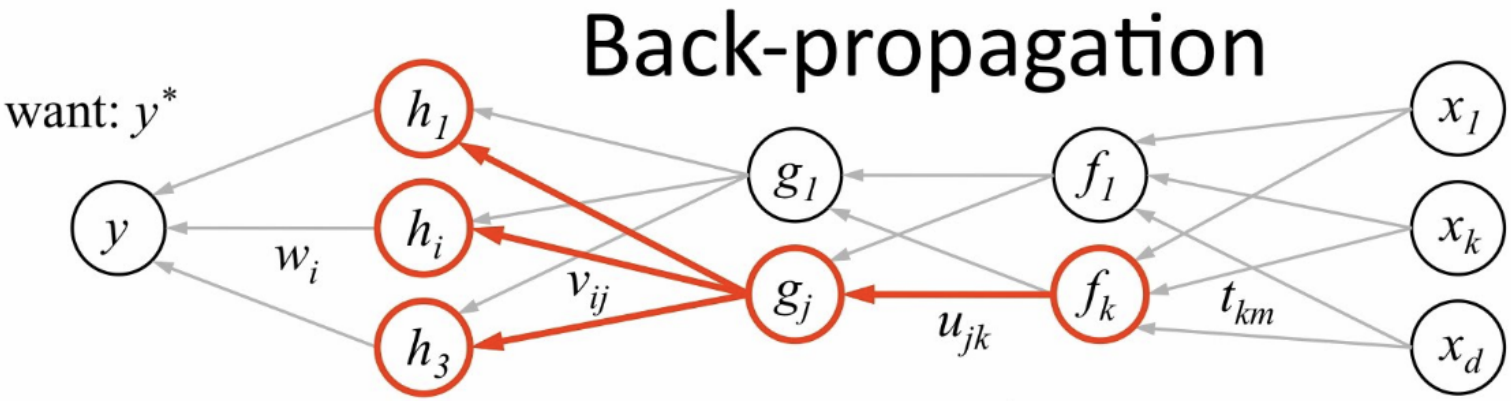
Ejercicio

Grafiquen la función de activación sigmoide y su derivada en el mismo canvas.

Problema: Desvanecimiento del Gradiente



Problema: Desvanecimiento del Gradiente



1. receive new observation $\boldsymbol{x} = [x_1 \ldots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer 1…L
   compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y\text{-}y^*)$
4. **back-propagate error:** for each unit g_j in each layer L…1

(a) compute error on g_j

$$\frac{\partial E}{\partial g_j} = \sum_i \sigma'(h_i) v_{ij} \frac{\partial E}{\partial h_i}$$

- $\frac{\partial E}{\partial g_j}$: should g_j be higher or lower?
- $\sigma'(h_i) v_{ij}$: how h_i will change as g_j changes
- $\frac{\partial E}{\partial h_i}$: was h_i too high or too low?

(b) for each u_{jk} that affects g_j

(i) compute error on u_{jk}

$$\frac{\partial E}{\partial u_{jk}} = \frac{\partial E}{\partial g_j} \sigma'(g_j) f_k$$

- $\frac{\partial E}{\partial g_j}$: do we want g_j to be higher/lower
- $\sigma'(g_j) f_k$: how g_j will change if u_{jk} is higher/lower

(ii) update the weight

$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

Problema: Desvanecimiento del Gradiente

- Este problema generó otro éxodo a inicios de los 2000s, ya que no fue claro qué provocaba el desvanecimiento.
- En el 2010, Xavier Glorot y Yoshua Bengio [2] encontraron ciertos elementos sospechosos: funciones de activación (sigmoide) y la inicialización de pesos (N(0,1)).

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

Abstract

Whereas before 2006 it appears that deep multi-layer neural networks were not successfully trained, since then several algorithms have been shown to successfully train them, with experimental results showing the superiority of deeper vs less deep architectures. All these experimental results were obtained with new initialization or training mechanisms. Our objective here is to understand better why standard gradient descent from random initialization is doing so poorly with deep neural networks, to better understand these recent relative successes and help design better algorithms in the future. We first observe the influence of the non-linear activations functions. We find that the logistic sigmoid activation is unsuited for deep networks with random initialization because of its mean value, which can drive especially the top hidden layer into saturation. Surprisingly, we find that saturated units can move out of saturation by themselves, albeit slowly, and explaining the plateaus sometimes seen when training neural networks. We find that a new non-linearity that saturates less can often be beneficial. Finally, we study how activations and gradients vary across layers and during training, with the idea that training may be more difficult when the singular values of the Jacobian associated with each layer are far from 1. Based on these considerations, we propose a new initialization scheme that brings substantially faster convergence.

1 Deep Neural Networks

Deep learning methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. They include learning methods for a wide array of *deep architectures*, including neural networks with many hidden layers (Vincent et al., 2008) and graphical models with many levels of hidden variables (Hinton et al., 2006), among others (Zhu et al., 2009; Weston et al., 2008). Much attention has recently been devoted to them (see (Bengio, 2009) for a review), because of their theoretical appeal, inspiration from biology and human cognition, and because of empirical success in vision (Ranzato et al., 2007; Larochelle et al., 2007; Vincent et al., 2008) and natural language processing (NLP) (Collobert & Weston, 2008; Mnih & Hinton, 2009). Theoretical results reviewed and discussed by Bengio (2009), suggest that in order to learn the kind of complicated functions that can represent high-level abstractions (e.g. in vision, language, and other AI-level tasks), one may need *deep architectures*.

Most of the recent experimental results with deep architecture are obtained with models that can be turned into deep supervised neural networks, but with initialization or training schemes different from the classical feedforward neural networks (Rumelhart et al., 1986). Why are these new algorithms working so much better than the standard random initialization and gradient-based optimization of a supervised training criterion? Part of the answer may be found in recent analyses of the effect of unsupervised pre-training (Erhan et al., 2009), showing that it acts as a regularizer that initializes the parameters in a "better" basin of attraction of the optimization procedure, corresponding to an apparent local minimum associated with better generalization. But earlier work (Bengio et al., 2007) had shown that even a purely supervised but greedy layer-wise procedure would give better results. So here instead of focusing on what unsupervised pre-training or semi-supervised criteria bring to deep architectures, we focus on analyzing what may be going wrong with good old (but deep) multi-layer neural networks.

Our analysis is driven by investigative experiments to monitor activations (watching for saturation of hidden units) and gradients, across layers and across training iterations. We also evaluate the effects on these of choices of activation function (with the idea that it might affect saturation) and initialization procedure (since unsupervised pre-training is a particular form of initialization and it has a drastic impact).

Appearing in Proceedings of the 13^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2010, Chia Laguna Resort, Sardinia, Italy. Volume 9 of JMLR: W&CP 9.

Problema: Desvanecimiento del Gradiente

¿Qué se puede hacer para resolver el problema del desvanecimiento de gradiente?

Problema: Desvanecimiento del Gradiente

¿Qué se puede hacer para resolver el problema del desvanecimiento de gradiente?

- En práctica, muchas librerías de redes neuronales ya tienen implementadas dichas soluciones.
- También se puede especificar algunos métodos como parámetros opcionales o capas al momento de crear y compilar la red.

Solución: Desvanecimiento del Gradiente

Inicialización de Glorot

Glorot y Bengio [2] proporcionan una solución: la señal que fluye en la red debe fluir adecuadamente en ambas direcciones:

- Hacia adelante para realizar las predicciones.
- Hacia atrás mediante backpropagation para determinar los gradientes.

Solución: Desvanecimiento del Gradiente

Inicialización de Glorot

Glorot y Bengio [2] proporcionan una solución: la señal que fluye en la red debe fluir adecuadamente en ambas direcciones:

- Hacia adelante para realizar las predicciones.
- Hacia atrás mediante backpropagation para determinar los gradientes.

Los autores proponen que la varianza de las salidas de cada capa deben ser igual a la varianza de las entradas.

Solución: Desvanecimiento del Gradiente

Inicialización de Glorot

Existen dos formas comunes para inicializar los pesos:

$$N\left(0, \frac{1}{\text{fan}_{\text{avg}}}\right) \tag{1}$$

$$U\left(-\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}, +\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}\right) \tag{2}$$

donde

$$\text{fan}_{\text{avg}} = \frac{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}{2}$$

fan_{in} es el número de entradas a la capa y fan_{out} es el número de neuronas de la capa.

Solución: Desvanecimiento del Gradiente

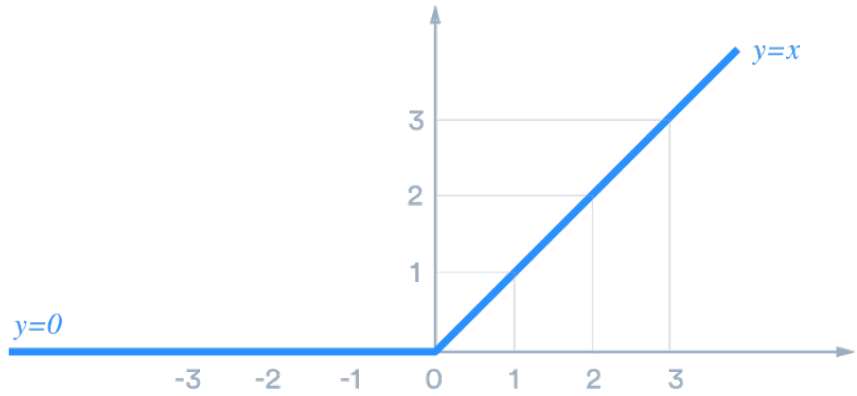
Inicialización de Glorot

```
model = keras.models.Sequential() #crea un modelo secuencial, el más simple en Keras
model.add(keras.layers.Flatten(input_shape = (28,28))) # la capa de entrada, Flatten covierte las imagenes en objetos de una dimensio
model.add(keras.layers.Dense(300, kernel_initializer = 'he_normal', activation='relu')) # la capa densa compuesta por 300 neuronas
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.Dense(100, activation='relu')) # segunda capa densa compuesta por 100 neuronas
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.Dense(10, activation='softmax')) # la capa de salida, una neurona por clase
```

Solución: Desvanecimiento del Gradiente

Funciones de Activación No Saturantes

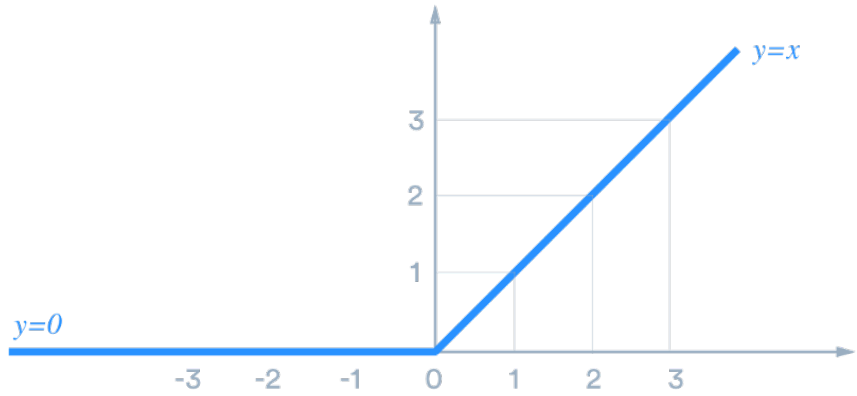
- La función sigmoide tiende a generar problemas ya que se satura.



Solución: Desvanecimiento del Gradiente

Funciones de Activación No Saturantes

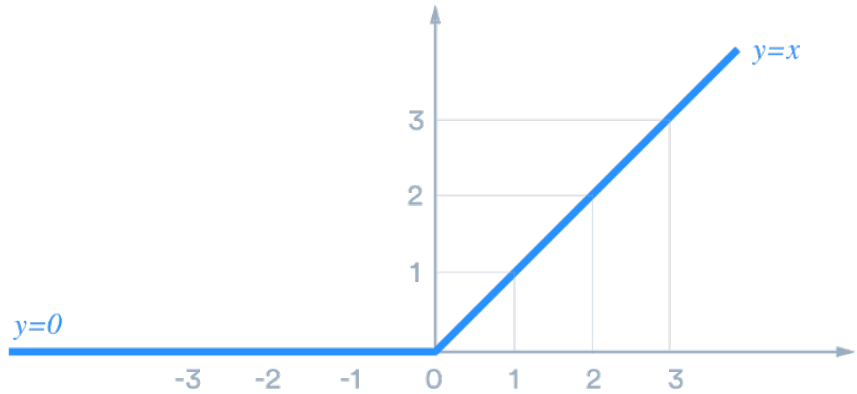
- La función sigmoide tiende a generar problemas ya que se satura.
- La ReLU también tiene sus problemas: muerte de neuronas por ReLu.



Solución: Desvanecimiento del Gradiente

Funciones de Activación No Saturantes

- La función sigmoide tiende a generar problemas ya que se satura.
- La ReLU también tiene sus problemas: muerte de neuronas por ReLu.
- Si la suma de las salidas de una neurona es negativa, ¡la ReLU convierte la salida en cero!



Solución: Desvanecimiento del Gradiente

Leaky ReLU

Una variante de la ReLU definida como

$$\text{LeakyReLU}_{\alpha}(z) = \text{máx}(\alpha z, z) \tag{3}$$

donde el parámetro α regula la fuga, usualmente toma el valor de 0.01.

La pequeña pendiente que se genera ocasiona que las neuronas no muera, pueden entrar en coma, pero pueden revivir en un futuro.

Solución: Desvanecimiento del Gradiente

Leaky ReLU

Una variante de la ReLU definida como

$$\text{LeakyReLU}_{\alpha}(z) = \text{máx}(\alpha z, z) \tag{3}$$

donde el parámetro α regula la fuga, usualmente toma el valor de 0.01.

La pequeña pendiente que se genera ocasiona que las neuronas no muera, pueden entrar en coma, pero pueden revivir en un futuro.

Ejercicio

¿Cómo se ve la función de activación LeakyReLU?

Solución: Desvanecimiento del Gradiente

Exponential Linear Unit

Otra función de activación propuesta es la *Exponential Linear Unit* (ELU) definida como

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & si \quad z < 0 \\ z & si \quad z \geq 0 \end{cases} \tag{4}$$

- La ELU tomar valores negativos y permite acercarlos a cero, sin ser cero.
- No tiene gradiente igual a cero para valores negativos que salen de una neurona; evita las neuronas muertas.
- Si $\alpha = 1$, es suave, lo que la hace derivable incluso en cero, acelerando el proceso del gradiente.

Solución: Desvanecimiento del Gradiente

Funciones de Activación No Saturantes

Ejercicio

¿Cómo se ve la ELU?

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & si \quad z < 0 \\ z & si \quad z \geq 0 \end{cases} \tag{5}$$

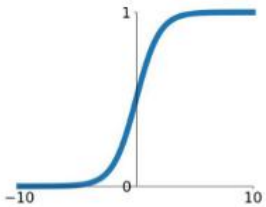
Solución: Desvanecimiento del Gradiente

Funciones de Activación No Saturantes

Activation Functions

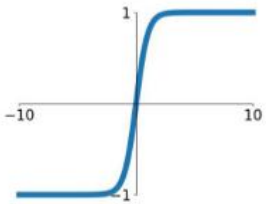
Sigmoid

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



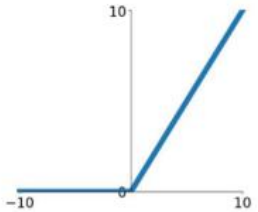
tanh

$$\tanh(x)$$



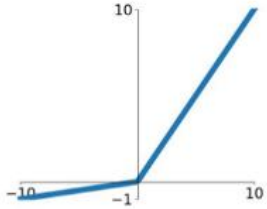
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

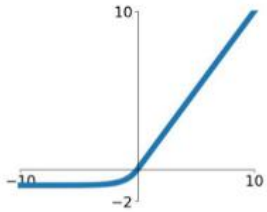


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Solución: Desvanecimiento del Gradiente

Funciones de Activación No Saturantes

```
model = keras.models.Sequential() #crea un modelo secuencial, el más simple en Keras
model.add(keras.layers.Flatten(input_shape = (28,28))) # la capa de entrada, Flatten covierte las imagenes en objetos de una dimensic
model.add(keras.layers.Dense(300, kernel_initializer = 'he_normal')) # la capa densa compuesta por 300 neuronas
model.add(keras.layers.LeakyReLU(alpha=0.2))
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.Dense(100, activation='relu')) # segunda capa densa compuesta por 100 neuronas
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.Dense(10, activation='softmax')) # la capa de salida, una neurona por clase
```

Solución: Desvanecimiento del Gradiente

Otras Opciones

Otras opciones (que se dejan como temas si tienen interés):

- Batch Normalization
- Gradient Clipping

Problema: ?

Problema

- Tenemos una red neuronal entrenada con un problema en particular. Por el momento, supongamos que es una red que permite clasificar 100 objetos distintos en una imagen: animales, plantas y rocas.
- Por otro lado, necesitamos hacer una tarea que parece similar: identificar razas de perros.
- ¿Qué se les ocurre hacer?

Problema: Reutilizar Capas Preentrenadas

Solución

Es posible reutilizar la red entrenada, específicamente entrenar las capas bajas de la red.

Problema: Reutilizar Capas Preentrenadas

Solución

Es posible reutilizar la red entrenada, específicamente entrenar las capas bajas de la red.

A lo anterior se le conoce como *Transfer Learning*:

- Disminuye el tiempo necesario para entrenar la red.
- Requiere una cantidad MENOR de datos.

Solución: Transfer Learning

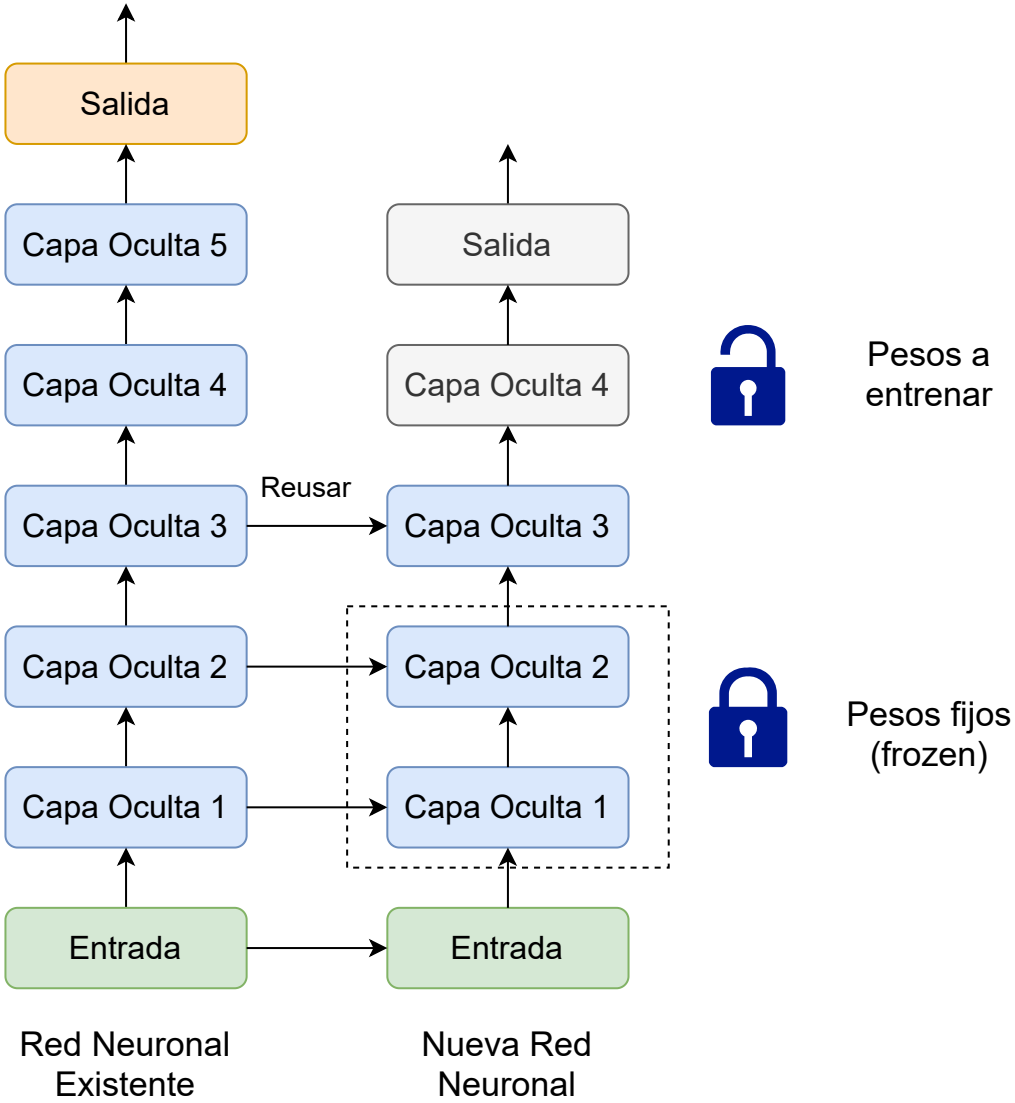


Figura: Idea básica del *Transfer Learning*.

Problema: Entrenamiento de la Red

- Entrenar una red neuronal puede llegar a ser lento. Muy lento.
- Entre más parámetros, mayor cantidad de ajustes debe calcular.
- Ya exploramos algunas opciones para acelerar el proceso:
  - Una buena inicialización de los pesos.
  - Batch normalization.*
  - Elegir una buena función de activación.
  - Considerar usar Transfer Learning.

Problema: Entrenamiento de la Red

- Entrenar una red neuronal puede llegar a ser lento. Muy lento.
- Entre más parámetros, mayor cantidad de ajustes debe calcular.
- Ya exploramos algunas opciones para acelerar el proceso:
  - Una buena inicialización de los pesos.
  - Batch normalization.*
  - Elegir una buena función de activación.
  - Considerar usar Transfer Learning.

Pregunta

¿Que otra cosa se puede hacer?

Problema: Entrenamiento de la Red

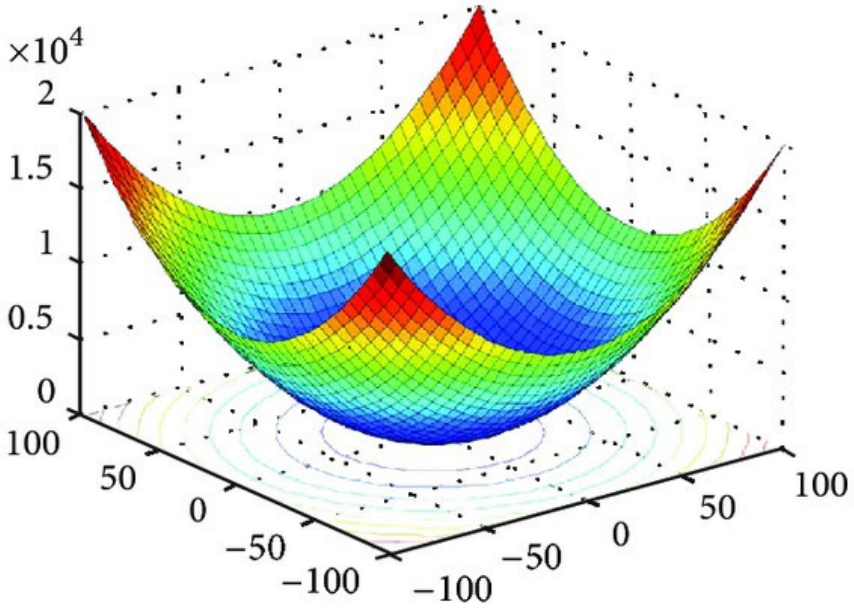
Solución

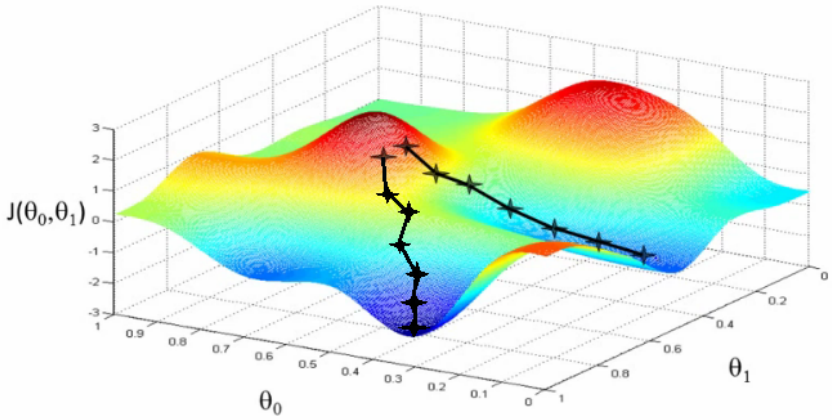
Gradiente Descendiente, como método de optimización, puede ser lento. Existen mejores algoritmos.

Problema: Entrenamiento de la Red

Solución

Gradiente Descendiente, como método de optimización, puede ser lento. Existen mejores algoritmos.





Solución: Momentum Optimization

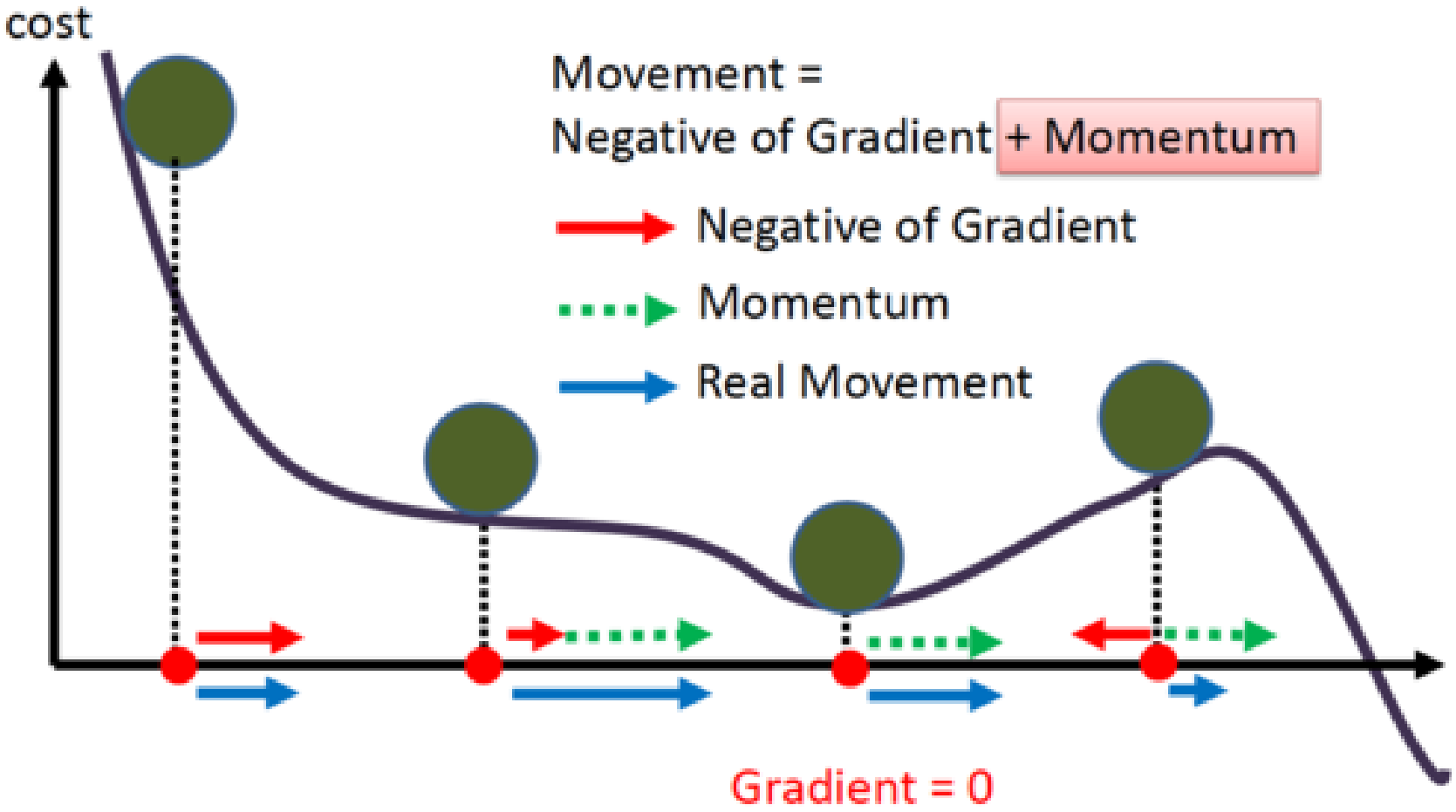


Figura: Propuesta por Polyak [4] en 1964, la idea es añadir momento al gradiente descendiente que solo considera pequeños pasos cuesta abajo.

Solución: Momentum Optimization

Gradiente descendiente actualiza los pesos θ restando directamente el gradiente de la función de costo ($J(\theta)$) respecto a los pesos ($\nabla_\theta J(\theta)$) multiplicado por la razón de aprendizaje η:

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \tag{6}$$

De la ecuación anterior, se aprecia que el gradiente no considera los gradientes de pasos anteriores, solo el de donde se encuentra.

Solución: Momentum Optimization

Este algoritmo de optimización:

- En cada iteración, substrae al vector de momento $\boldsymbol{m}$ el gradiente local (multiplicado por η),
- Actualiza los pesos añadiendo el vector de momento.

$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{7}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m} \tag{8}$$

- β simula la fricción y es un valor entre 0 y 1, usualmente 0.9.

Solución: Momentum Optimization

Otras opciones (se dejan a su curiosidad):

- Nesterov Accelerated Gradient
- AdaGrad
- RMSProp
- Adam y Nadam Optimization

Problema: Sobreajuste en Redes Nerunales

Una red neuronal puede contener miles de parámetros, puede que millones. Esto le permite ajustarse a diversos conjuntos de datos que modelan sistemas complejos.

Problema: Sobreajuste en Redes Nerunales

Una red neuronal puede contener miles de parámetros, puede que millones. Esto le permite ajustarse a diversos conjuntos de datos que modelan sistemas complejos.

Sin embargo, esta flexibilidad también puede llevar al sobreajuste en el conjunto de entrenamiento.

Problema: Sobreajuste en Redes Nerunales

Una red neuronal puede contener miles de parámetros, puede que millones. Esto le permite ajustarse a diversos conjuntos de datos que modelan sistemas complejos.

Sin embargo, esta flexibilidad también puede llevar al sobreajuste en el conjunto de entrenamiento.

¿Qué debemos hacer?

Problema: Sobreajuste en Redes Nerunales

Una red neuronal puede contener miles de parámetros, puede que millones. Esto le permite ajustarse a diversos conjuntos de datos que modelan sistemas complejos.

Sin embargo, esta flexibilidad también puede llevar al sobreajuste en el conjunto de entrenamiento.

¿Qué debemos hacer? ¡Regularización!

Solución: Regularización

Dropout

- Dropout es la técnica más usada para regularizar redes neuronales.
- Propuesta en el 2012 por Hinton [3], permite mejoras en precisión del 1-2 %.
- En cada paso del entrenamiento, cada neurona tiene una probabilidad p de ser ignorada.
- El parámetro p oscila entre 10-50 %:
  - 20-30 % para redes neuronales.
  - 40-50 % para redes neuronales convolucionales.

Solución: Regularización

Dropout

Pregunta

¿Por qué creen que esto funciona?

Solución: Regularización

Otras opciones:

- Monte Carlo Dropout
- Max-Norm Regularization

Bibliografía Sugerida

[1] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). November 2015.

[2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9, pages 249–256. PMLR, 2010.

[3] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. July 2012.

[4] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.