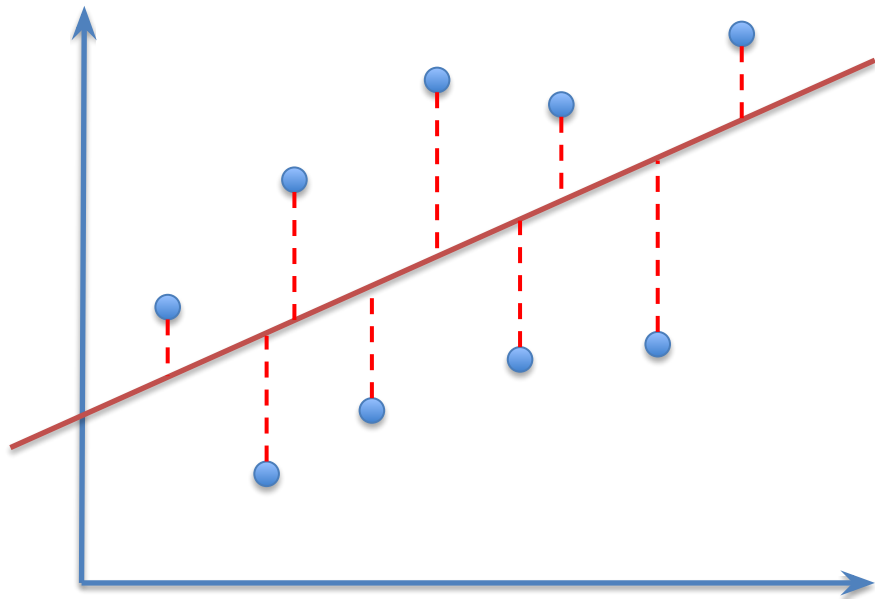


Técnicas Avanzadas de Data Mining y Sistemas Inteligentes

Maestría en Informática
Escuela de Posgrado
Pontificia Universidad Católica del Perú

2018-2

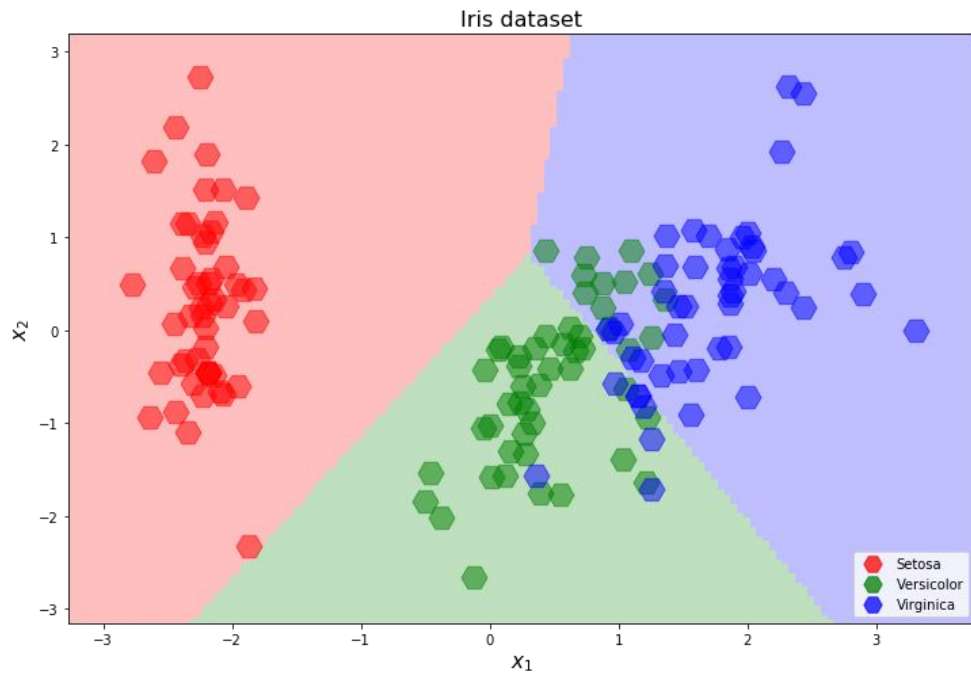
Regresión lineal



$$f(x_i) = ax_i + b$$

$$L = \frac{1}{2m} \sum_{i=1}^m (f(x_i) - y_i)^2$$

Regresión logística

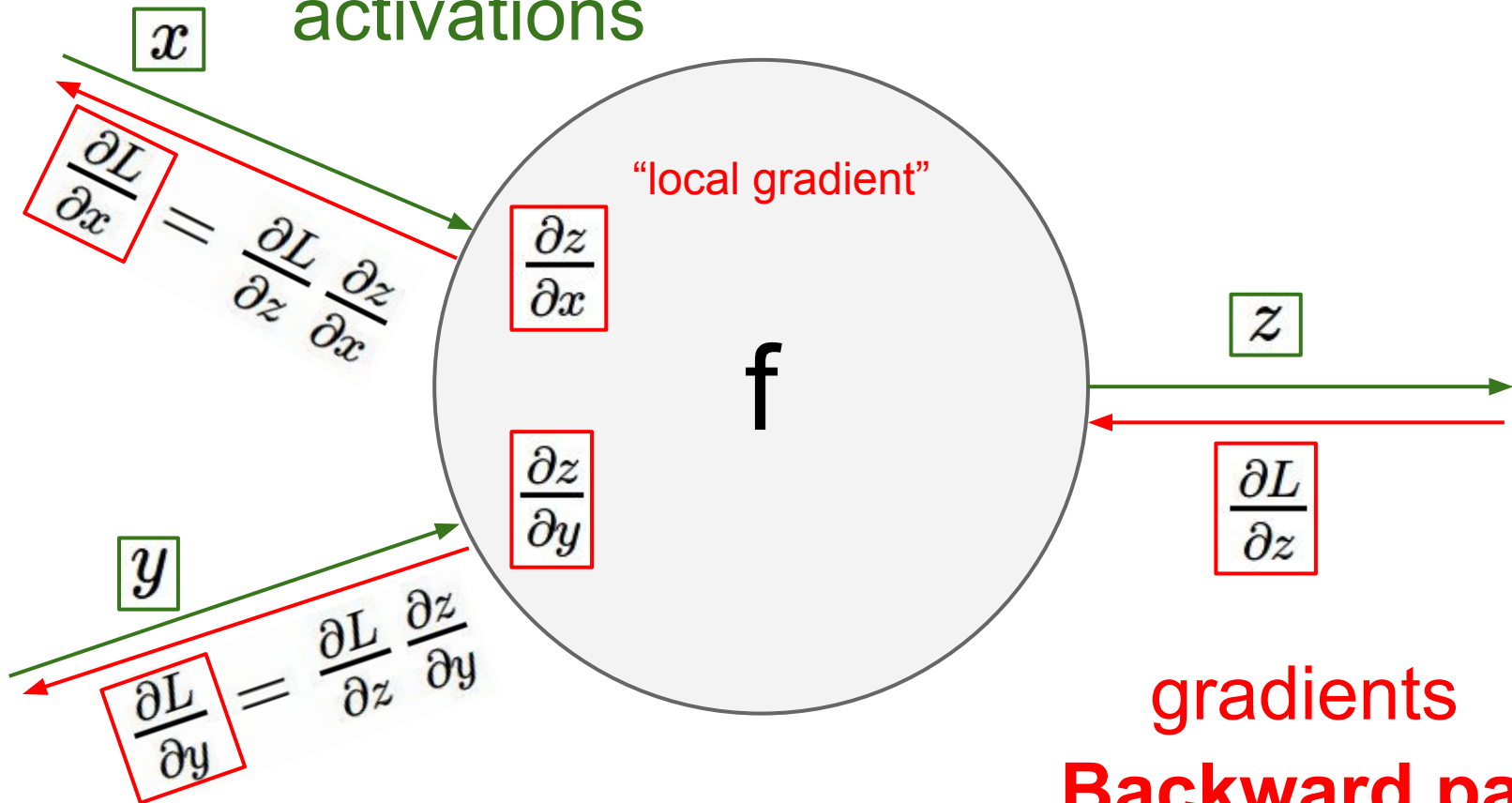


$$f(x_i) = \text{softmax}(Wx_i + b)$$

$$L = - \sum_{i=1}^m y_i \log(\hat{y}_i)$$

Forward pass

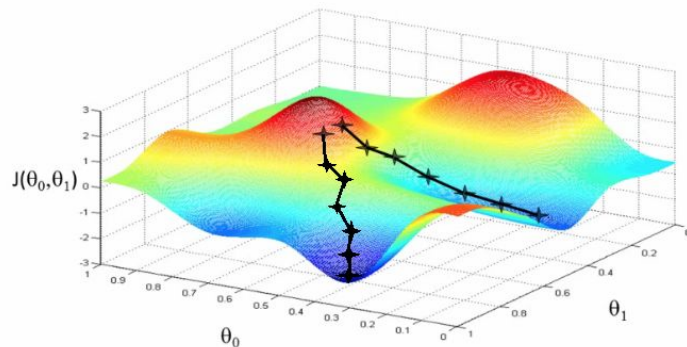
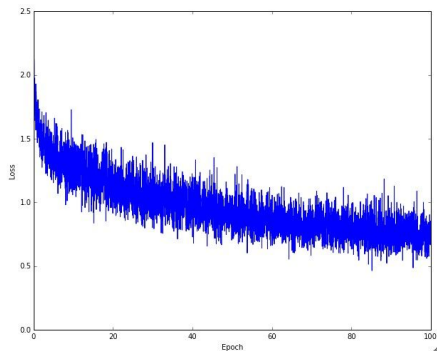
activations



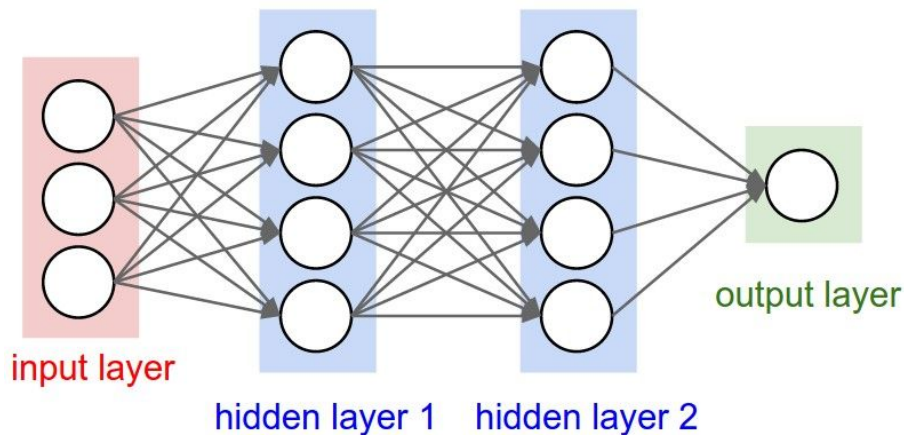
Optimización: Mini-batch Gradient descent

Iterar:

1. **Sample:** obtener una muestra de la data.
2. **Forward:** obtener la pérdida (Loss).
3. **Backprop:** calcular las gradientes.
4. **Actualizar** los parámetros del modelo.



Redes Neuronales



$$h_1(x_i) = \sigma(W_1 x_i + b_1)$$

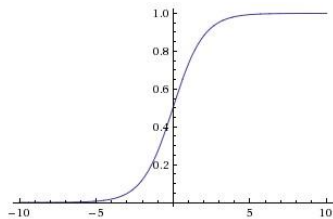
$$h_2(x_i) = \sigma(W_2 h_1(x_i) + b_2)$$

$$f(x_i) = \textit{softmax}(W_m h_n(x_i) + b_m)$$

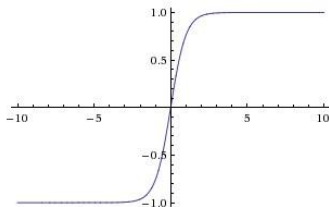
Funciones de Activación

Sigmoid

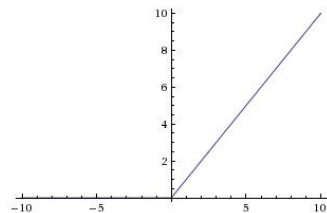
$$\sigma(x) = 1/(1 + e^{-x})$$



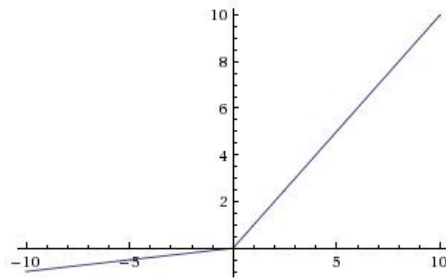
tanh tanh(x)



ReLU max(0,x)

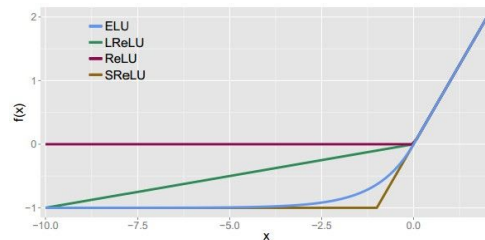


Leaky ReLU max(0.1x, x)

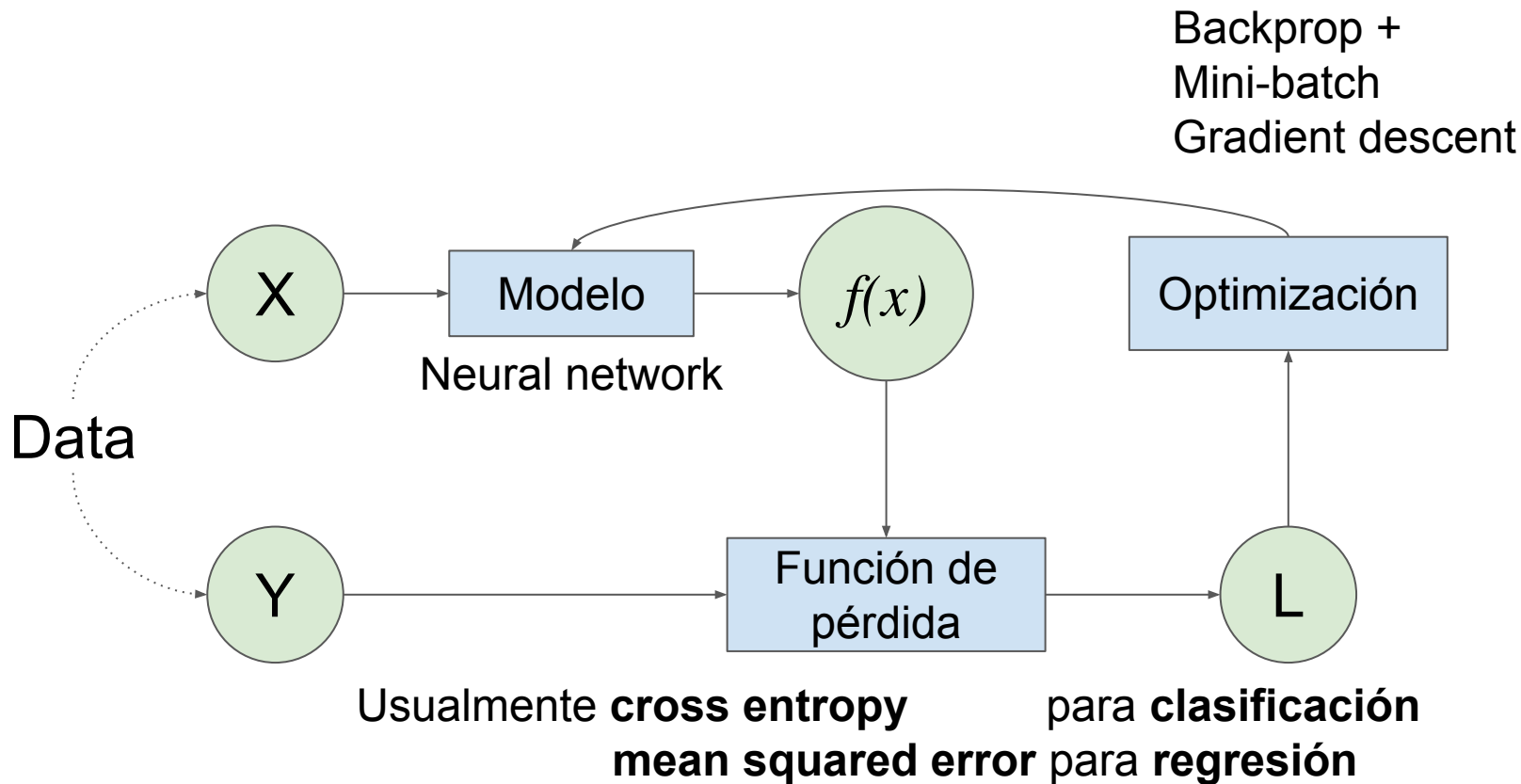


ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

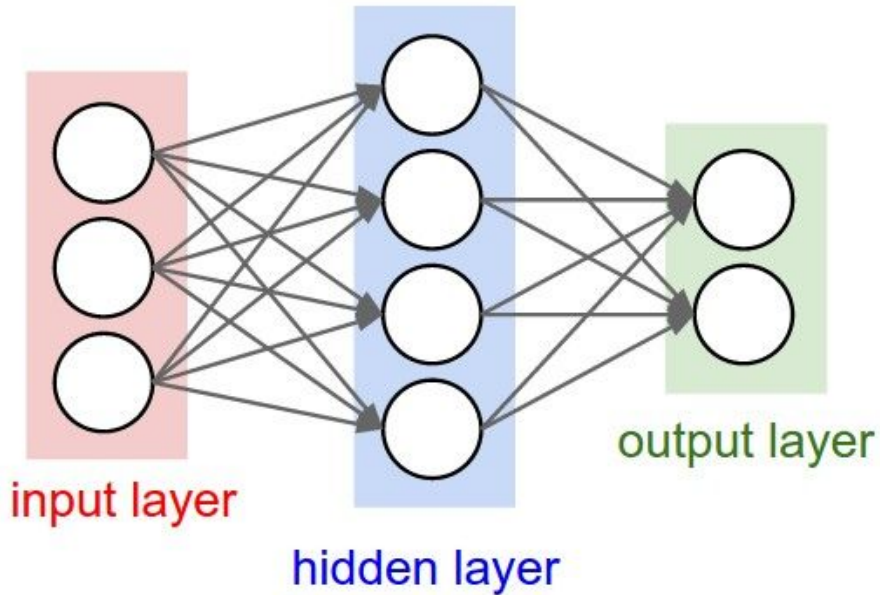


Redes Neuronales

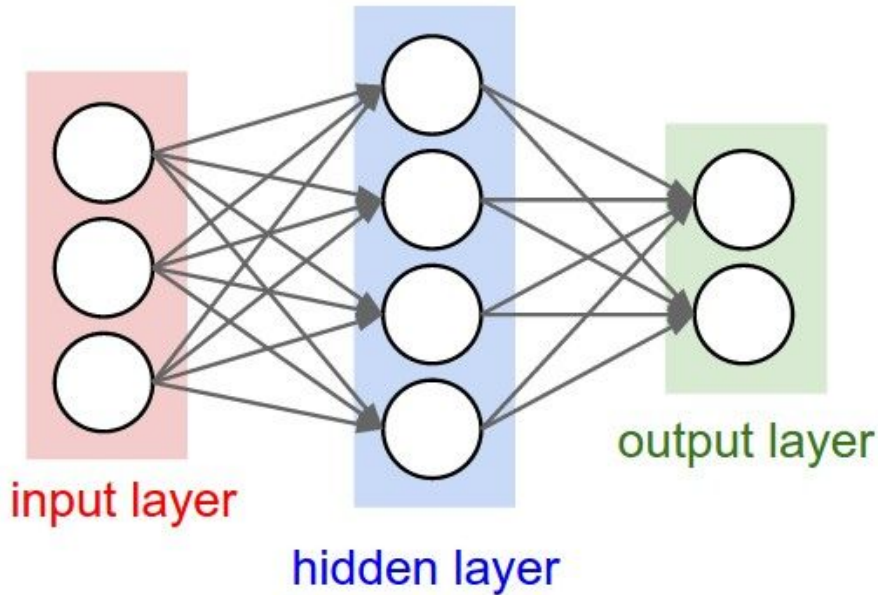


Inicialización de pesos

Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.

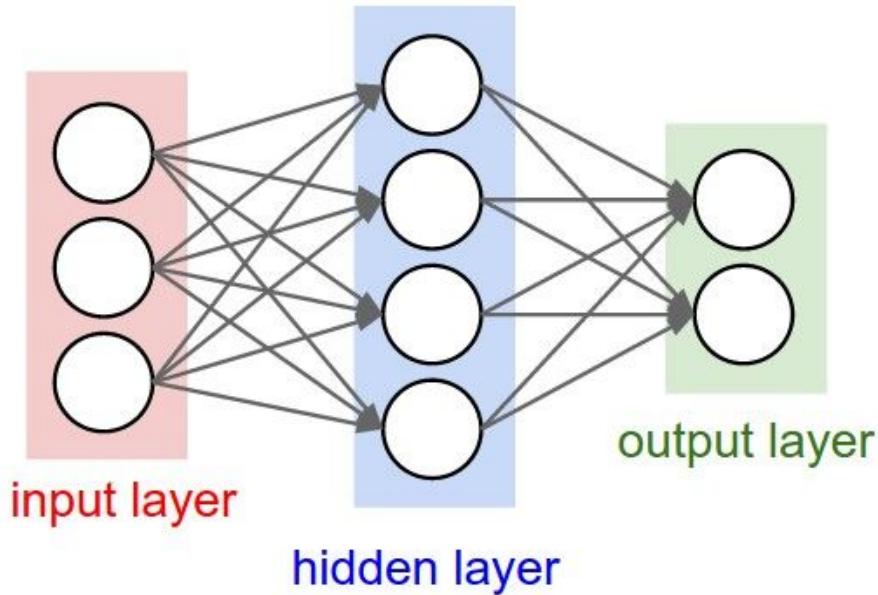


Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.



¿Qué pasa si iniciamos la red con todos los **W=0**?

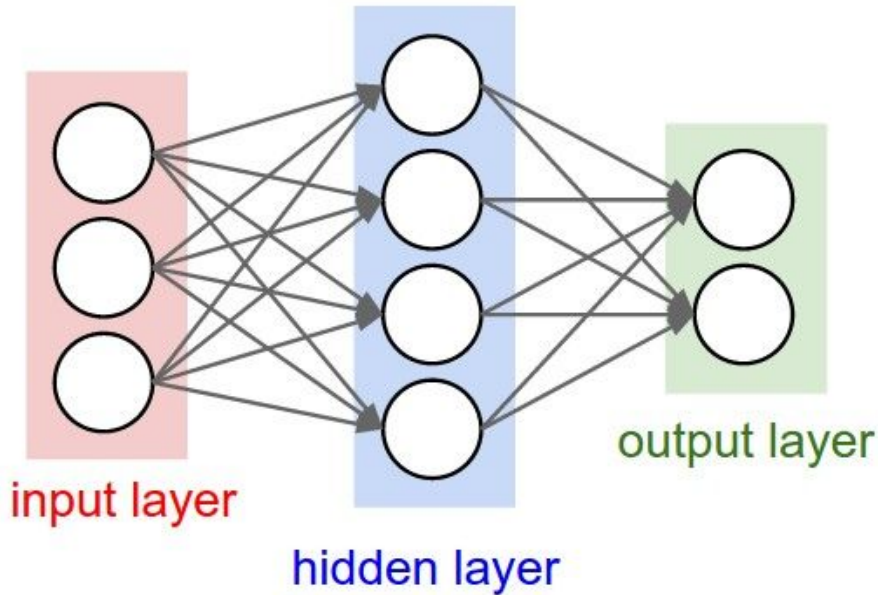
Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.



¿Qué pasa si iniciamos la red con todos los **W=0**?

Todas las neuronas van a hacer las mismas operaciones, van a tener las mismas gradientes y se van a actualizar de forma parecida.

Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.

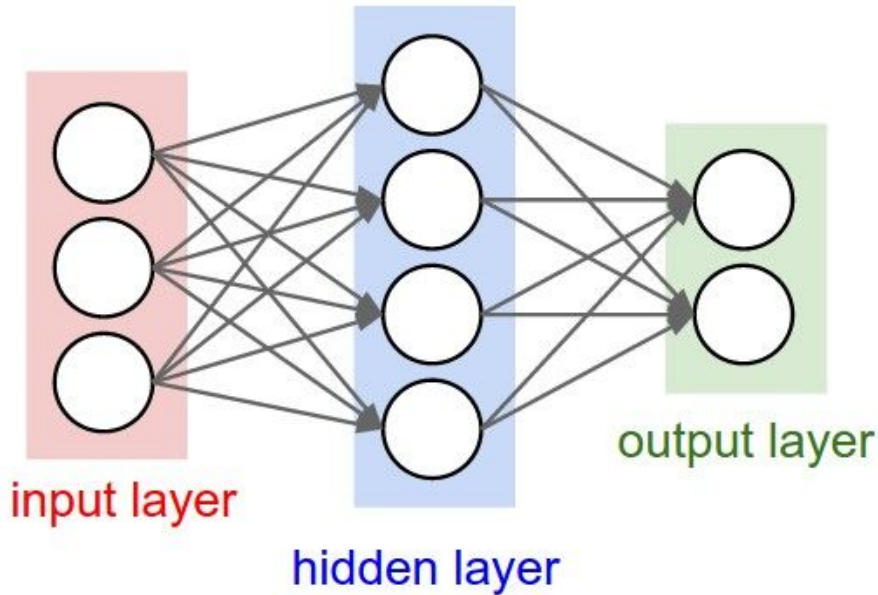


¿Qué pasa si iniciamos la red con **W** pequeños?

(ej: una distribución normal centrada en 0, con una desviación estándar de 0.01)

> `np.random.normal(loc=0.0, scale=0.01)`

Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.



¿Qué pasa si iniciamos la red con **W** pequeños?

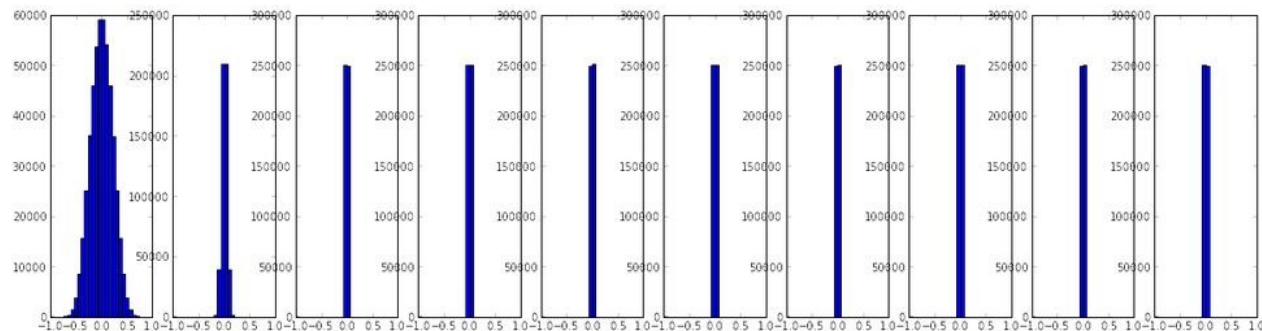
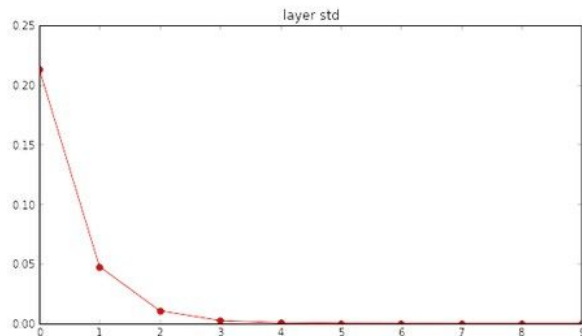
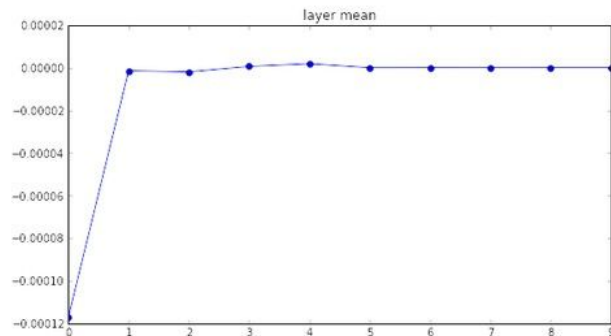
(ej: una distribución normal centrada en 0, con una desviación estándar de 0.01)

> `np.random.normal(loc=0.0, scale=0.01)`

Puede funcionar bien para redes pequeñas, pero puede conducir a distribuciones no homogéneas de activaciones a medida que incrementamos el número de capas de la red.

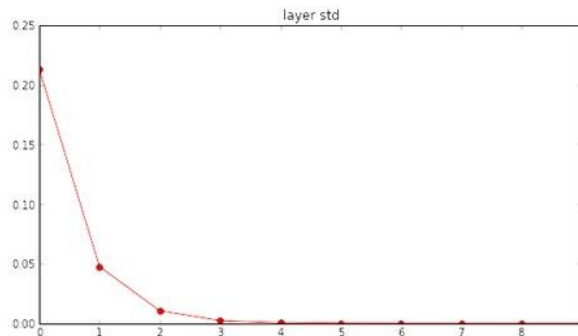
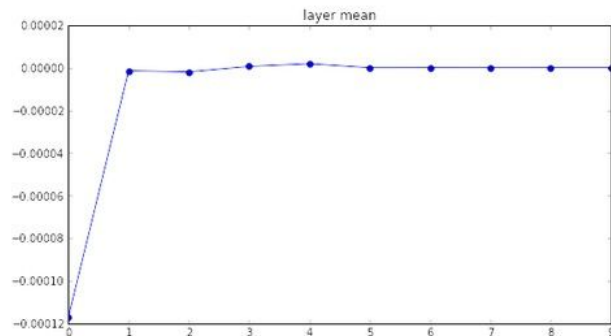
Ej: 10 capas, 500 neuronas por capa, activaciones tanh

input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000

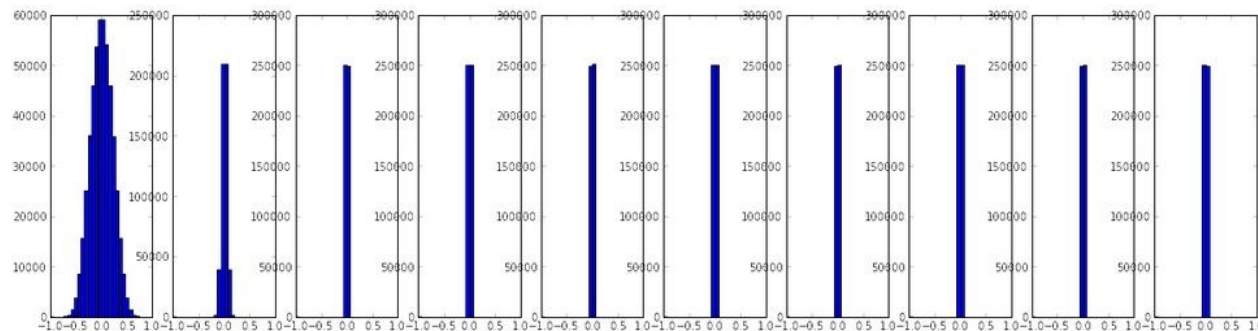


Ej: 10 capas, 500 neuronas por capa, activaciones tanh

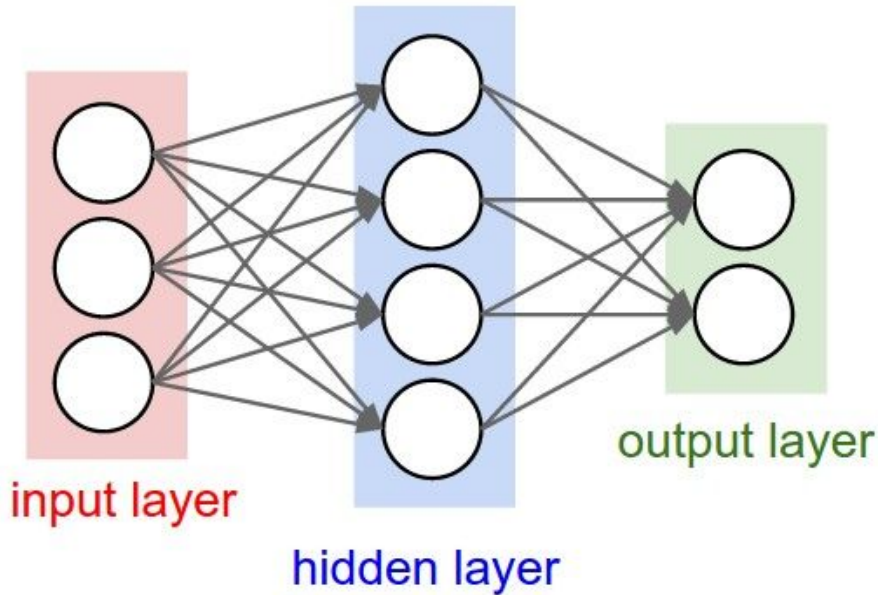
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000



Todas las
activaciones se
centran en cero



Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.



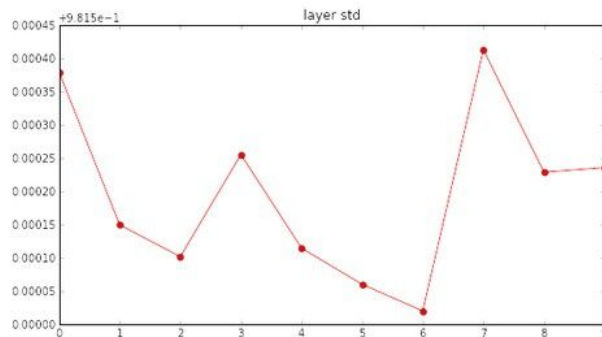
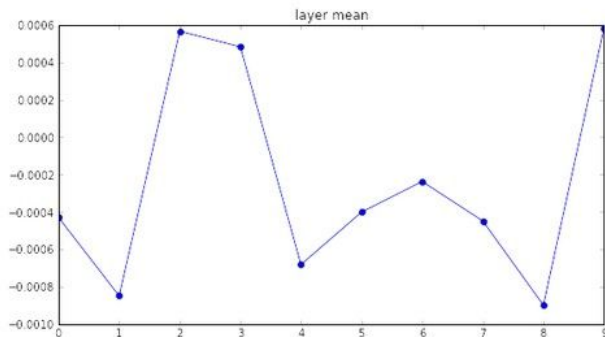
¿Y si intentamos con **W** más grandes?

(ej: una distribución normal centrada en 0, con una desviación estándar de 1)

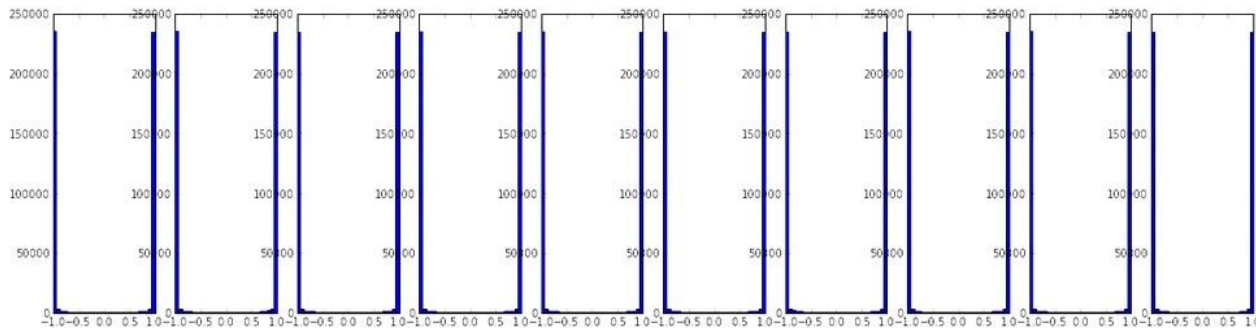
> `np.random.normal(loc=0.0, scale=1.0)`

Ej: 10 capas, 500 neuronas por capa, activaciones tanh

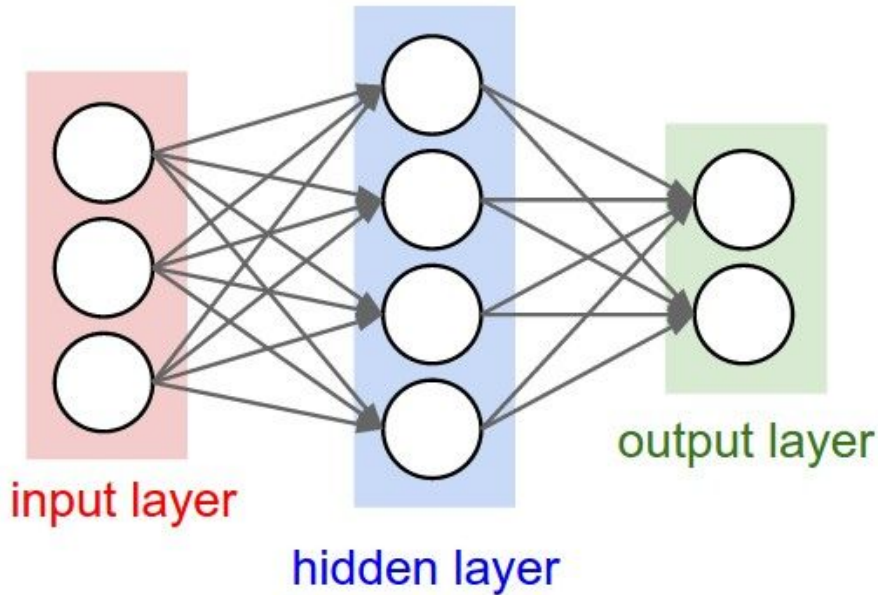
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736



Las neuronas se saturan, llegando a tomar valores de -1 o 1



Al iniciar una red neuronal, tenemos que inicializar los pesos **W** con valores numéricos.



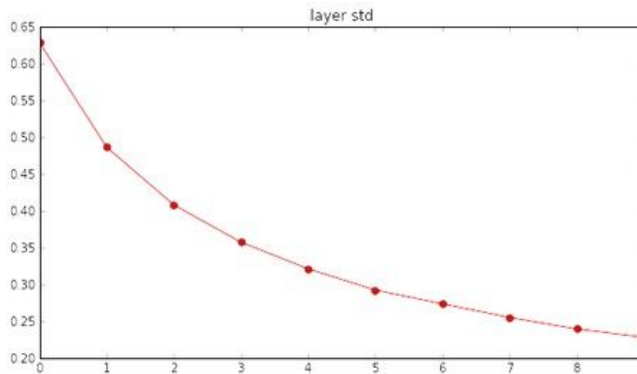
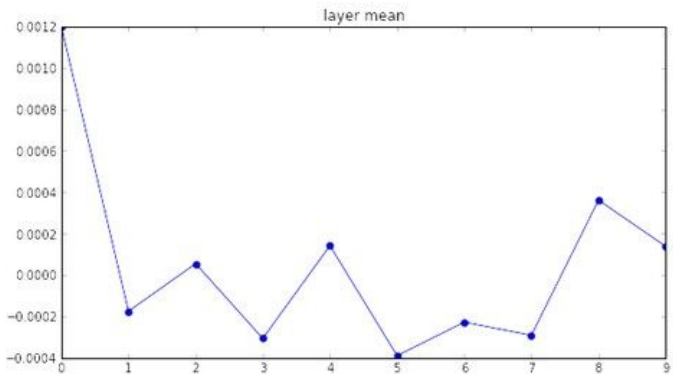
Idea:
Escalar la varianza
dependiendo del número
de neuronas.

```
> np.random.normal(scale=1/np.sqrt(input_neurons))
```

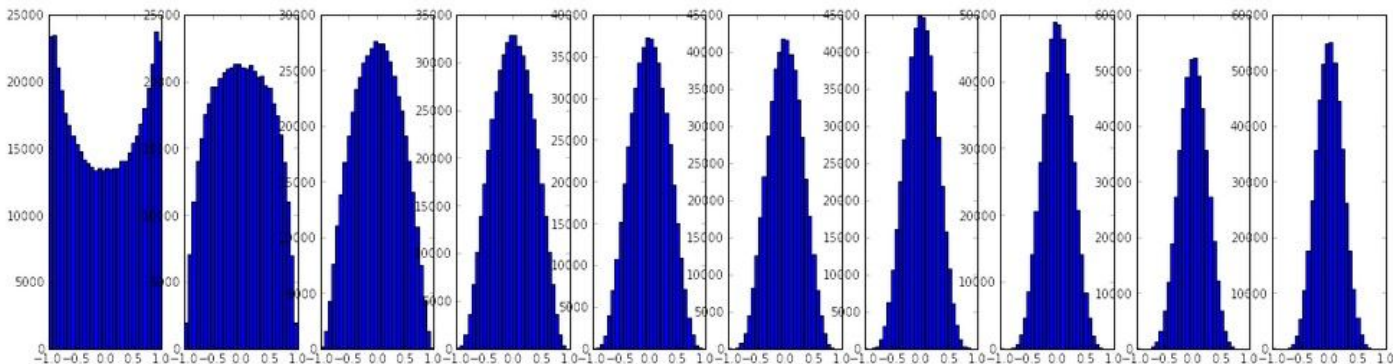
“Xavier initialization”

Ej: 10 capas, 500 neuronas por capa, activaciones tanh

“Xavier initialization”
[Glorot et al., 2010]



Se conserva
la distribución
de las salidas
de cada capa



Xavier Initialization

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

$$\text{Var}(W_iX_i) = E[X_i]^2 \text{Var}(W_i) + E[W_i]^2 \text{Var}(X_i) + \text{Var}(W_i)\text{Var}(X_i)$$

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

Xavier Initialization

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

$$\text{Var}(W_iX_i) = \cancel{E[X_i]^2\text{Var}(W_i)} + \cancel{E[W_i]^2\text{Var}(X_i)} + \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(W_iX_i) = \text{Var}(W_i)\text{Var}(X_i)$$

Asumimos que los datos se encuentran normalizados, por lo que el $E[X] = 0$.
Y los pesos vienen de una distribución centrada en 0, por lo que $E[W] = 0$.

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

Xavier Initialization

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

$$\text{Var}(W_iX_i) = \cancel{E[X_i]^2\text{Var}(W_i)} + \cancel{E[W_i]^2\text{Var}(X_i)} + \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(W_iX_i) = \text{Var}(W_i)\text{Var}(X_i)$$

$$\boxed{\text{Var}(Y)} = \text{Var}(W_1X_1 + W_2X_2 + \cdots + W_nX_n) = n\text{Var}(W_i)\boxed{\text{Var}(X_i)}$$

Lo que queremos es que la variación de X se conserve en Y

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

Xavier Initialization

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

$$\text{Var}(W_iX_i) = \cancel{E[X_i]^2\text{Var}(W_i)} + \cancel{E[W_i]^2\text{Var}(X_i)} + \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(W_iX_i) = \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(Y) = \text{Var}(W_1X_1 + W_2X_2 + \cdots + W_nX_n) = n\text{Var}(W_i)\text{Var}(X_i)$$

Queremos que esta expresión sea igual a 1

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

Xavier Initialization

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

$$\text{Var}(W_iX_i) = \cancel{E[X_i]^2\text{Var}(W_i)} + \cancel{E[W_i]^2\text{Var}(X_i)} + \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(W_iX_i) = \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(Y) = \text{Var}(W_1X_1 + W_2X_2 + \cdots + W_nX_n) = n\text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(W_i) = \frac{1}{n} = \frac{1}{n_{\text{in}}}$$

Queremos que esta expresión sea igual a 1

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

Xavier Initialization

$$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$

$$\text{Var}(W_iX_i) = \cancel{E[X_i]^2\text{Var}(W_i)} + \cancel{E[W_i]^2\text{Var}(X_i)} + \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(W_iX_i) = \text{Var}(W_i)\text{Var}(X_i)$$

$$\text{Var}(Y) = \text{Var}(W_1X_1 + W_2X_2 + \cdots + W_nX_n) = \boxed{n\text{Var}(W_i)}\text{Var}(X_i)$$

$$\text{Var}(W_i) = \frac{1}{n} = \frac{1}{n_{\text{in}}}$$

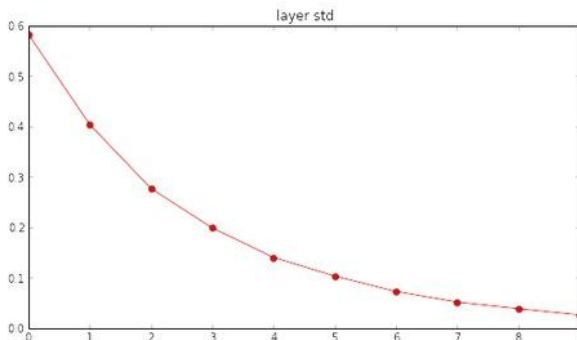
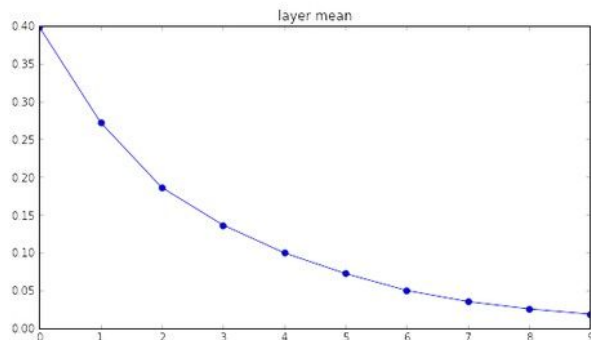
Queremos que esta expresión sea igual a 1

`np.random.normal(scale=1/np.sqrt(input_neurons))`

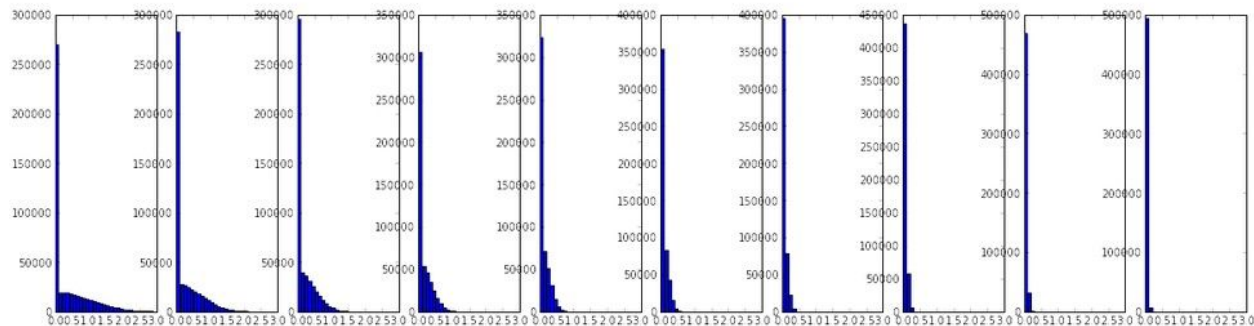
Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

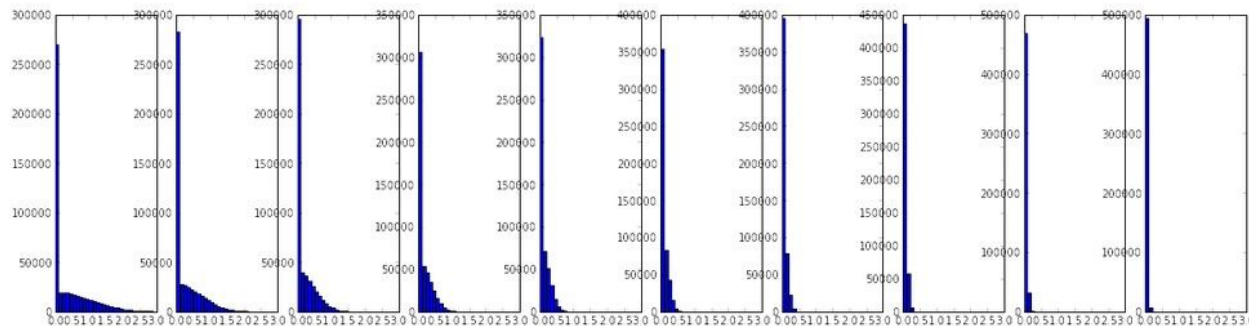
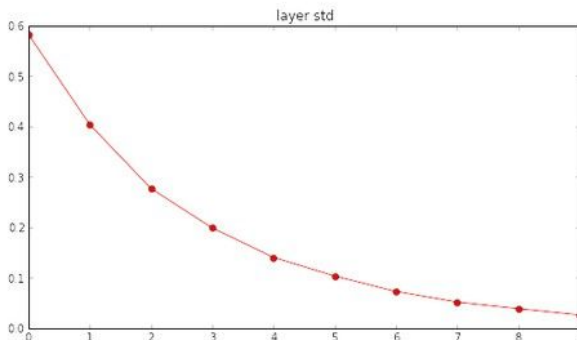
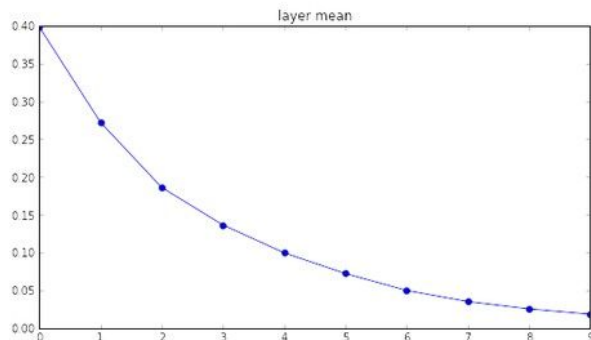
`np.random.normal(scale=1/np.sqrt(input_neurons))`



Pero cuando
usamos Relu...

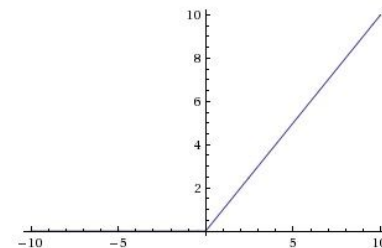


```
np.random.normal(scale=1/np.sqrt(input_neurons))
```

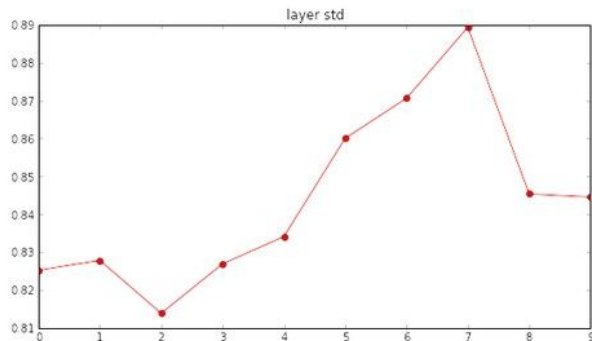
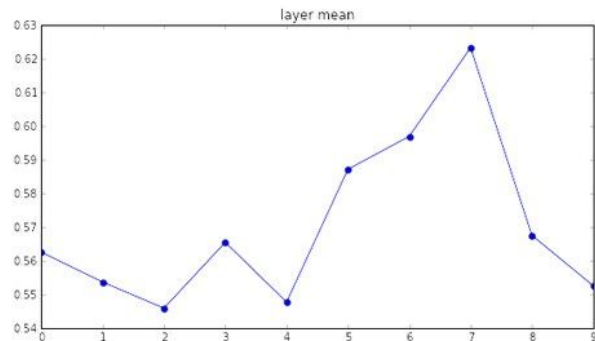


Pero cuando
usamos Relu...

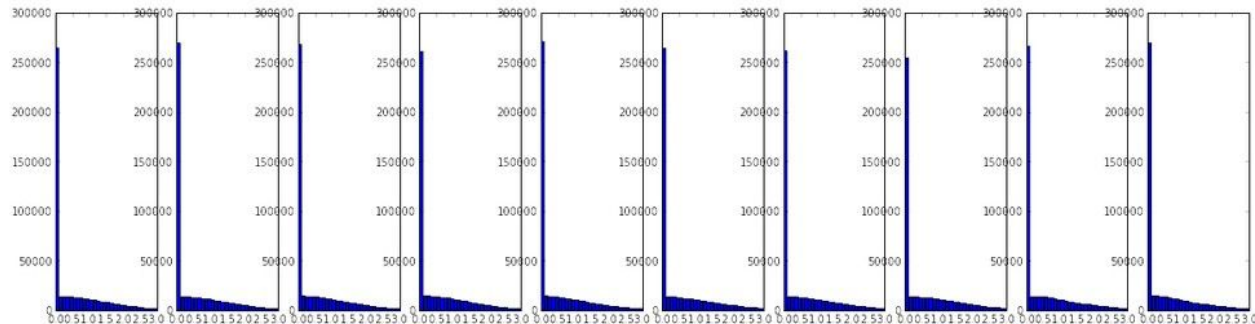
El problema es que las salidas son números positivos, por lo que tenemos sólo la mitad de la varianza.



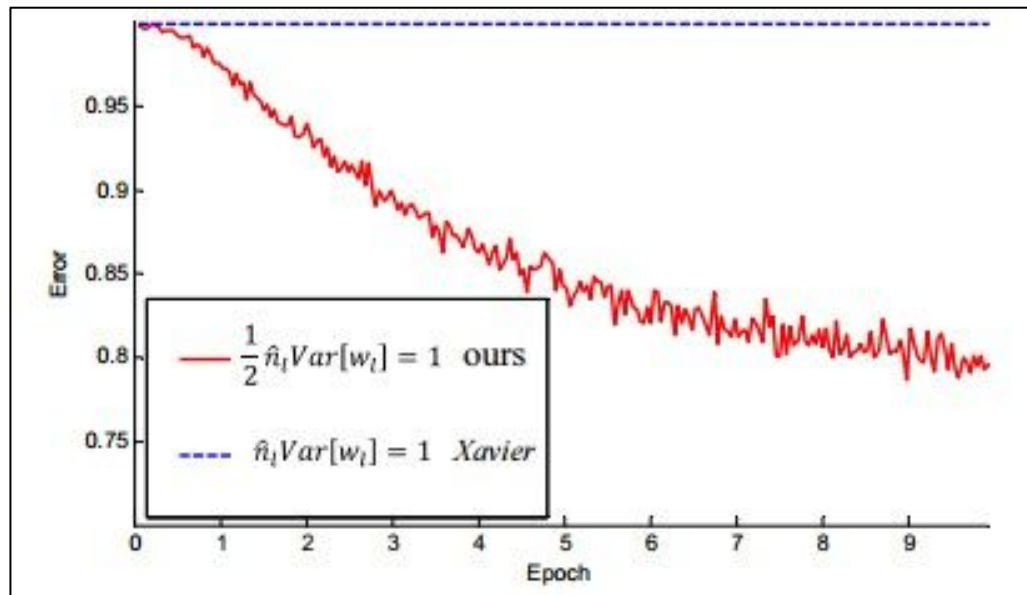
`np.random.normal(scale=1/np.sqrt(input_neurons / 2))`



Ajustamos la distribución
de números aleatorios.
“He initialization”
[He et al., 2015]



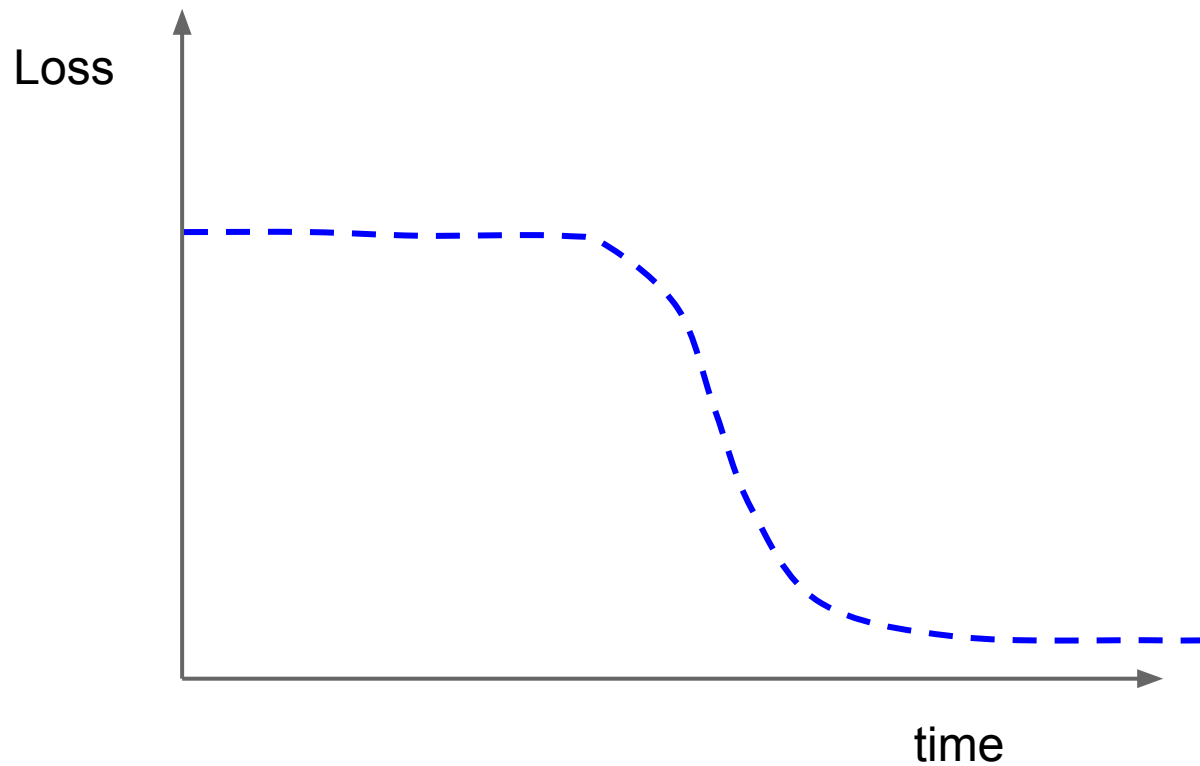
He Initialization



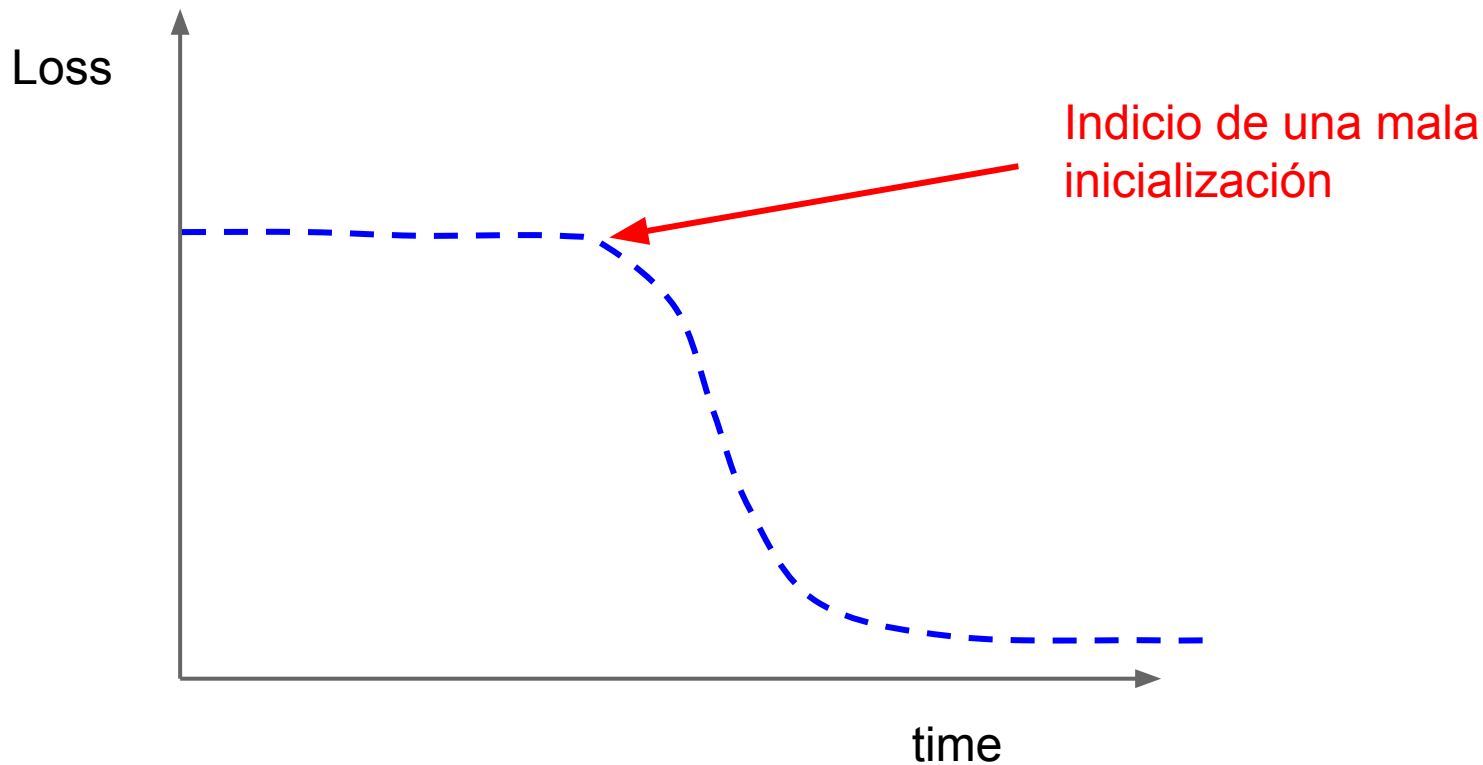
$$\text{Var}(W) = \frac{2}{n_{\text{in}}}$$

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification
by He et al., 2015

Inicialización de pesos



Inicialización de pesos



Inicialización de pesos

- Xavier initialization (también conocida como glorot):

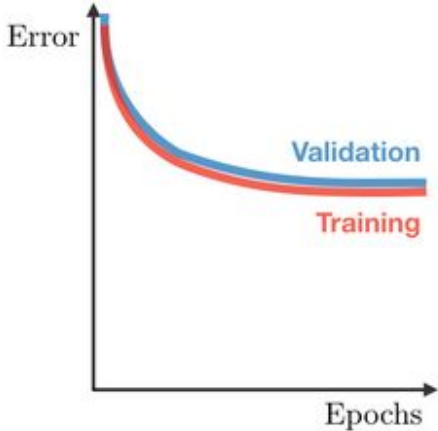
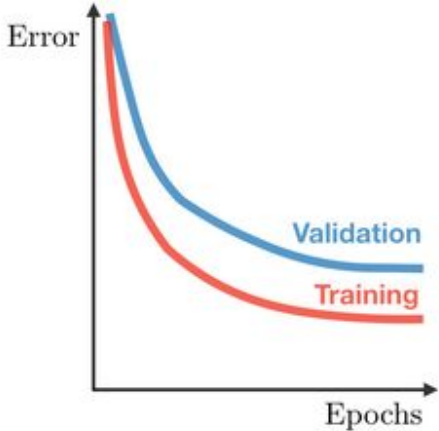
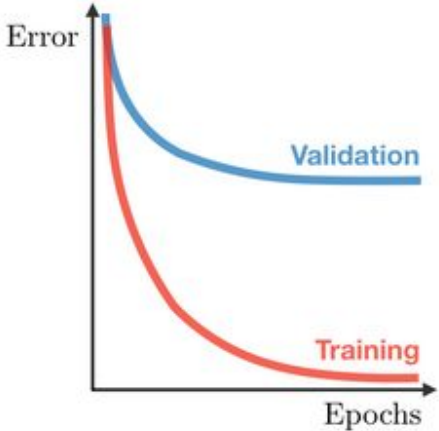
```
Dense(100, activation='relu', kernel_initializer='glorot_normal')
```

- He initialization (también conocida como kaiming):

```
Dense(100, activation='relu', kernel_initializer='he_normal')
```

Regularización

Regularización

	Underfitting	Just right	Overfitting
Deep learning illustration			
Possible remedies	<ul style="list-style-type: none">• Complexify model• Add more features• Train longer		<ul style="list-style-type: none">• Perform regularization• Get more data

Regularización

- Weight regularization
- Batch normalization
- Dropout

Weight regularization

Weight regularization

$$\hat{y} = f(x_i) = Wx_i + b$$

Si encontramos los pesos W que hacen que la pérdida sea $L=0$.

Weight regularization

$$\hat{y} = f(x_i) = Wx_i + b$$

Si encontramos los pesos W que hacen que la pérdida sea $L=0$.

¿Serán estos los únicos valores de W que hagan que la pérdida sea 0?

Weight regularization

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

Weight regularization

regularization strength
(hyperparameter)

$$L + \lambda R(W)$$

L2 regularization

L1 regularization

Elastic net (L1 + L2)

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

Weight regularization

L2 regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$R(w_1) = 1$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$R(w_2) = 0.25$$

$$w_1^T x = w_2^T x = 1$$

Weight regularization

L2 regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$R(w_1) = 1$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$R(w_2) = 0.25$$

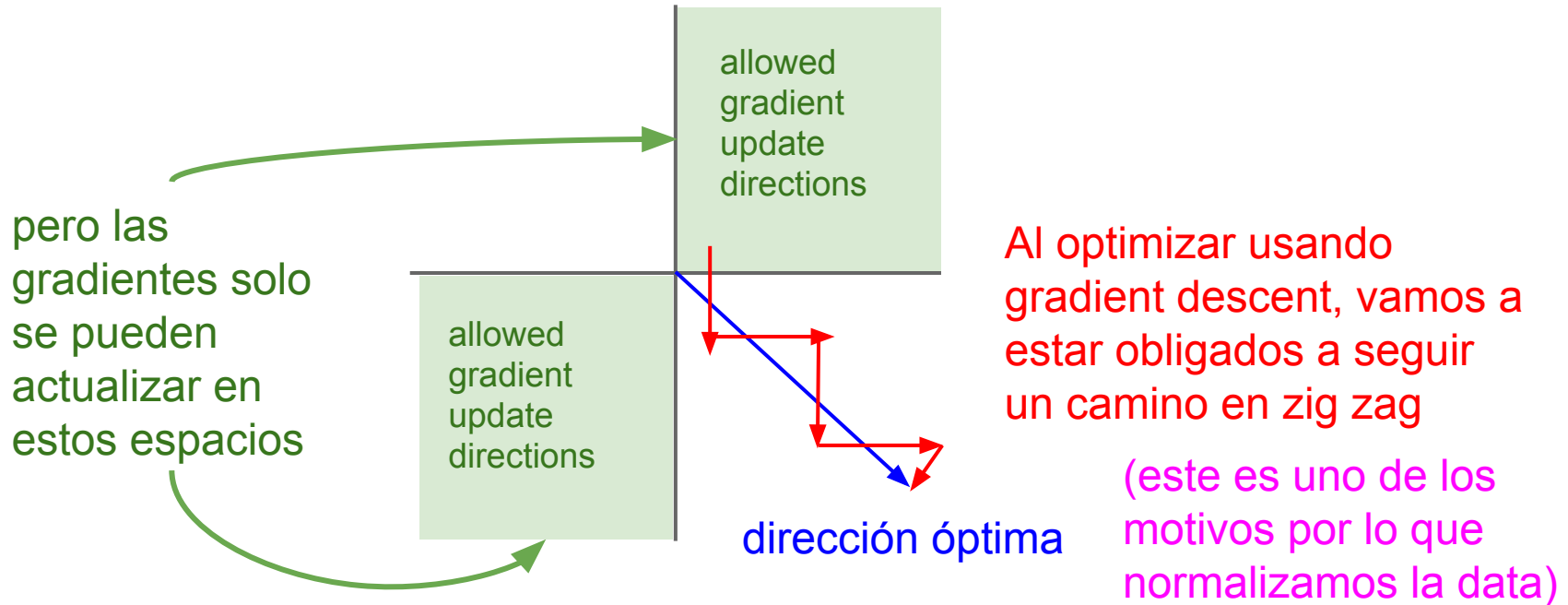
$$w_1^T x = w_2^T x = 1$$

**Se regulariza la matriz de pesos, pero no los bias.*

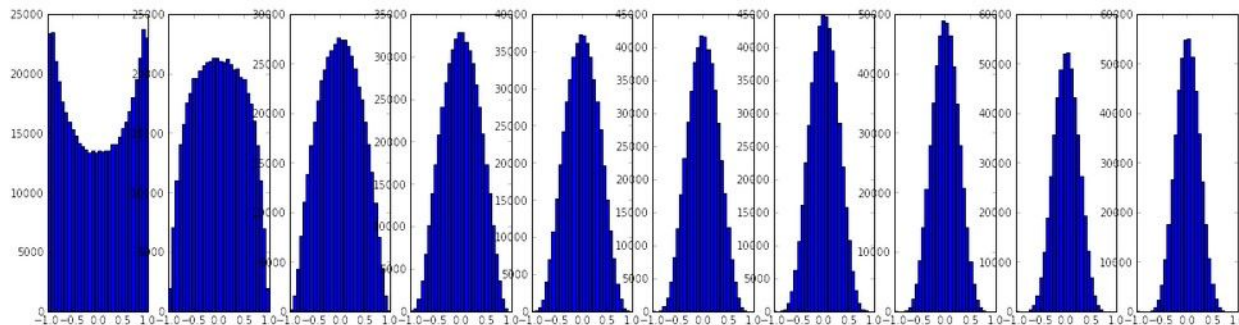
Batch Normalization

Cuando el input (x) de una neurona siempre es positivo, las gradientes van a ser todas positivas o todas negativas.

Espacio de optimización de la gradiente:



Batch Normalization



Situación ideal

¿Cómo? Normalizamos las activaciones de cada capa.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
by Sergey Ioffe, Christian Szegedy 2015

Batch Normalization

1. Normalizamos las activaciones:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization

1. Normalizamos las activaciones:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

2. Mitigamos el efecto de la normalización:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Es posible que la red aprenda los valores::

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

Batch Normalization


1. Normalizamos las activaciones:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

2. Mitigamos el efecto de la normalización:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Esto anularía el efecto del batch normalization



Es posible que la red aprenda los valores::

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

Batch Normalization

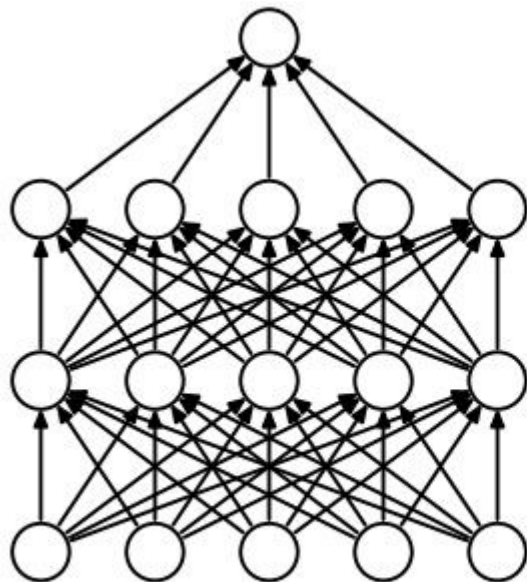
Al usar una capa de batch normalization, es importante no usar bias en la capa anterior. Ej:

```
model.add(Dense(32, activation='relu', use_bias=False))  
model.add(BatchNormalization())
```

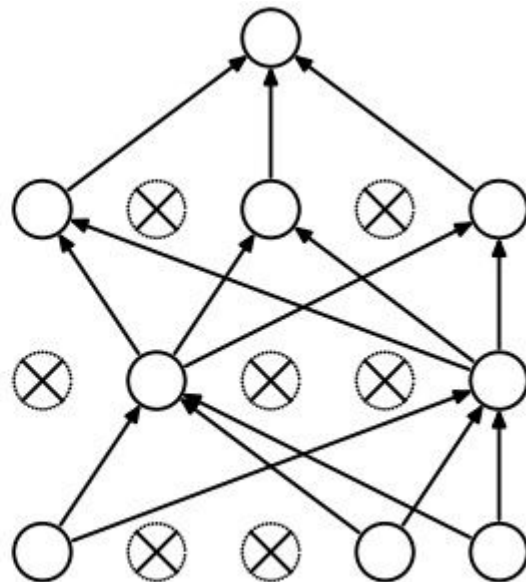
**La capa de batch normalization es la que se va a encargar de aplicar el bias.*

Dropout

Dropout



(a) Standard Neural Net



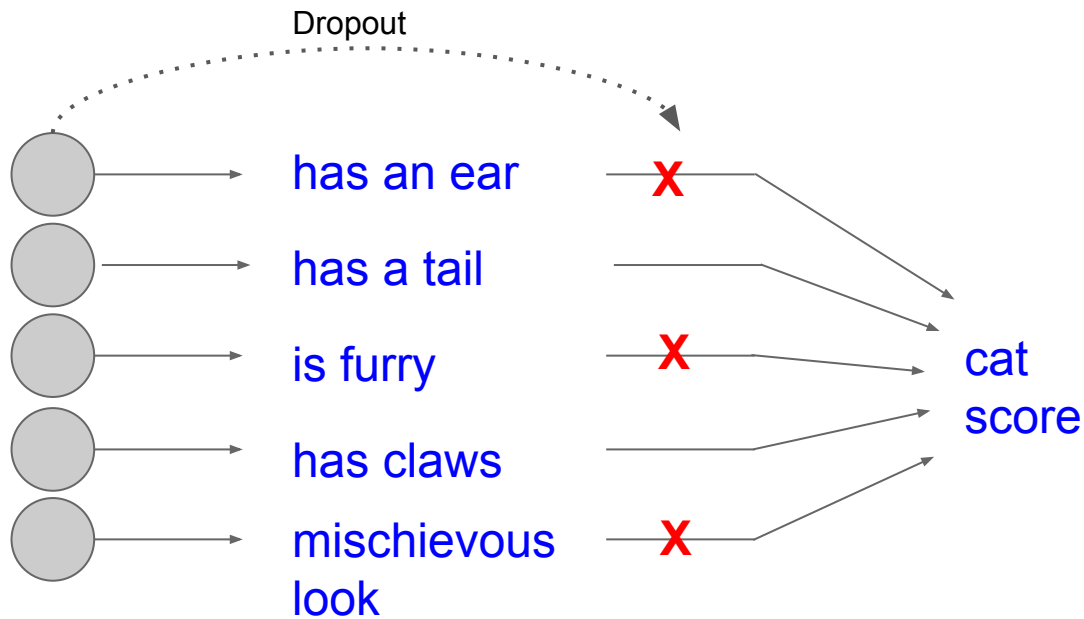
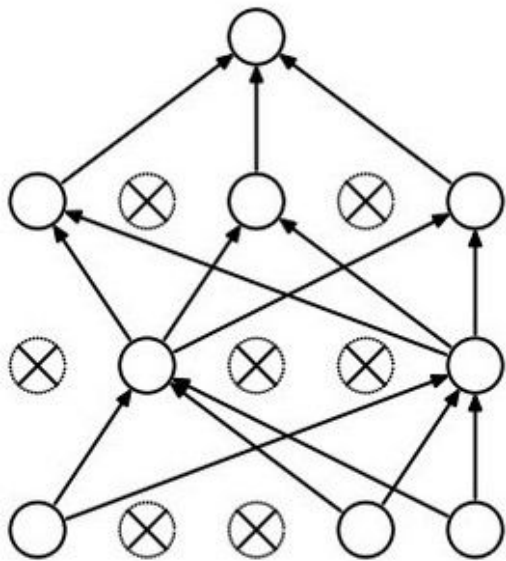
(b) After applying dropout.

[Srivastava et al., 2014]

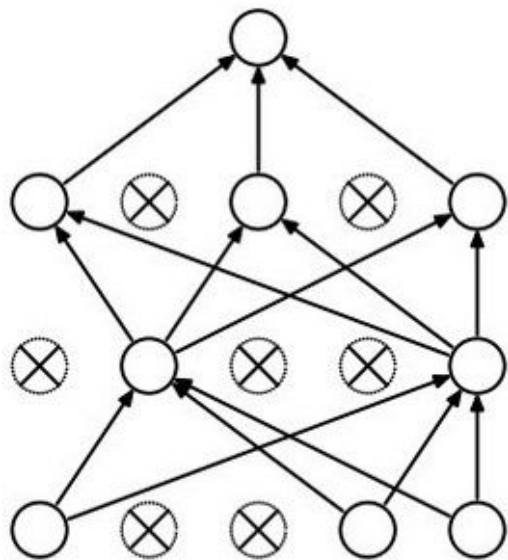
Dropout: A Simple Way to Prevent Neural Networks from Overfitting

by Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov 2014

Dropout



Dropout




Se puede interpretar dropout como un ensemble.

La red va a observar un conjunto distinto de características en cada batch de entrenamiento.

Y para hacer las predicciones se usan todas las características.

Regularización

- Weight regularization

$$L + \lambda R(W)$$


```
from keras import regularizers  
Dense(100, activation='relu', kernel_regularizer=regularizers.l2(0.01))
```

- Batch normalization

```
from keras.layers import BatchNormalization  
model.add(Dense(32, activation='relu', use_bias=False))  
model.add(BatchNormalization())
```

- Dropout

```
from keras.layers import Dropout  
model.add(Dropout(0.5))  
model.add(Dense(100, activation='relu'))
```

Optimización

Optimización

- Simple gradient descent update.
- Momentum update.
- Nesterov momentum update.
- Adagrad update.
- RMSProp update.
- Adam update.

Optimización: Mini-batch Gradient descent

Iterar:

1. **Sample**: obtener una muestra de la data.
2. **Forward**: obtener la pérdida (Loss).
3. **Backprop**: calcular las gradientes.
4. **Actualizar** los parámetros del modelo.

```
for batch in data:
    L = forward(batch)
    dw = backward()
    w -= lr*dw
```

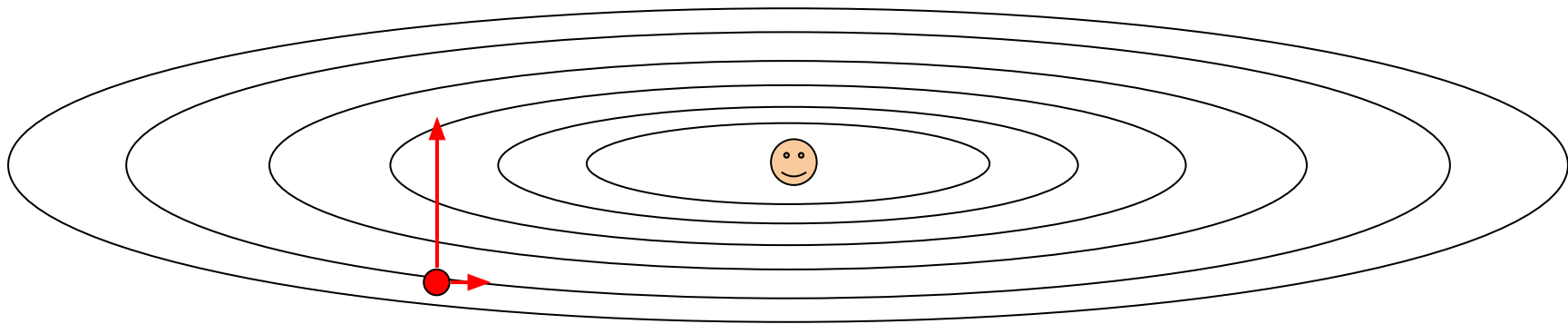
1 Sample
2 Forward
3 Backprop
4 Actualizar

Simple gradient descent update

```
# Simple gradient descent update  
...  
    w -= lr * dw
```

Simple gradient descent update

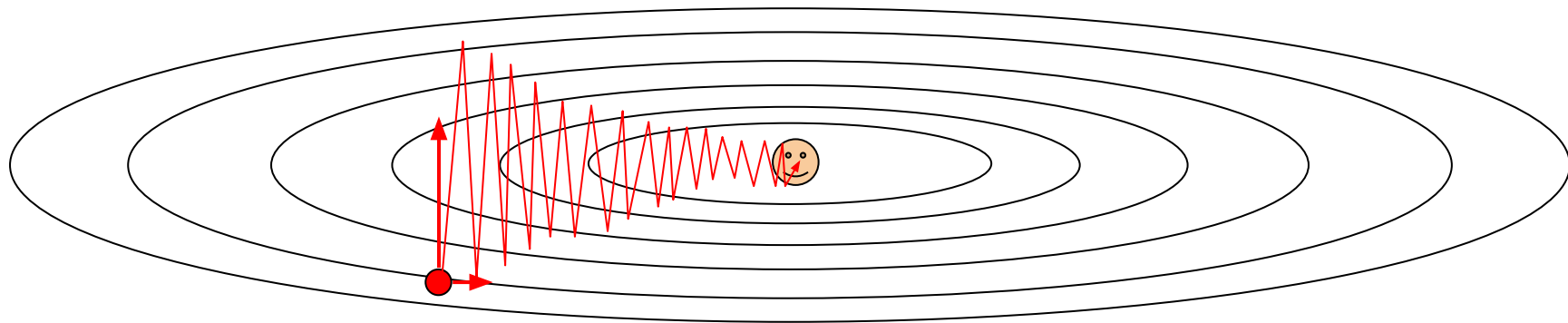
Supongamos una función de pérdida que resulta en gradientes verticales variables con valores altos y gradientes horizontales estables con valores bajos.



¿Qué trayectoria tomará la optimización?

Simple gradient descent update

Supongamos una función de pérdida que resulta en gradientes verticales variables con valores altos y gradientes horizontales estables con valores bajos.

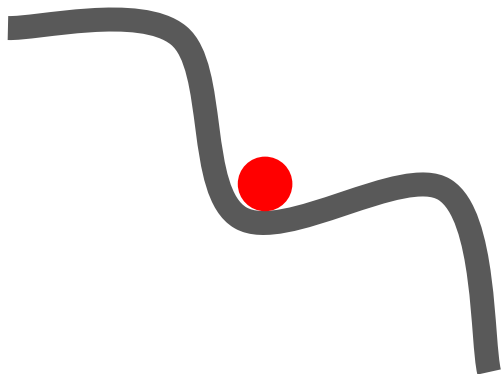


¿Qué trayectoria tomará la optimización?

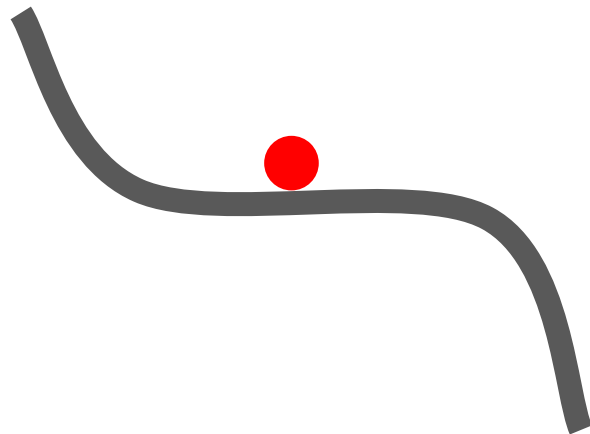
Avanzará lentamente en la dirección horizontal mientras oscila en la dirección vertical.

Simple gradient descent update

Otros problemas:



Local minima



Saddle points

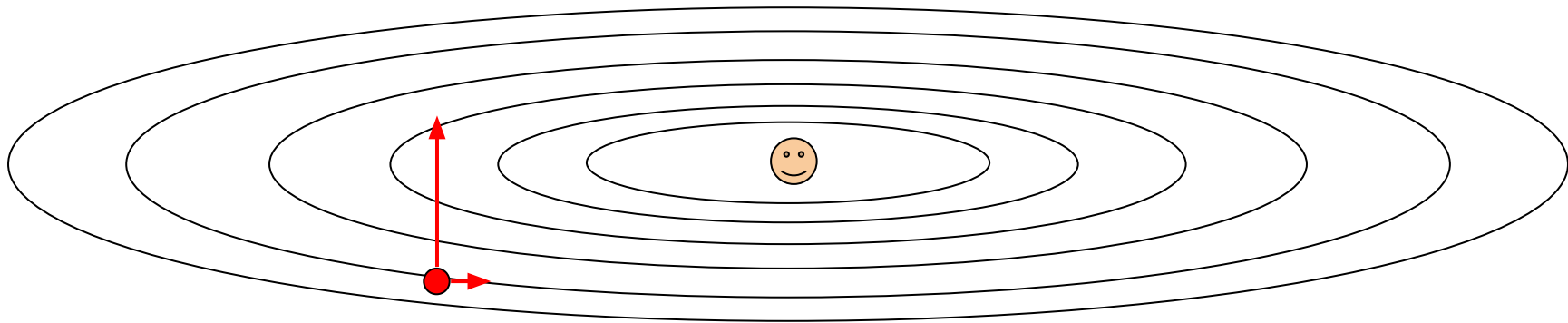
Momentum update

```
# Simple gradient descent update
...
w -= lr * dw
```

[illegible]

Momentum update

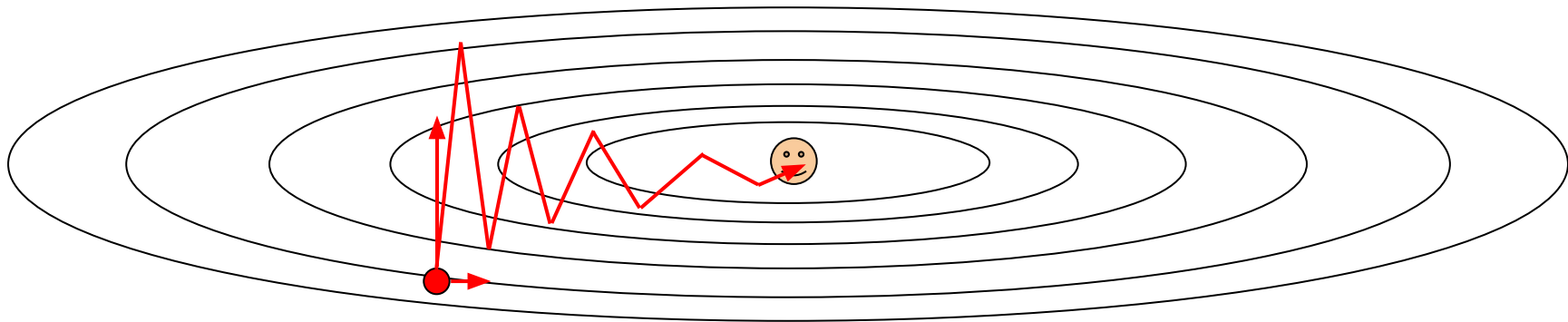
Supongamos una función de pérdida que resulta en gradientes verticales variables con valores altos y gradientes horizontales estables con valores bajos.



Ahora, ¿Qué trayectoria tomará la optimización?

Momentum update

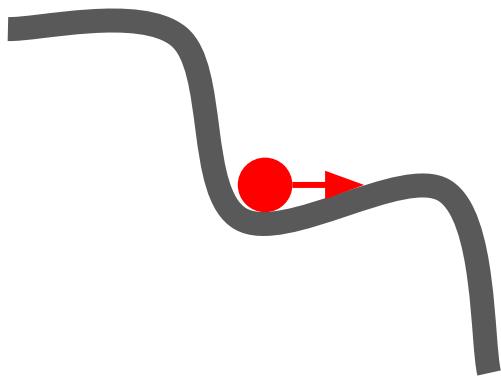
Supongamos una función de pérdida que resulta en gradientes verticales variables con valores altos y gradientes horizontales estables con valores bajos.



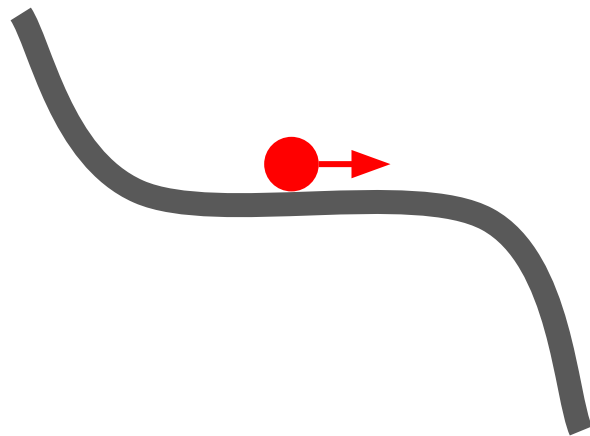
Ahora, ¿Qué trayectoria tomará la optimización?

Momentum aumenta para las dimensiones cuyos gradientes apuntan en las mismas direcciones y reduce las dimensiones cuyos gradientes cambian de dirección. Como resultado, obtenemos una convergencia más rápida y una oscilación reducida.

Momentum update

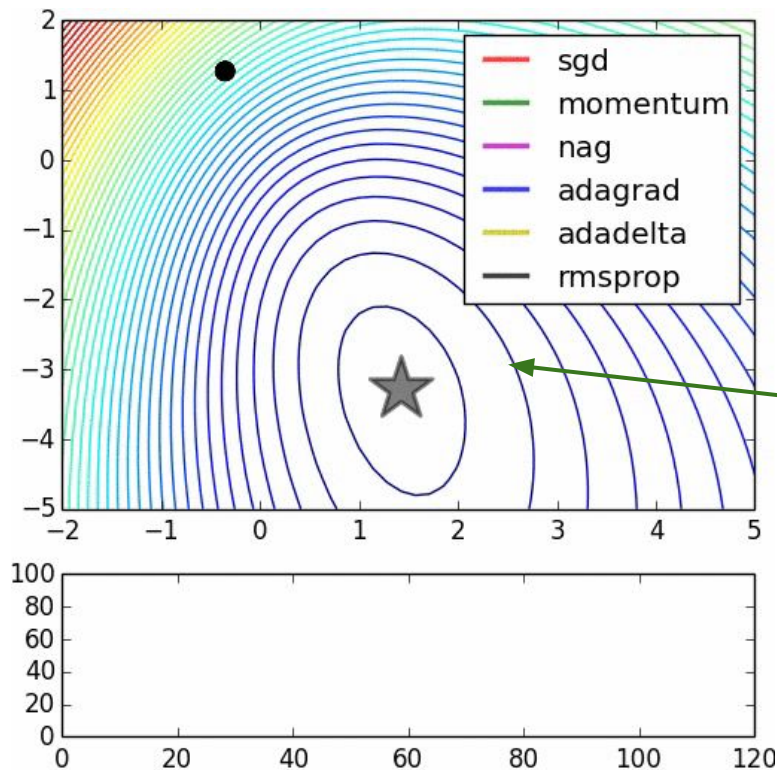


Local minima



Saddle points

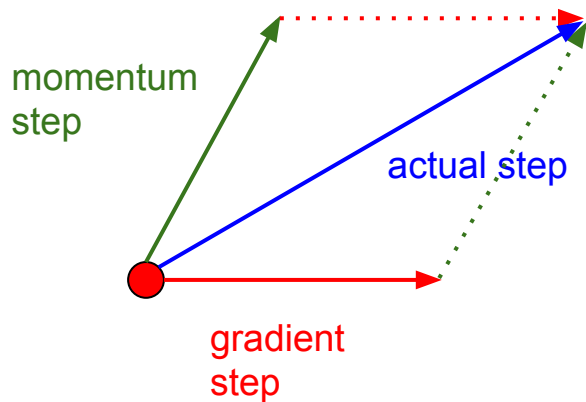
Momentum update



Momentum sobrepasa el target en un inicio, pero llega a converger más rápido que SGD.

Nesterov momentum update

```
# Momentum update  
...  
v = mu*v + lr*dw  
w -= v
```



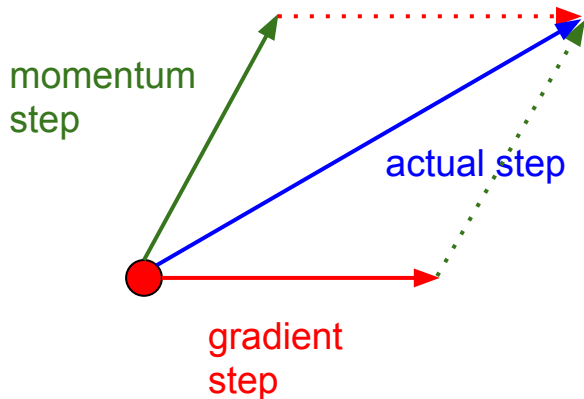
Nesterov momentum update

```
# Momentum update
```

```
...
```

```
v = mu*v + lr*dw
```

```
w -= v
```



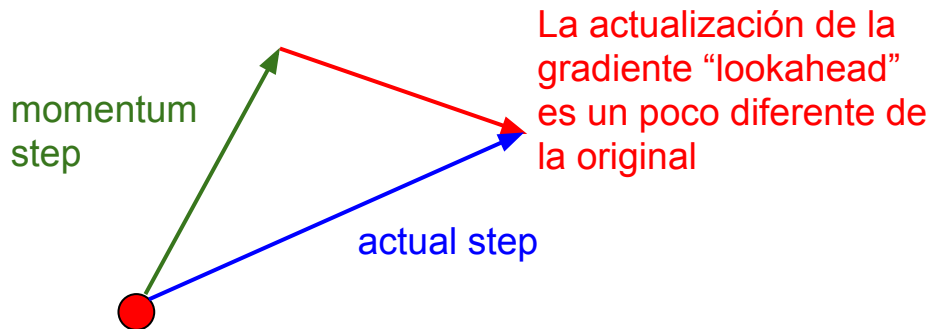
```
# Momentum update
```

```
...
```

```
w_ahead = w + mu*v
```

```
v = mu*v + lr*dw_ahead
```

```
w -= v
```



Nesterov momentum update

```
# Momentum update
...
w_ahead = w + mu*v
v = mu*v + lr*dw_ahead
w -= v
```

Pero el backpropagation nos da las
gradientes en terminos de w...

Nesterov momentum update

```
# Momentum update
```

```
...
```

```
    w_ahead = w + mu*v
```

```
    v = mu*v + lr*dw_ahead
```

```
    w -= v
```

```
# Rewrite en términos de dw
```

```
...
```

```
    v_prev = v
```

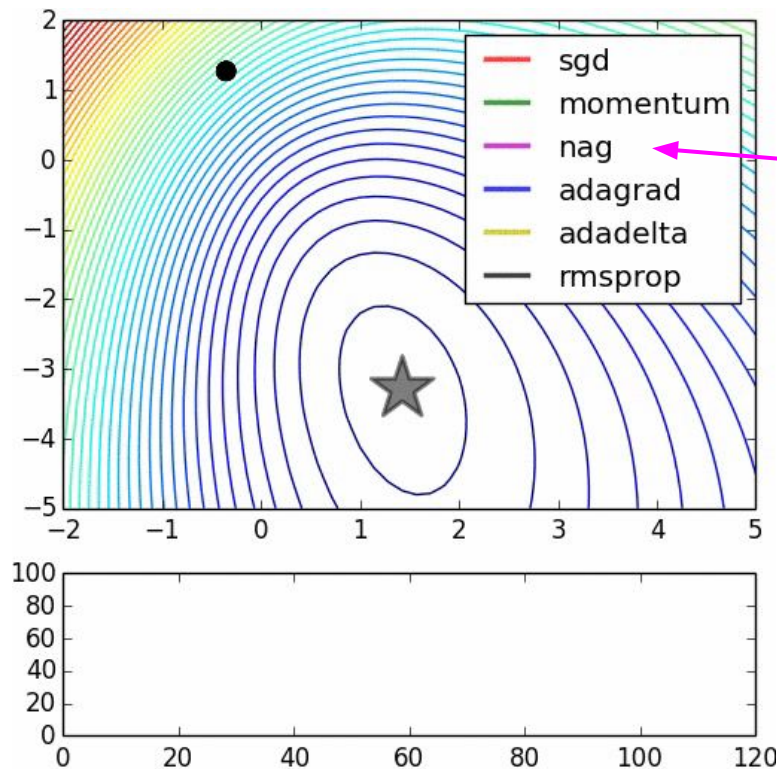
```
    v = mu*v + lr*dw
```

```
    w -= mu*v_prev + (1 + mu)*v
```

Advances in Optimizing Recurrent Networks

by Bengio, Nicolas Boulanger-Lewandowski and Razvan Pascanu 2012

Nesterov momentum update



nag =
Nesterov
Accelerated
Gradient

Adagrad update

```
# Adagrad update
...
cache += dx**2
w -= lr * dx / (np.sqrt(cache) + 1e-7)
```

Se escalan las gradientes basándose en la suma histórica de cuadrados de cada dimensión.

Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

by John Duchi, Elad Hazan, Yoram Singer 2011

Adagrad update

```
# Adagrad update
...
cache += dx**2
w -= lr * dx / (np.sqrt(cache) + 1e-7)
```



¿Qué sucede con cada una de las dimensiones?

Adagrad update

```
# Adagrad update
...
cache += dx**2
w -= lr * dx / (np.sqrt(cache) + 1e-7)
```



¿Qué sucede con cada una de las dimensiones?

La dimensión vertical se divide con valores mayores: se desacelera el aprendizaje.

La dimensión horizontal se divide con valores menores: se acelera el aprendizaje.

Adagrad update

```
# Adagrad update
...
cache += dx**2
w -= lr * dx / (np.sqrt(cache) + 1e-7)
```



¿Qué pasa cuando entrenamos por un tiempo extendido?

Adagrad update

```
# Adagrad update
...
cache += dx**2
w -= lr * dx / (np.sqrt(cache) + 1e-7)
```



¿Qué pasa cuando entrenamos por un tiempo extendido?
El cache se puede hacer muy grande, degradando el aprendizaje a ~ 0 .

RMSProp update

```
# Adagrad update
```

```
...
```

```
    cache += dx**2
```

```
    w -= lr * dx / (np.sqrt(cache) + 1e-7)
```

```
# RMSProp update
```

```
...
```

```
    cache += decay_rate*cache + (1 - decay_rate)*(dx**2)
```

```
    w -= lr * dx / (np.sqrt(cache) + 1e-7)
```

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
 - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight
$$MeanSquare(w, t) = 0.9 MeanSquare(w, t-1) + 0.1 \left(\frac{\partial E}{\partial w}(t) \right)^2$$
- Dividing the gradient by $\sqrt{MeanSquare(w, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introducido en una diapositiva de Geoff Hinton (Coursera class, lecture 6)

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

Adam update

```
# RMSProp update
```

```
...
```

```
    cache += decay_rate*cache + (1 - decay_rate)*(dx**2)
    w -= lr * dx / (np.sqrt(cache) + 1e-7)
```

```
# Adam update (incompleto)
```

```
...
```

```
    m = beta1*m + (1-beta1)*dw          # ~momentum
    v = beta2*v + (1-beta2)*(dx**2)      # ~RMSProp
    w -= lr * m / (np.sqrt(v) + 1e-7)
```


Adam update

```
# Adam update (incompleto)
...
m = beta1*m + (1-beta1)*dw      # ~momentum
v = beta2*v + (1-beta2)*(dx**2)  # ~RMSProp
w -= lr * m / (np.sqrt(v) + 1e-7)
```

¿Qué pasa en la primera iteración?

Adam update

```
# Adam update (incompleto)
...
m = beta1*m + (1-beta1)*dw      # ~momentum
v = beta2*v + (1-beta2)*(dx**2)  # ~RMSProp
w -= lr * m / (np.sqrt(v) + 1e-7)
```

¿Qué pasa en la primera iteración?

- 1) β_1 y β_2 son valores de decay, usualmente ~ 0.9 .
- 2) Dado que v se inicializa en 0, el primer v seria un numero muy pequeño.
- 3) Al actualizar los pesos, dividiendo entre v (que es un número muy pequeño), da como resultado una actualización de pesos inusualmente grande.

Adam update

```
# Adam update (completo)
...
m = beta1*m + (1-beta1)*dw      # ~momentum
v = beta2*v + (1-beta2)*(dx**2) # ~RMSProp
mb = m / (1-beta1**t)           # bias correction
vb = v / (1-beta2**t)           # bias correction
w -= lr * mb / (np.sqrt(vb) + 1e-7)
```

El bias correction compensa el hecho de que m y v se inicializan en 0, y necesitan de algunas iteraciones para funcionar correctamente.

Adam: A Method for Stochastic Optimization
by Diederik Kingma, Jimmy Ba 2014

Optimización: Keras code

```
from keras.optimizers import SGD, Adagrad, RMSprop, Adam

# Simple gradient descent update
SGD(lr=0.01)

# Momentum update
SGD(lr=0.01, momentum=0.9)

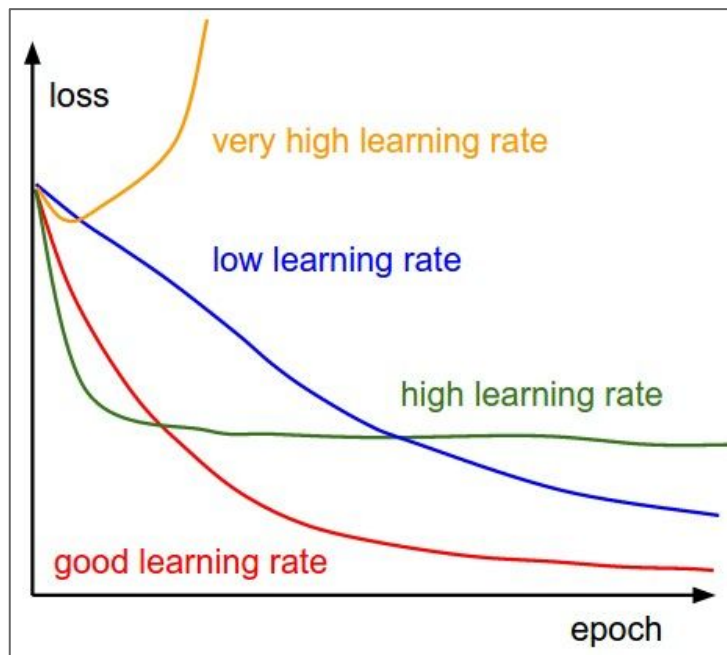
# Nesterov momentum update
SGD(lr=0.01, momentum=0.9, nesterov=True)

# Adagrad update
Adagrad(lr=0.01)

# RMSProp update
RMSprop(lr=0.001, rho=0.9)

# Adam update
Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

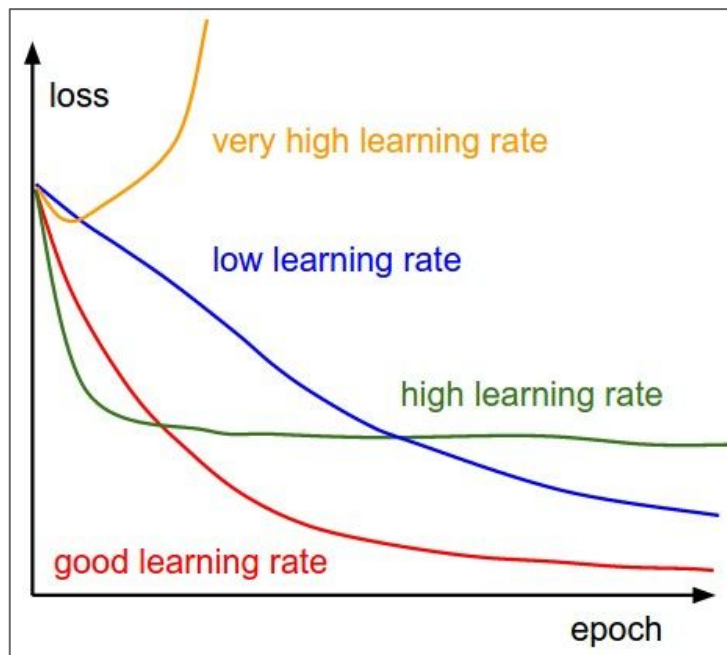
Optimización



Todos los algoritmos de optimización que hemos visto tienen como hiperparámetro al learning rate.

De estos, ¿Qué learning rate deberíamos usar?

Optimización

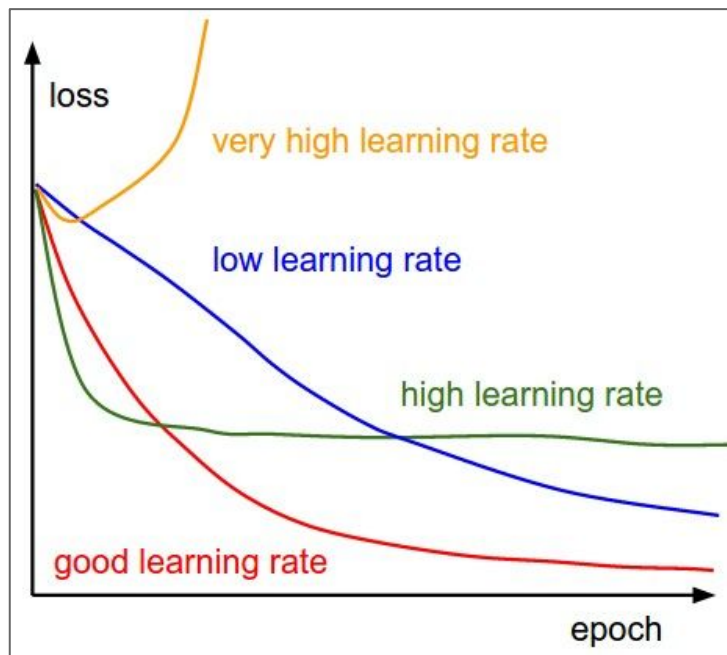


Todos los algoritmos de optimización que hemos visto tienen como hiperparámetro al learning rate.

De estos, ¿Qué learning rate deberíamos usar?

Ninguno estático, el learning rate debería variar con el tiempo.

Optimización



Learning rate decay:

step decay:

El learning rate disminuye a la mitad cada cierto número de iteraciones.

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

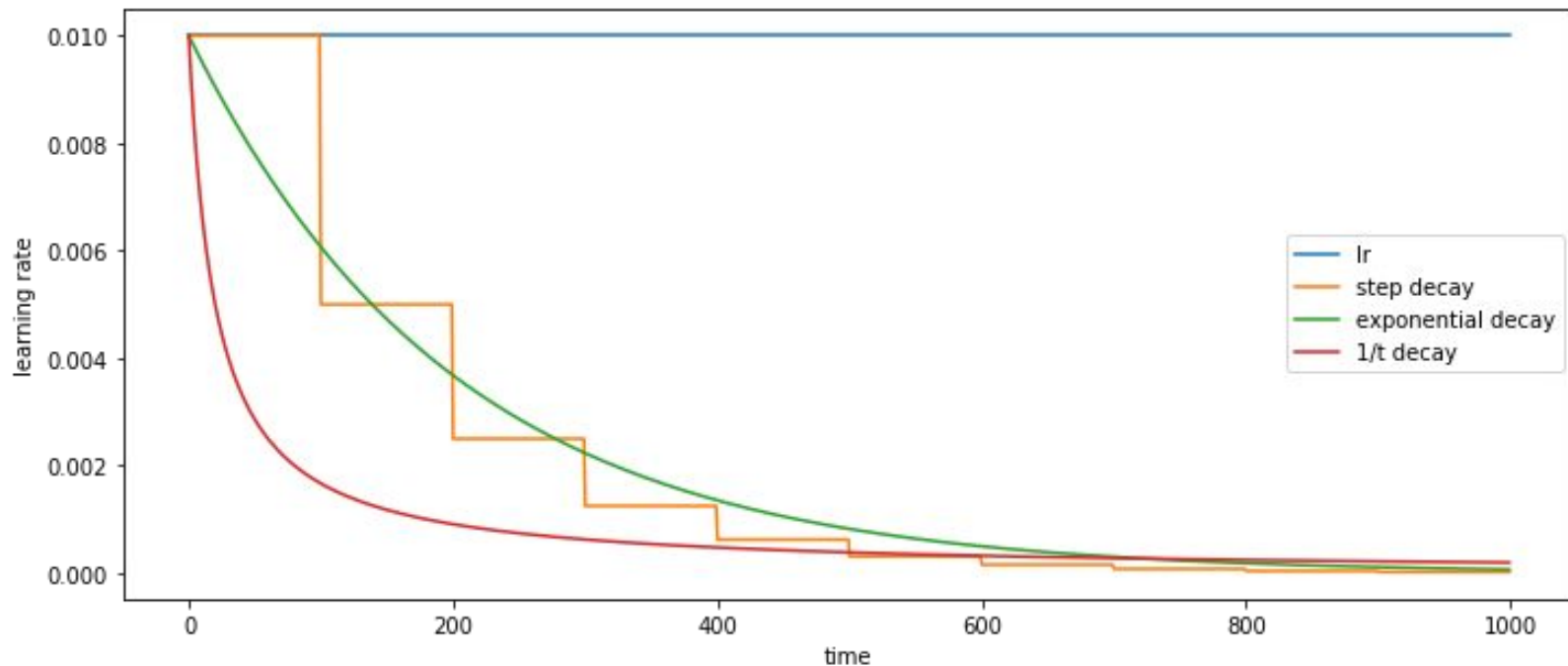
Time based decay:

$$\alpha = \alpha_0 / (1 + kt)$$

$$\text{step decay} = \frac{lr_0}{2^{(time//100)}}$$

$$\text{exponential decay} = lr_0 (e^{-0.005(time)})$$

$$\text{time decay} = \frac{lr_0}{1+0.05(time)}$$



Optimización: keras code

```
# En keras parte de los parámetros del optimizador es "decay"  
# Este parámetro es el K en time based decay  
SGD(lr=0.01, decay=0.05)
```

$$time\ decay = \frac{lr_0}{1+K(time)}$$

Optimización: keras code

```
# En keras parte de los parámetros del optimizador es "decay"  
# Este parámetro es el K en time based decay  
SGD(lr=0.01, decay=0.05)
```

$$time\ decay = \frac{lr_0}{1+K(time)}$$

¿Y los otros tipos de decay?

Optimización: keras callbacks

```
# Uno de los parámetros del método fit, para entrenar un modelo, es
# "callbacks", esta es una lista de instancias de keras.callbacks
model.fit(x_train, y_train, epochs=10,
          callbacks=[callback1, callback2, ...])

# Estos callbacks pueden ejecutar métodos antes y después de un batch
# o una época.

# Para variar el learning rate en base a la época usamos un callback
# predefinido: "LearningRateScheduler"

from keras.callbacks import LearningRateScheduler
```

Optimización: keras callbacks

Ej:

$$\text{exponential decay} = lr_0(e^{-0.005(time)})$$

```
from keras.callbacks import LearningRateScheduler

lr = 0.1
def exp_decay(epoch):
    return lr*np.exp(-0.005*epoch)

sched = LearningRateScheduler(exp_decay, verbose=1)

model.fit(x_train, y_train, epochs=10, callbacks=[sched])
```

Optimización: keras callbacks

Ej:

$$\text{exponential decay} = lr_0(e^{-0.005(\text{time})})$$

Epoch 1/10

Epoch 00001: LearningRateScheduler reducing learning rate to 0.1
100/100 [=====] - 0s 71us/step - loss: 1.1348
Epoch 2/10

Epoch 00002: LearningRateScheduler reducing learning rate to 0.09950124791926823
100/100 [=====] - 0s 88us/step - loss: 1.1320
Epoch 3/10

Epoch 00003: LearningRateScheduler reducing learning rate to 0.09900498337491681
100/100 [=====] - 0s 56us/step - loss: 1.1298
Epoch 4/10

Epoch 00004: LearningRateScheduler reducing learning rate to 0.09851119396030628
100/100 [=====] - 0s 50us/step - loss: 1.1270

Optimización: Técnicas avanzadas

Optimización: Técnicas avanzadas

- Learning rate finder
- Cyclical learning rates
- Snapshot ensembles

Learning rate finder

Idea:

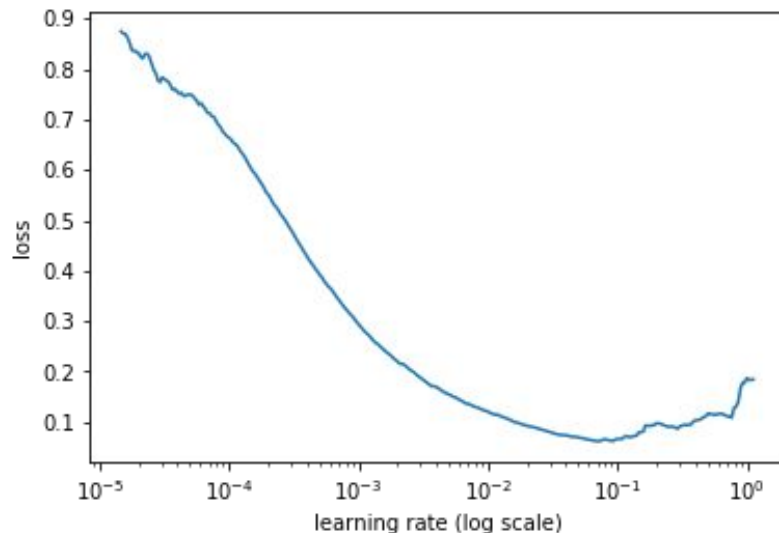
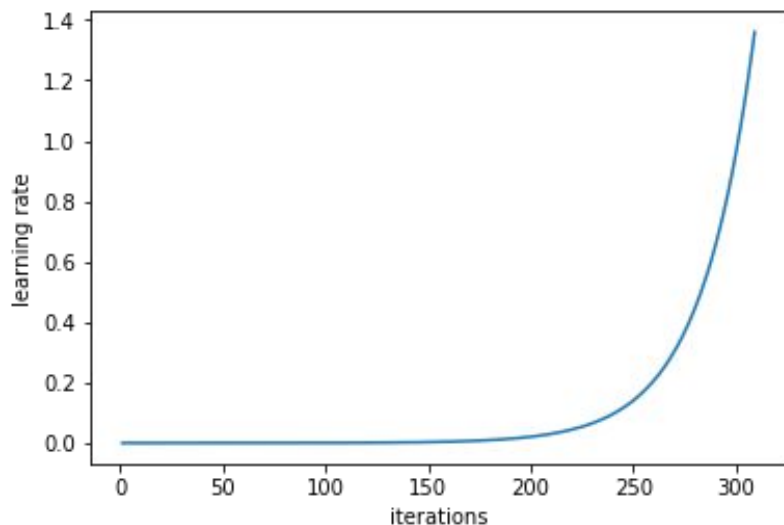
1. Comenzamos con un learning rate muy pequeño.
2. En cada batch incrementamos el valor del learning rate.
3. Observamos el comportamiento del aprendizaje.

Cyclical Learning Rates for Training Neural Networks

by Leslie N. Smith 2017

Learning rate finder

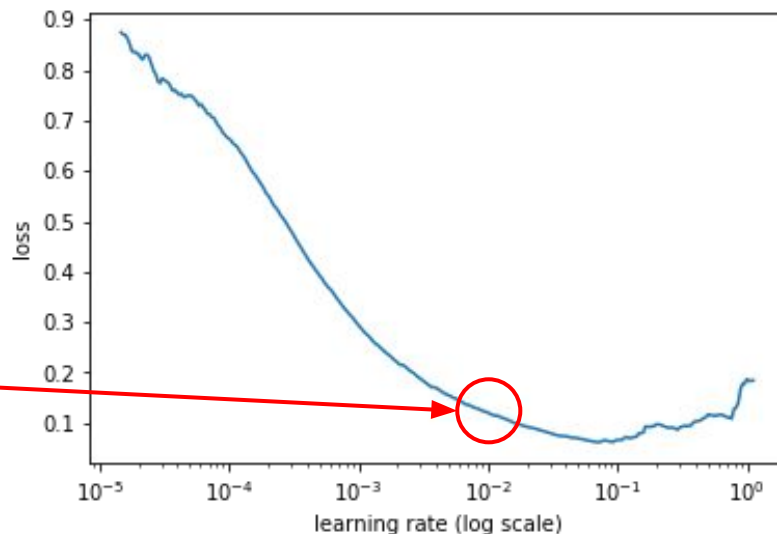
1. Comenzamos con un learning rate muy pequeño.
2. En cada batch incrementamos el valor del learning rate.
3. Observamos el comportamiento del aprendizaje.



Learning rate finder

1. Comenzamos con un learning rate muy pequeño.
2. En cada batch incrementamos el valor del learning rate.
3. Observamos el comportamiento del aprendizaje.

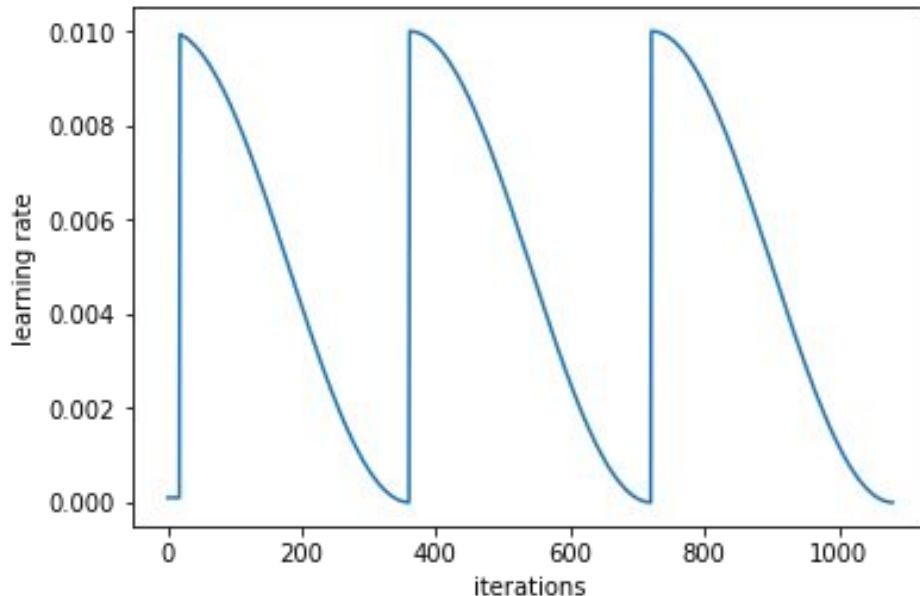
Escogemos un learning rate alto, donde aún se vea mejora en el aprendizaje (ej: 10^{-2})



Cyclical learning rates

La variación agresiva del learning rate permite que el modelo converja rápidamente a una nueva y mejor solución en cada ciclo. Los autores encontraron empíricamente que se requieren entre 2 y 4 veces menos épocas usando esta técnica.

Además, Incrementar el learning rate cada cierto tiempo, ayuda a escapar de mínimos locales.



SGDR: Stochastic Gradient Descent with Warm Restarts

by Ilya Loshchilov, Frank Hutter 2017

Cyclical Learning Rates for Training Neural Networks

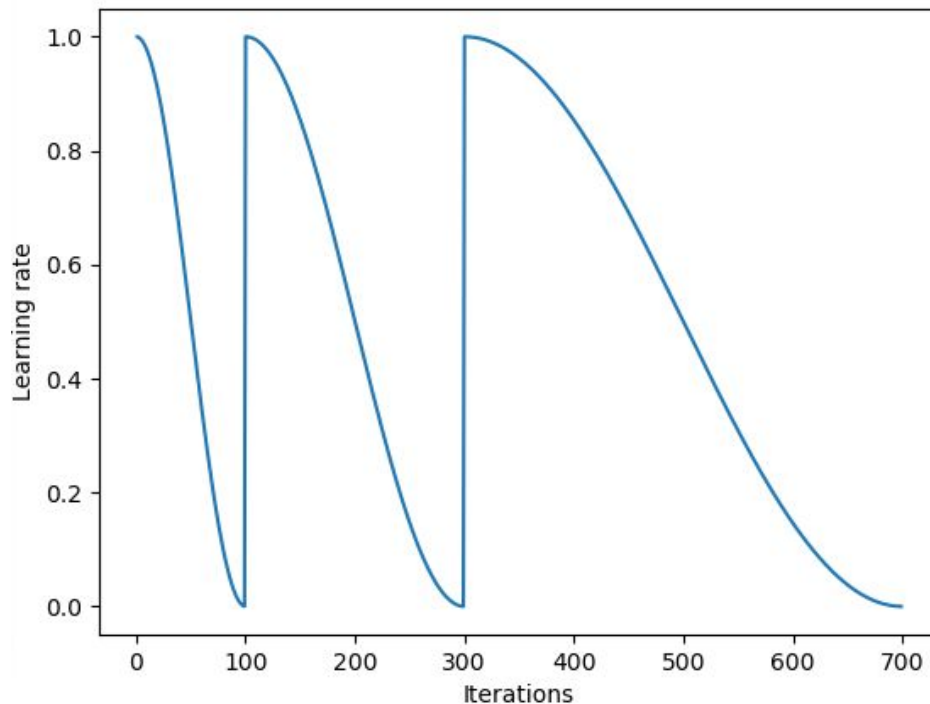
by Leslie N. Smith 2017

Cyclical learning rates

Un parámetro adicional a explorar es la cantidad de épocas que ve cada ciclo.

En este caso se tienen 3 ciclos, donde cada uno ve 1, 2 y 4 épocas respectivamente.

Nota: este método funciona mejor en los algoritmos de optimización que no adaptan las gradientes (SGD, Momentum, Nesterov).

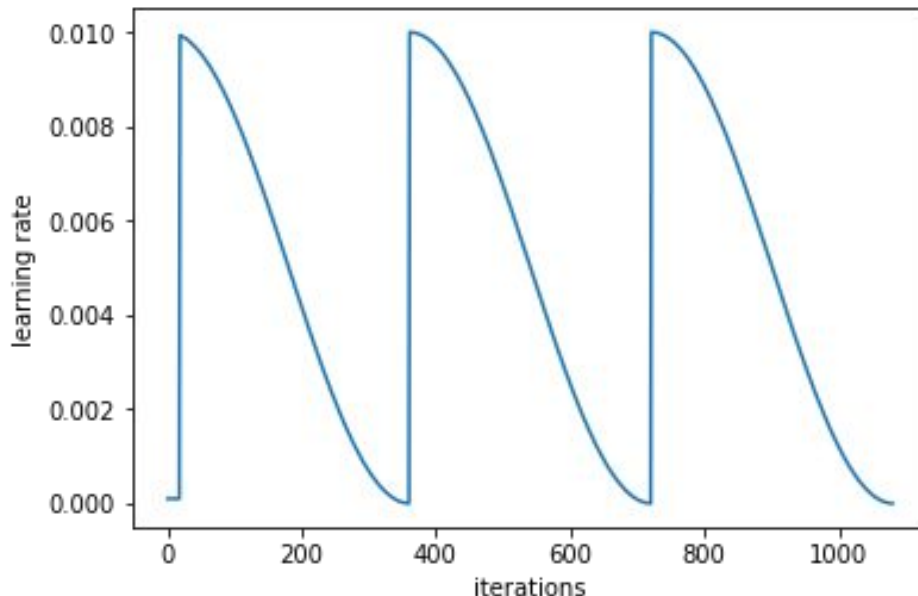


Snapshot ensembles

Las tasas de error luego de cada ciclo son similares, pero tienden a cometer diferentes errores.

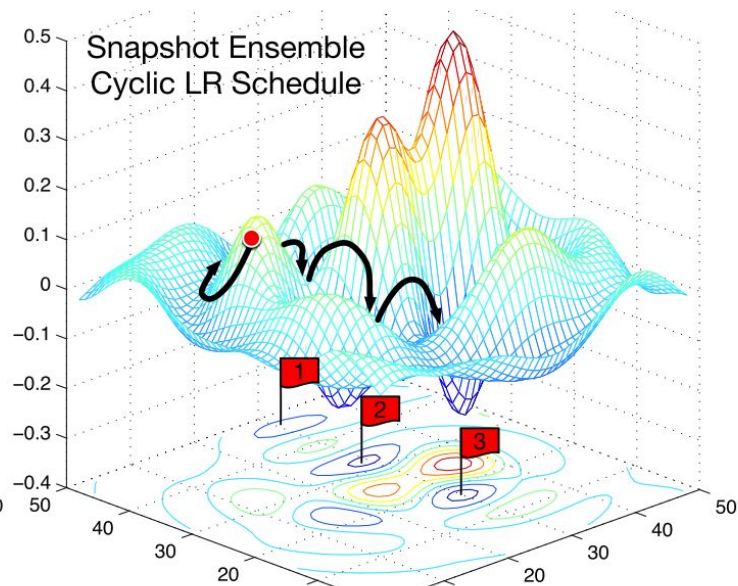
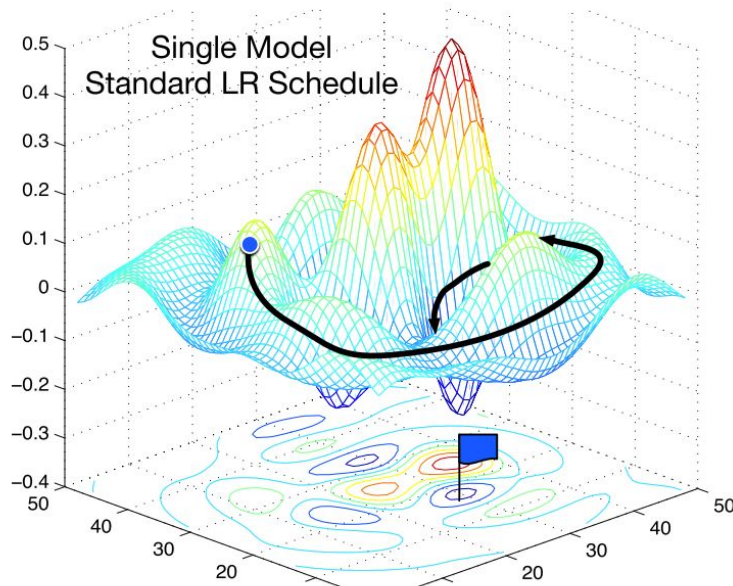
Esta diversidad puede explotarse a través de un **ensemble** de distintas versiones de la red neuronal en distintos puntos mínimos.

Promediar sobre las predicciones de estos modelos conduce a reducciones en las tasas de error.



Snapshot Ensembles: Train 1, get M for free
by Gao Huang, et. al. 2017

Snapshot ensembles



Snapshot Ensembles: Train 1, get M for free
by Gao Huang, et. al. 2017

