



Aprendizaje Automático Profundo

Clase 9 - Tópicos avanzados

Profs: Franco Ronchetti - Facundo Quiroga

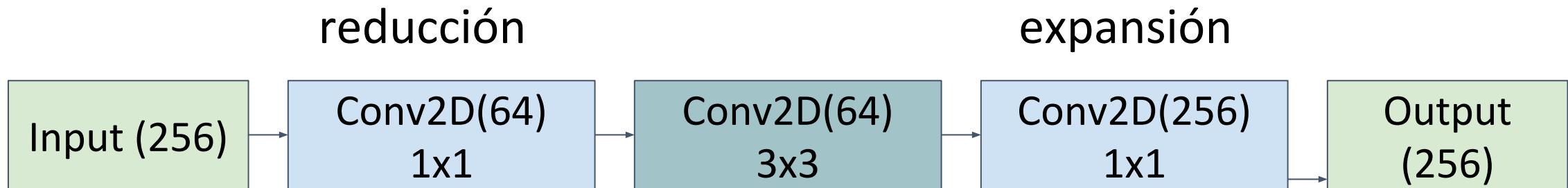


Modelo Residual Network (ResNets)

Redes residuales

Bloques Bottleneck

- Objetivo: menor coste computacional
 - $1 \times 1 \times 256 + 3 \times 3 \times 64 + 1 \times 1 \times 256 = 1088$ parámetros vs $3 \times 3 \times 256 = 2304$



```
def bottleneck(x,n_reduction,n_expansion):  
    x = Conv2D(n_reduction,(1,1),padding="same",activation="relu")  
    x = Conv2D(n_reduction,(3,3),padding="same",activation="relu")  
    x = Conv2D(n_expansion,(1,1),padding="same",activation="relu")  
    return x
```

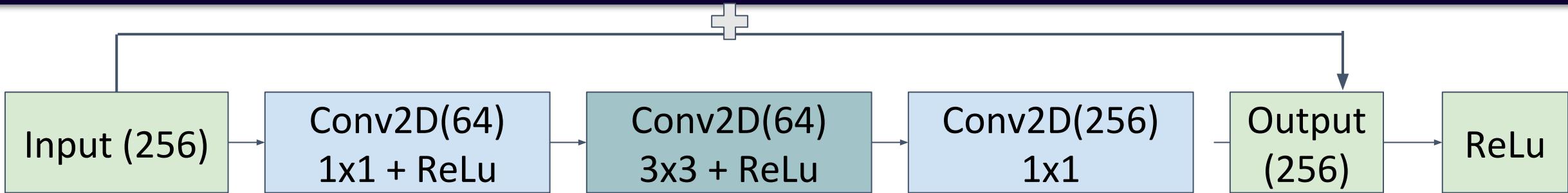
Bloques Residuales

- Bloques residuales
 - Aprenden modificación (vs transformación) de la entrada
 - Al comenzar entrenamiento, la capa es la función identidad
 - La salida es igual a la entrada (mismas dimensiones)
 - Puedo poner muchas más capas
 - En el peor caso, no hacen ninguna modificación.



```
def residual(x):  
    x_mod = Capa(...)(x)  
    return x + xmod
```

Bloques Conv Bottleneck + Residuales



```
def residual_bottleneck(x,n_reduction,n_expansion):  
    x_mod=x  
    x_mod = Conv2D(n_reduction,(1,1),activation="relu")(x_mod)  
    x_mod = Conv2D(n_reduction,(3,3),activation="relu")(x_mod)  
    x_mod = Conv2D(n_expansion,(1,1),activation=None)(x_mod)  
    new_x = Add()([x, x_mod])  
    new_x = Activation("relu")(new_x) # mismas dimensiones  
    return new_x
```

Modelo ResNet (Red Residual) ([paper](#), [notebook](#))

- Primeras en entrenar redes con hasta 152 capas

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			$7 \times 7, 64, \text{stride } 2$		
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Modelo ResNet (Red Residual) ([notebook](#))

- ResNet50

```
def ResNet50(input_shape,classes):  
    input = Input(shape=input_shape)  
    x = Conv2D(64,(7,7),strides=(2,2),padding="same",activation="relu")(input)  
    reductions = [64,128,256,512]  
    blocks      = [3,4,6,3]  
    for i,(block_n,reduction) in enumerate(zip(blocks,reductions)):  
        expansion=reduction*2  
        x = Conv2D(expansion,(3,3),strides=(2,2),padding="same",activation="relu")(x)  
        for j in range(block_n):  
            x = residual_bottleneck(x,reduction,expansion)  
        x = GlobalAveragePooling2D(name="gap")(x)  
        x = Dense(classes,activation="softmax")(x)  
    model = Model(inputs=[input],outputs=[x])  
    return model
```

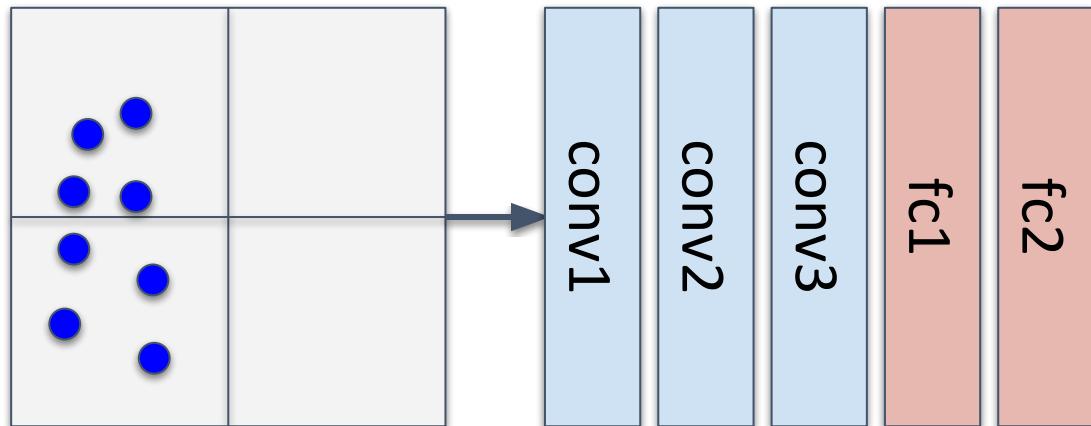
Batch normalization

(normalización por batchs)

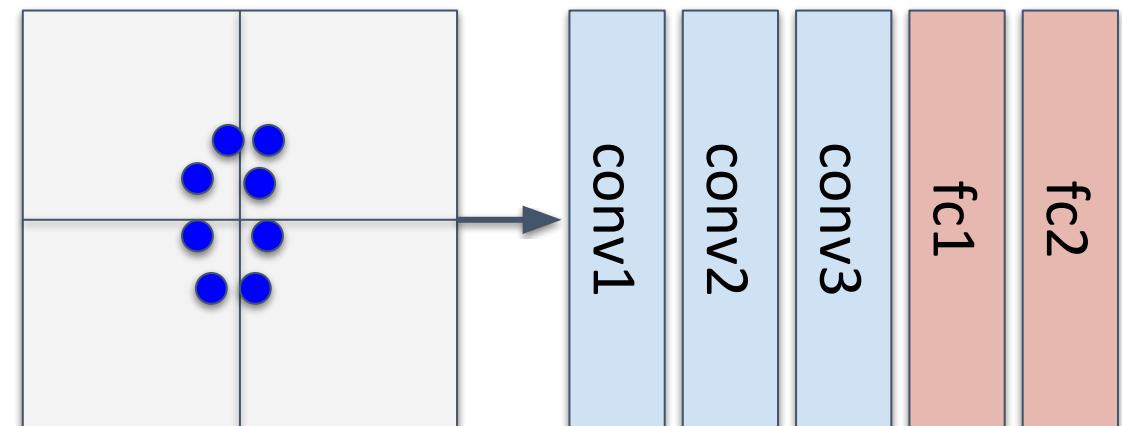
Normalización de la entrada

- ¿Por qué normalizamos los datos?
 - Mejora el aprendizaje
 - Mejora el accuracy
 - Estandariza las magnitudes
 - Comparables a través de modelos/datasets

Sin normalizar



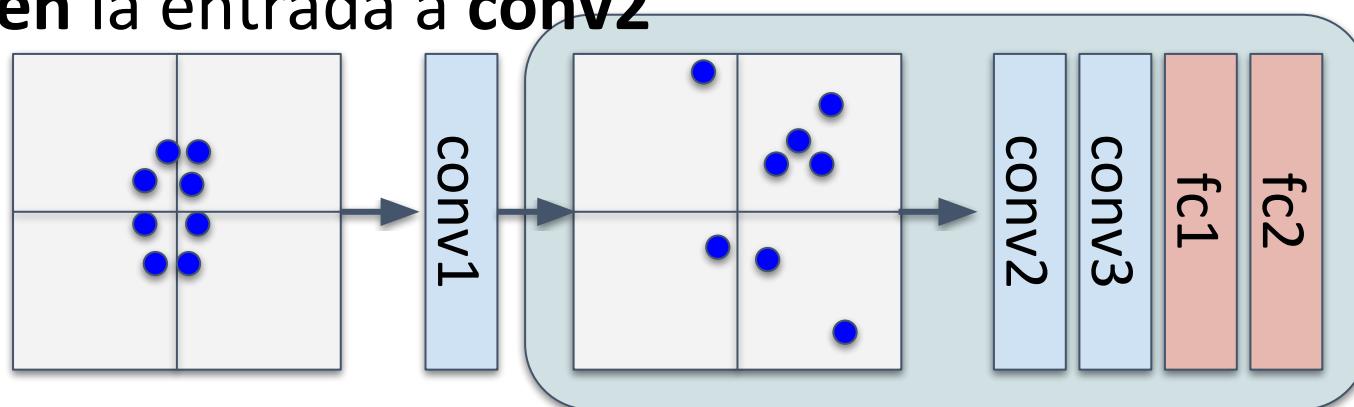
Normalización Z



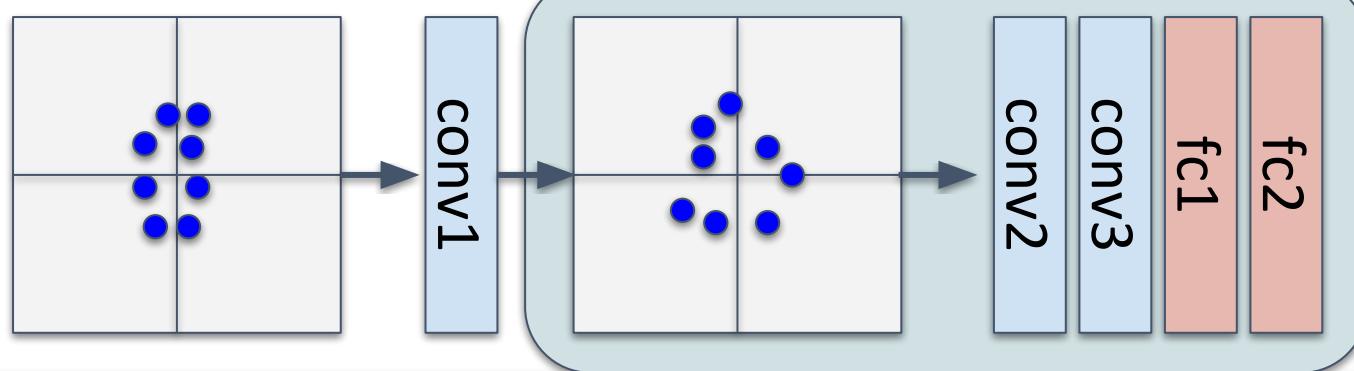
Capa Batch Normalization (BN)

- ¿Qué pasa entre capas (conv1 y conv2)?
- Estructura recursiva de las redes
 - Una red sin su primer capa también es una red
 - Idea: Normalicemos **también** la entrada a **conv2**

- Situación actual



- Situación ideal



Capa Batch Normalization (BN)

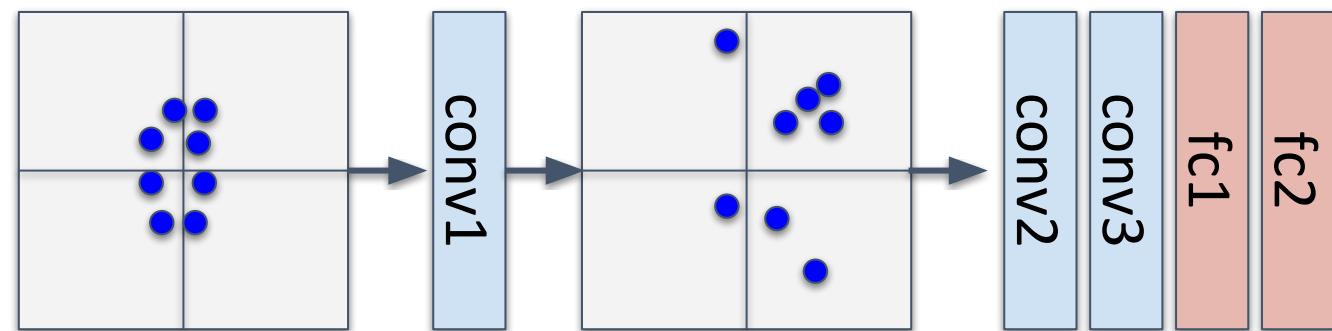
- Problema durante el entrenamiento
 - La salida de conv1 cambia luego de cada batch
 - Cambia el batch y parámetros!

- Batch 1: conv1 tiene pesos

w

- Se actualizan

- $w := w - dw * \alpha$



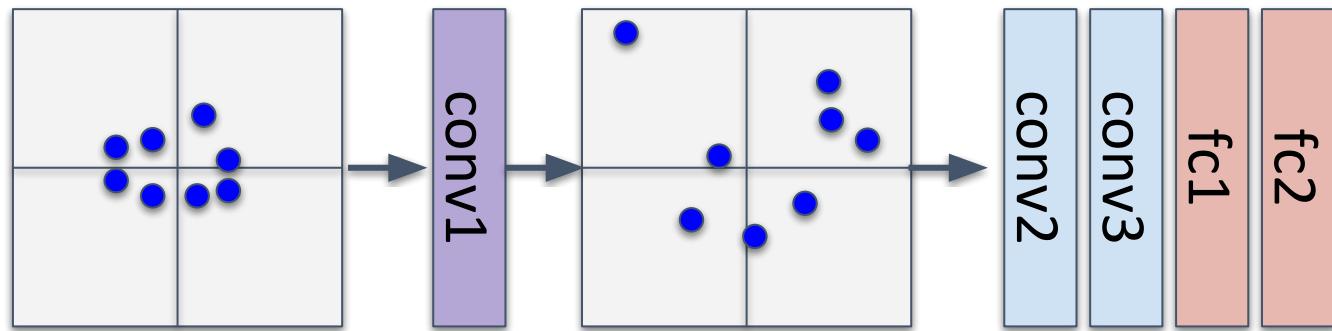
- Batch 2: conv1 tiene pesos

w

- Se actualizan

- $w := w - dw * \alpha$

- (mismo para batch N)

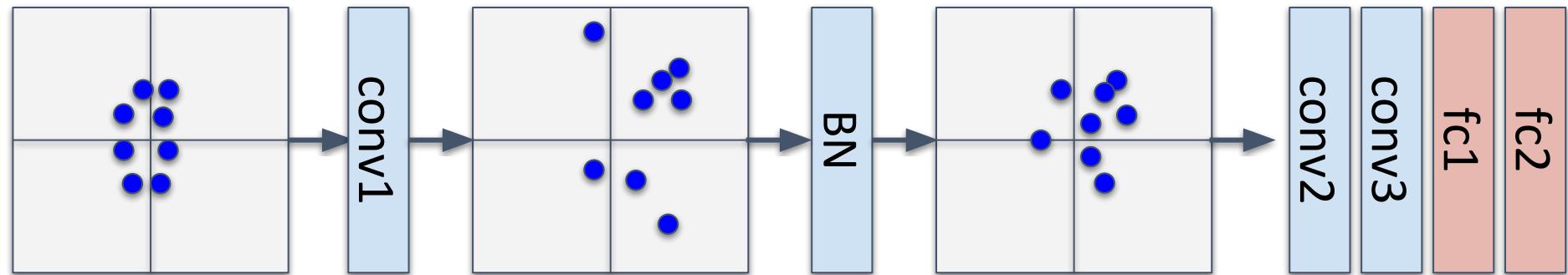


Capa Batch Normalization (BN)

- Solución
 - Normalización Z en cada batch
 - Estimar parámetros (μ y σ) para cada batch

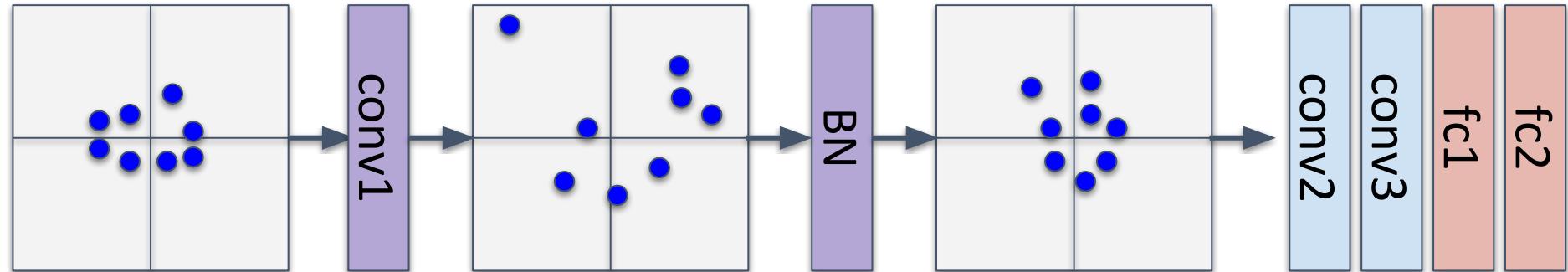
- Batch 1:

- BN calcula μ y σ
- Normaliza



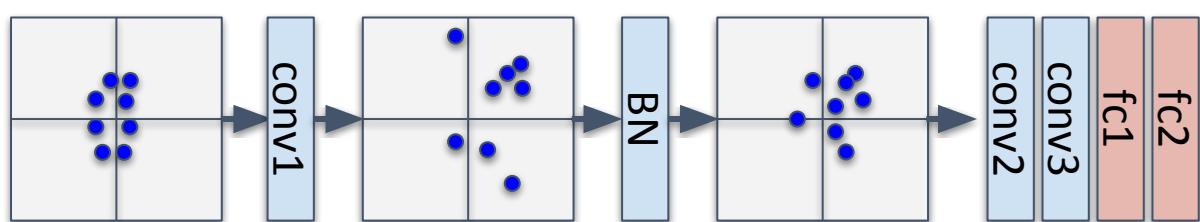
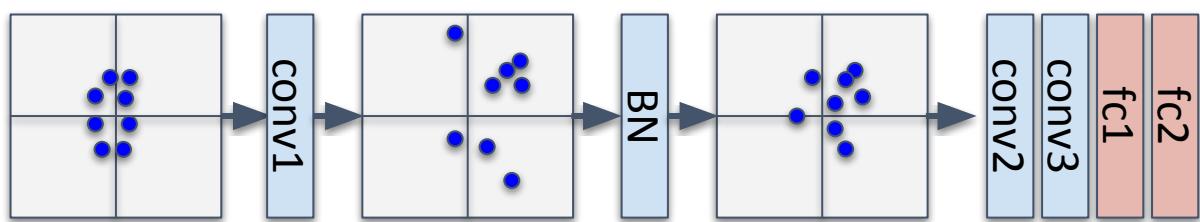
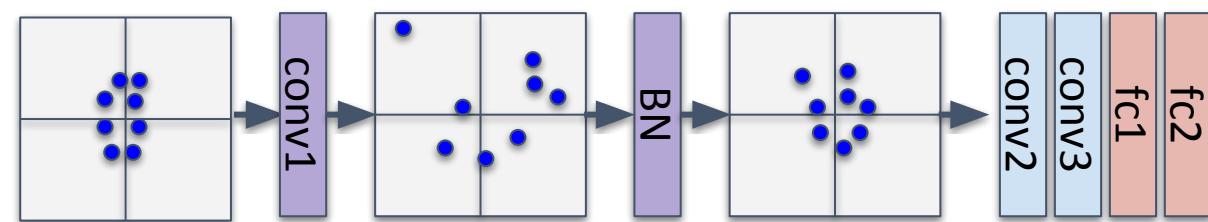
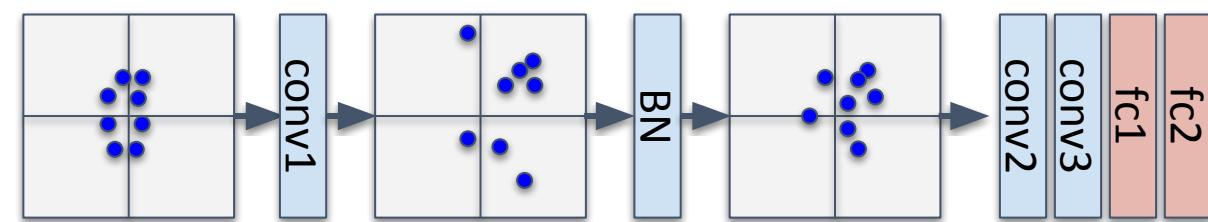
- Batch 2:

- BN calcula μ y σ
- Normaliza



Capa Batch Normalization (BN)

- Capa Batch Normalization (BN o batchnorm)
 - Tiene parámetros μ y σ para normalizar entre capas
 - Acelera el entrenamiento
- Durante el entrenamiento μ y σ se actualizan
 - En ejecución μ y σ quedan fijos



Capa Batch Normalization (BN) en Keras

- Muy fácil de usar ([notebook](#))

```
model = Sequential()
model.add(Conv2D(conv_filters, activation="relu", ...))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(dense_filters, activation="relu"))
model.add(BatchNormalization())
model.add(Dense(classes, activation="softmax"))
```

- Alternativa: BatchNormalization ANTES de ReLU (puede funcionar mejor)

```
model.add(Dense(dense_filters, activation=None))
model.add(BatchNormalization())
model.add(Activation("relu")) #ReLU “desactiva”. BN no la deja
```

Capa Batch Normalization (BN) ([paper](#))

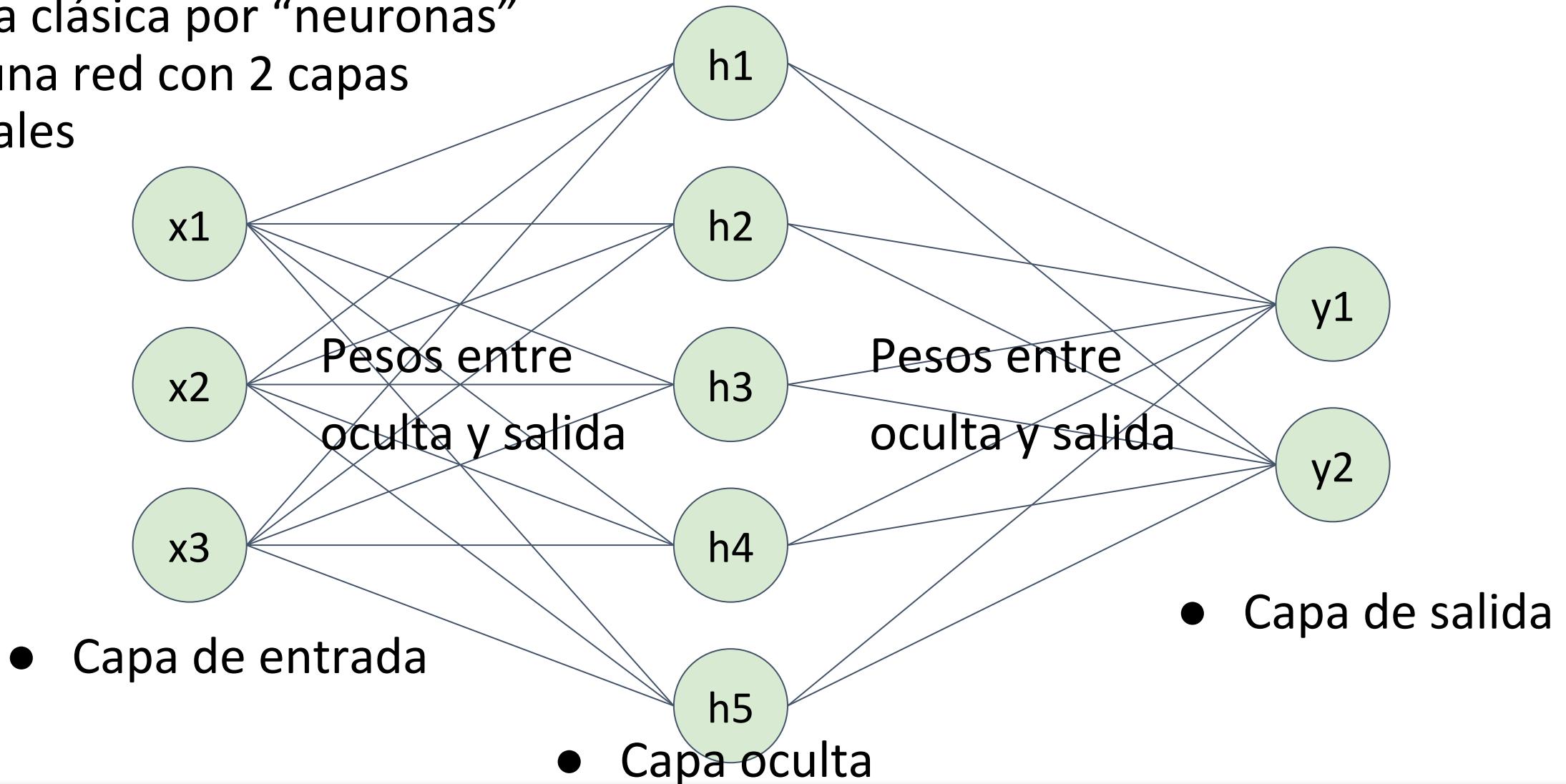
- Detalles avanzados (no es nuestro foco)
 - El cálculo de μ y σ no es parte de la optimización (descenso de gradiente)
 - BN agrega dos parámetros γ y β que sí se optimizan
 - Restauran poder de expresividad perdido por la normalización
 - Posiblemente, mantiene los **autovalores** de los tensores intermedios en un [régimen estable](#)
 - Los valores intermedios no son todos 0, ni son muy grandes
 - Tiene muy poco coste computacional
 - Como el cálculo de μ y σ es por batch, agregan un poco de ruido
 - Sirve como leve regularización del modelo
 - No se entiende completamente todavía
 - Pero se usa porque entrena más rápido

Dropout

(suspender neuronas)

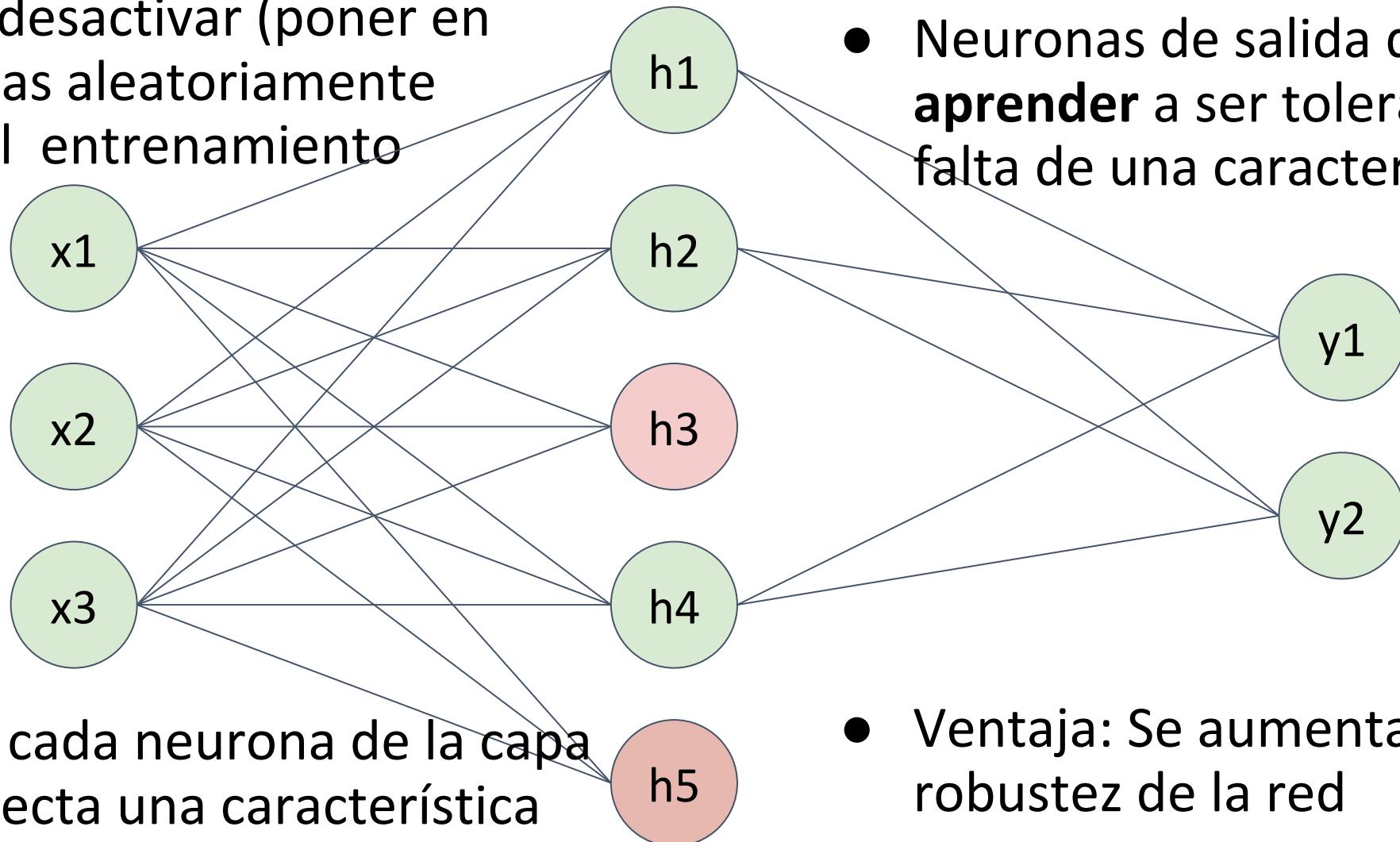
Dropout

- Vista clásica por “neuronas” de una red con 2 capas lineales



Dropout

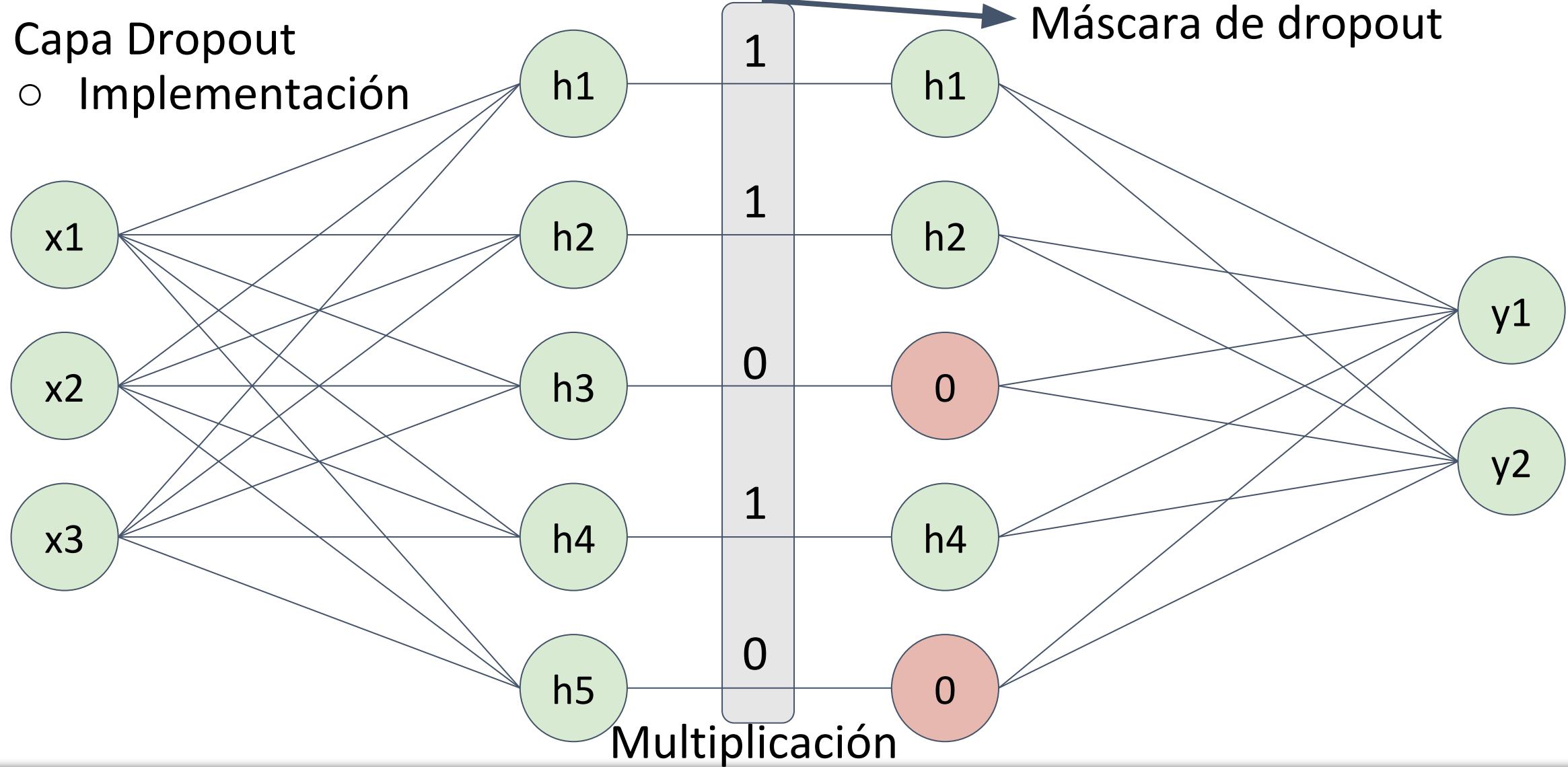
- Dropout: desactivar (poner en 0) neuronas aleatoriamente durante el entrenamiento



- Neuronas de salida deben **aprender** a ser tolerantes a la falta de una característica
- Asunción: cada neurona de la capa oculta detecta una característica
- Ventaja: Se aumenta la robustez de la red

Dropout

- Capa Dropout
 - Implementación

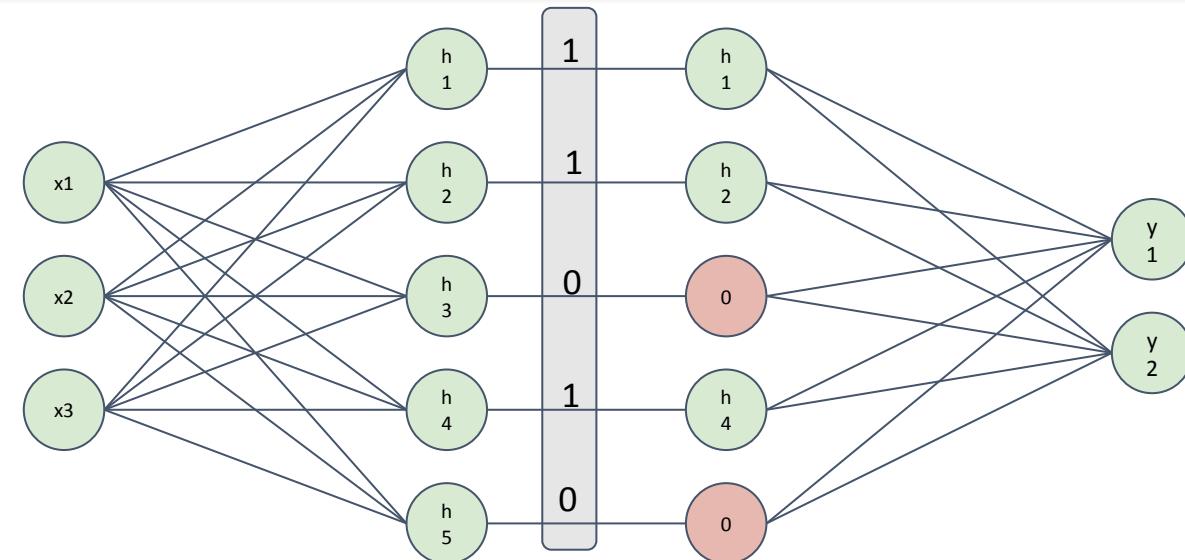


Dropout

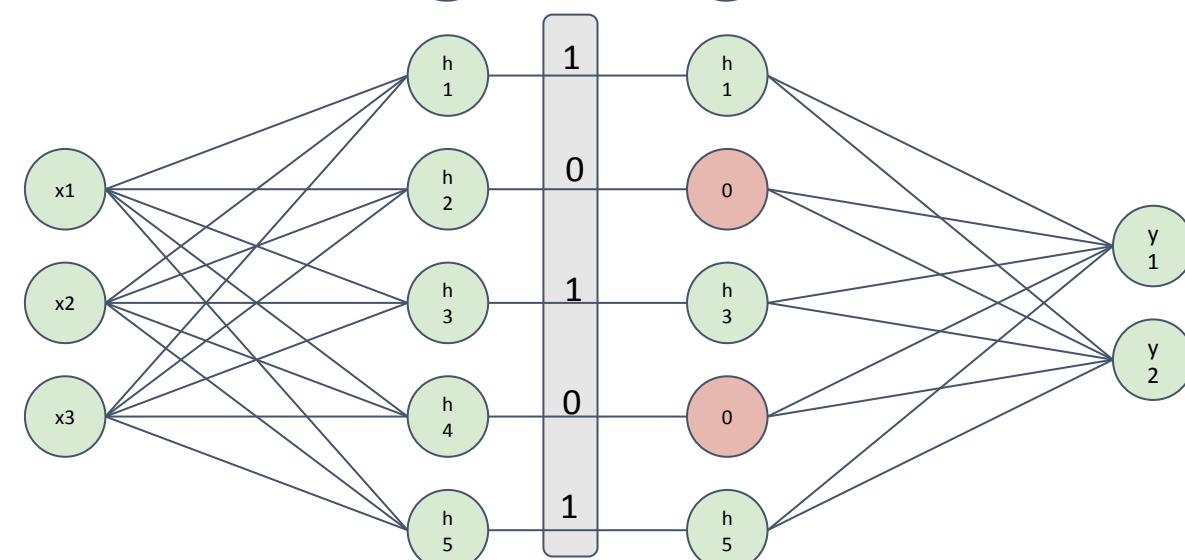
- Capa Dropout
 - Las neuronas a activar **cambian en cada batch**
 - La máscara de Dropout se recalcula
 - Cada neurona se activa con una probabilidad p

```
def dropout_mask(n, p):  
    m = np.zeros(n)  
    for i in range(n):  
        if random.rand()<p:  
            m[i]=1  
    return m
```

Batch 1



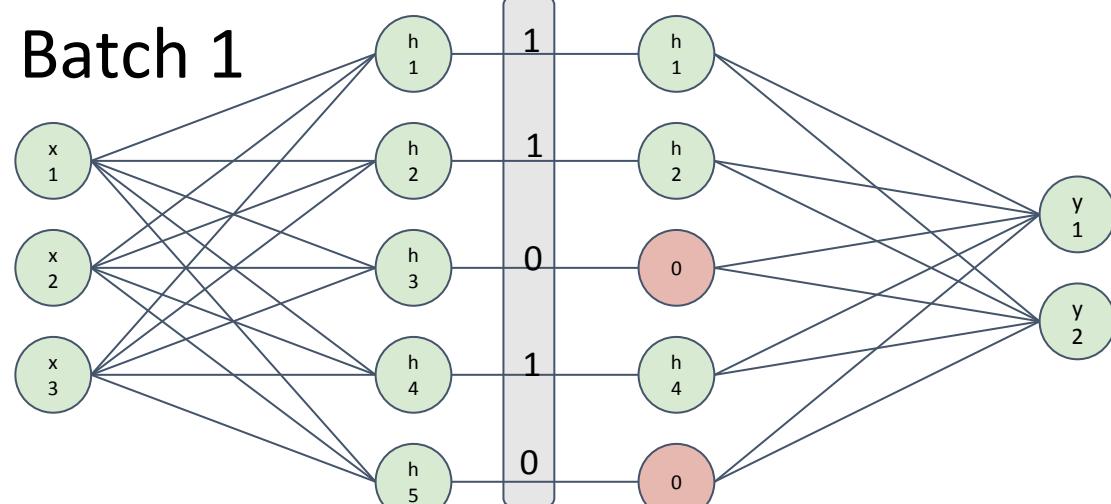
Batch 2



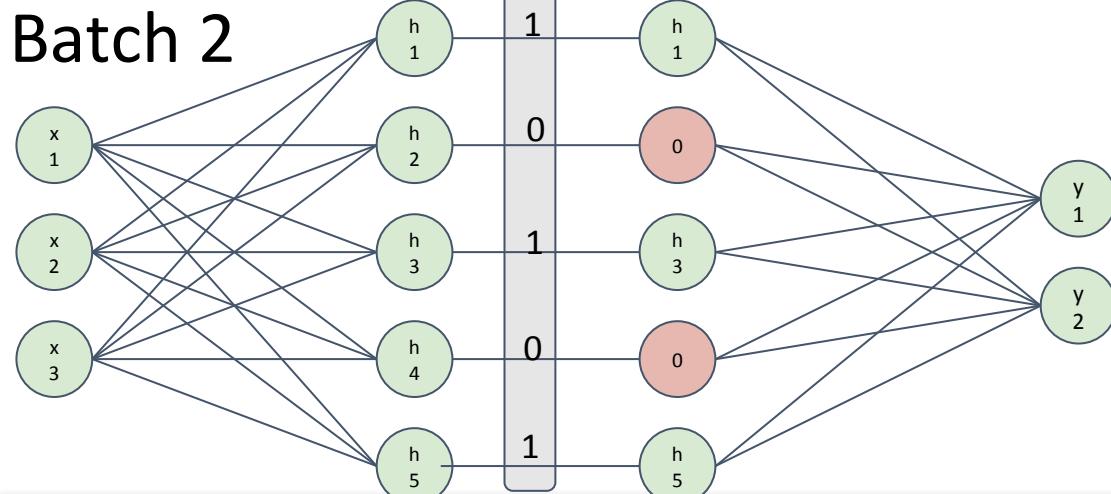
Dropout: Problema de normalización

- Durante el entrenamiento:

Batch 1



Batch 2

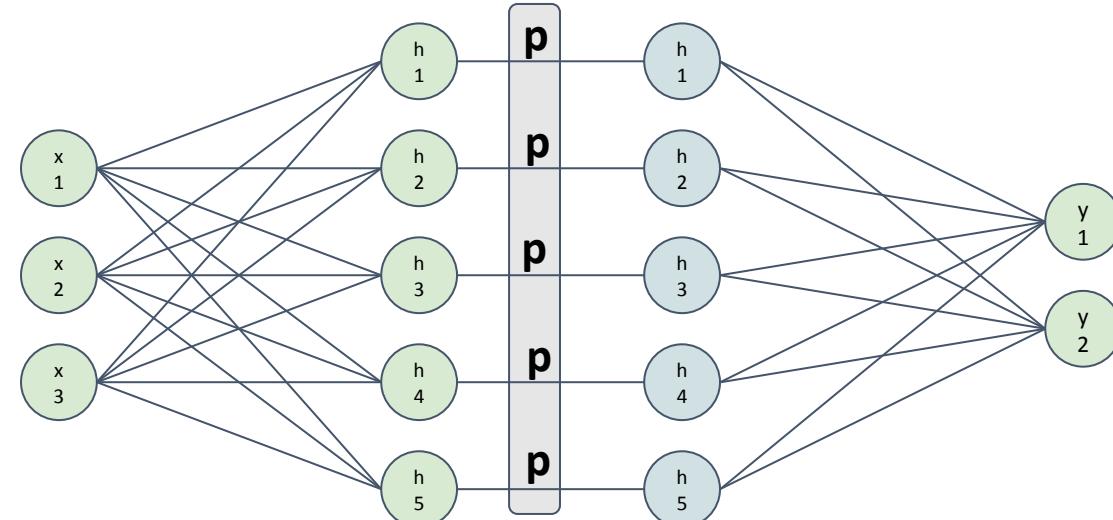


- Al entrenar

- y_1 e y_2 están *acostumbradas* a tener $5 \cdot p \leq 5$ neuronas *activadas*

- Al predecir:

- Las activaciones se multiplican por p
- Mantiene **constante la magnitud** promedio de h_1, h_2, h_3, h_4, h_5



Dropout

- Capa Dropout
 - Comportamiento distinto en train /test
 - Poco coste computacional
 - Regulariza el modelo:
 - Activaciones aleatorias
 - **Modelo distinto en cada batch!**
 - Anular ciertas salidas efectivamente cambia la arquitectura.
 - Valores de p : más bajo en capas más profundas
 - Típicamente: 0.9, 0.7, 0.5

```
def dropout(act,p,phase):  
    # act: activation vector  
    # p: probability to keep act  
    if phase == "test":  
        return act*p  
    elif phase == "train":  
        n = act.shape[0]  
        mask = dropout_mask(n,p)  
        return act*mask
```

Dropout en Keras ([notebook](#))

```
input_shape=(32,32,3)
classes=10

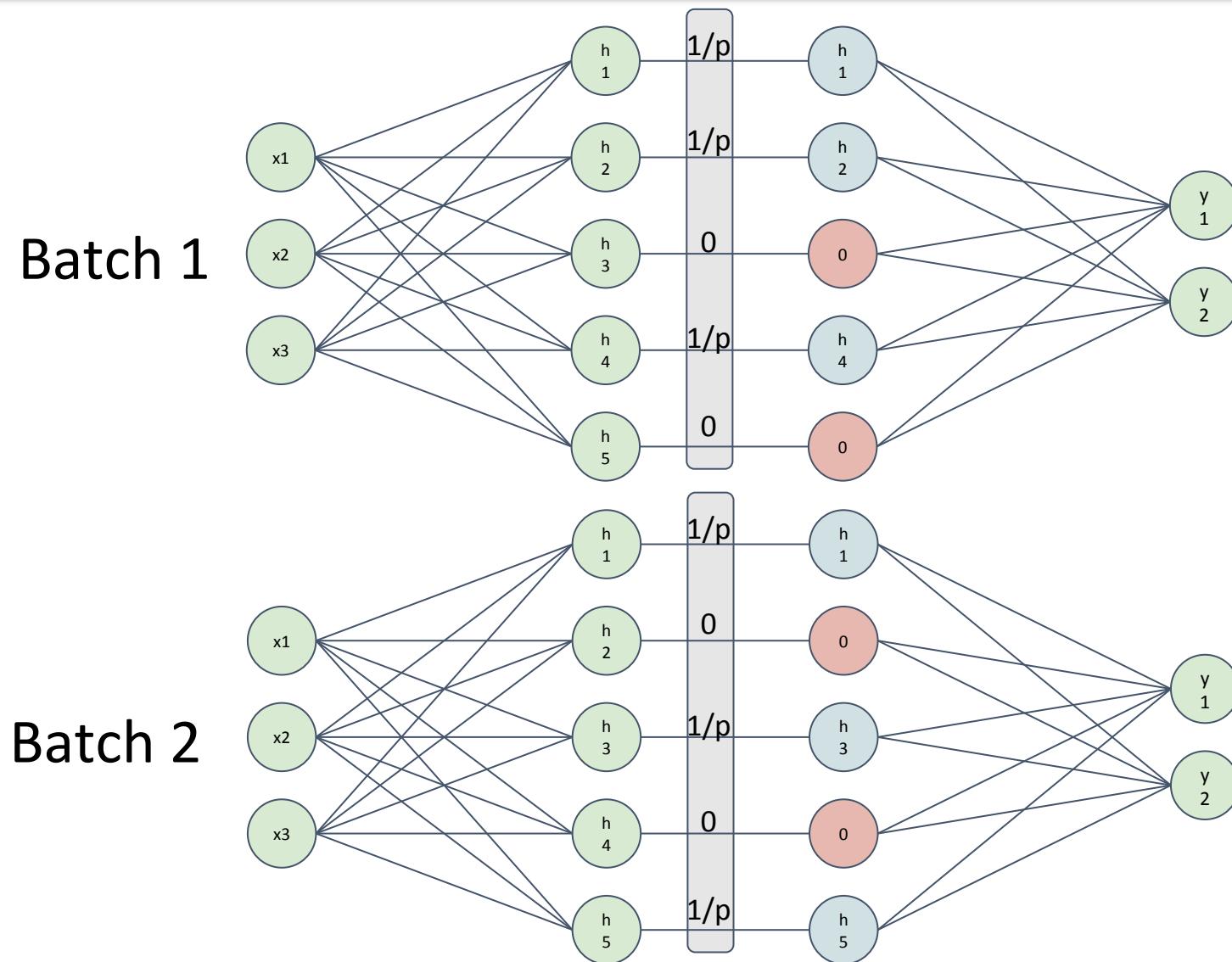
model = Sequential()
model.add(Conv2D(32,(3,3),input_shape=input_shape,...))
model.add(Flatten())
model.add(Dense(64,activation="relu"))
# Agrego Dropout con probabilidad de 0.7
model.add(Dropout(1-0.7)) #recibe 1-p
model.add(Dense(128,activation="relu"))
# Agrego Dropout con probabilidad de 0.5
model.add(Dropout(1-0.5)) #recibe 1-p
model.add(Dense(classes,activation="softmax"))
print(model.summary())
```

Layer (type)	Output Shape	Param #
=====		
conv2d_4 (Conv2D)	(None, 30, 30, 32)	896
flatten_3 (Flatten)	(None, 28800)	0
dense_5 (Dense)	(None, 64)	1843264
dropout_3 (Dropout)	(None, 64)	0
dense_6 (Dense)	(None, 128)	8320
dropout_4 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 10)	1290
=====		
Total params: 1,853,770		

Dropout - Alternativa

- Implementacion alternativa:
 - En la predicción, **NO** multiplicar activaciones por p
 - En entrenamiento, multiplicar activaciones activadas por $1/p$
 - Pred. más eficiente

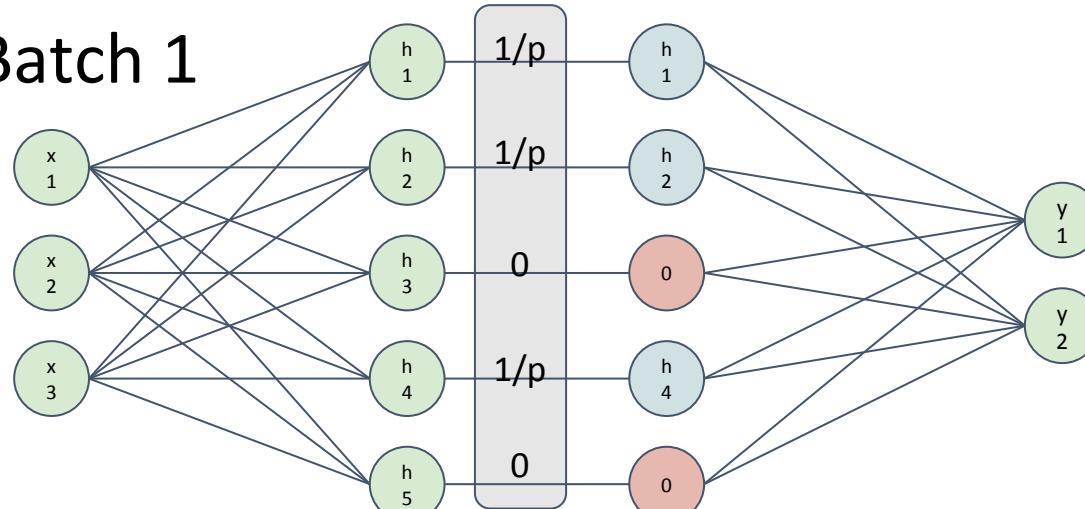
```
def dropout_mask2(n, p):  
    m = np.zeros(n)  
    for i in range(n):  
        if random.rand()<p:  
            m[i]=1/p  
    return m
```



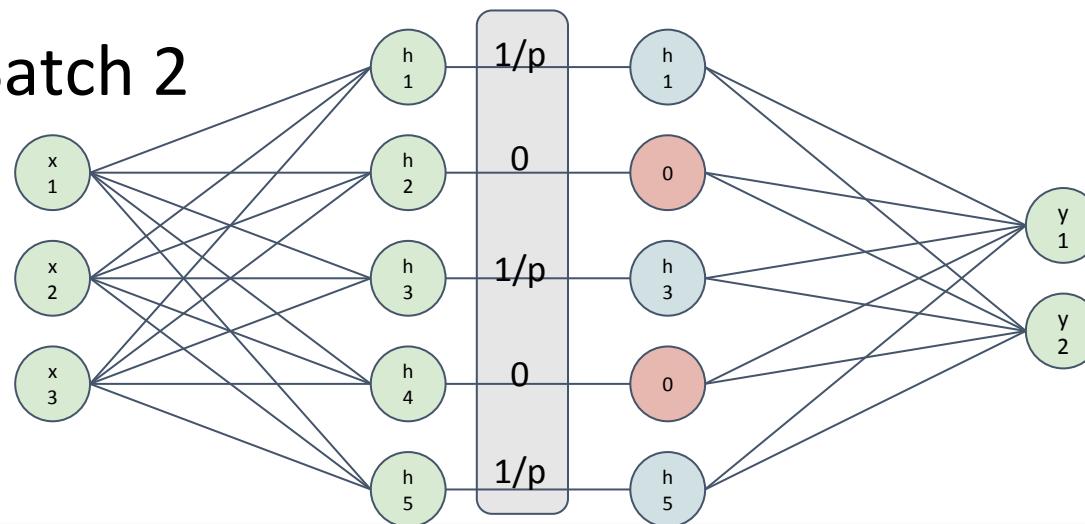
Dropout - Alternativa

- Durante el entrenamiento:

Batch 1



Batch 2

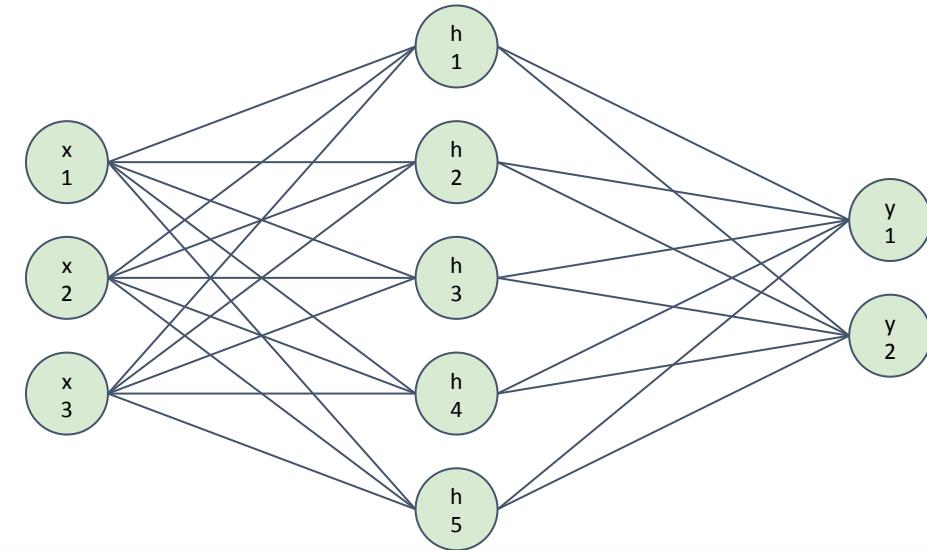


- Al entrenar

- **y_1 e y_2 están acostumbradas a tener la magnitud de $5*p*1/p = 5$ neuronas**

- Al predecir:

- No hacemos nada!
- Podemos “quitar” la capa



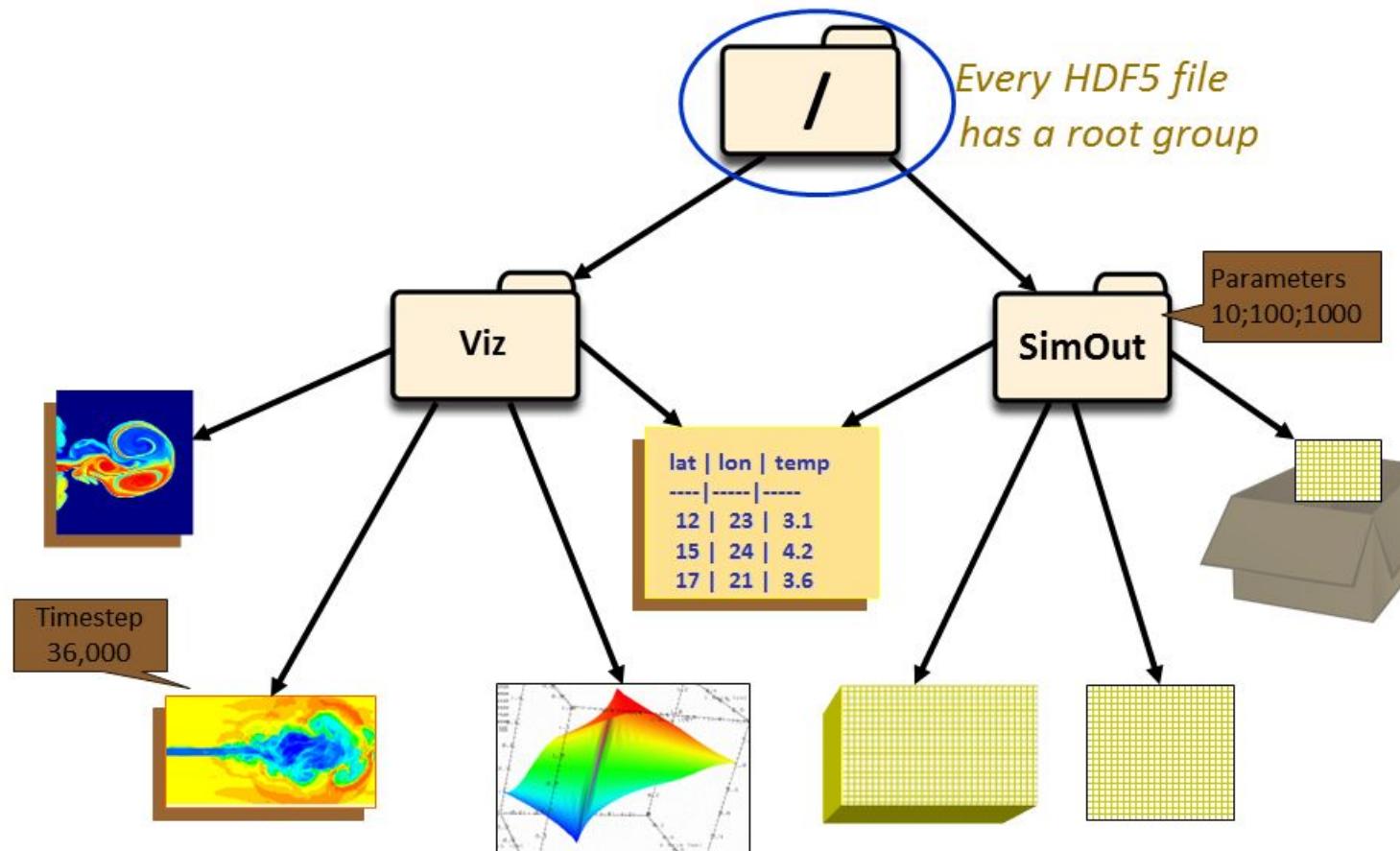
Dropout - Alternativa

```
def dropout2(act,p,phase):  
  
    # act: activation vector  
    # p: probability to keep act  
  
    if phase == "test":  
        # no hacer nada  
        return act  
    elif phase == "train":  
        n = act.shape[0]  
        mask = dropout_mask2(n,p)  
        return act*mask
```

Salvar y cargar modelos

Formato HDF5

- Diseñado y optimizado para guardar y cargar datos numéricos (y de otros tipos)
- Extensión
.h5



Salvar modelos

- Entrenás un modelo, lo probás y es bueno ¿y ahora?
- Guardar el modelo para usar después
- Opción 1: Guardar todo el modelo

```
model = ...
model.fit(..)
...
model.save("ungranmodelo.h5")
```

- Opción 2: Guardar solo los pesos (requiere definición del modelo para cargar)

```
model = ...
model.fit(..)
...
model.save_weights("pesos_de_ungranmodelo.h5")
```

Cargar modelos

- Opción 1: Cargar todo el modelo

```
from keras.models import load_model  
model = load_model('ungranmodelo.h5')
```

- Opción 2: Cargar sólo los pesos (requiere definición del modelo para cargar)

```
model = ... (definición del modelo)  
...  
model.load_weights("pesos_de_ungranmodelo.h5")
```

Modelos pre-entrenados, Transferencia de Aprendizaje y Finetuning

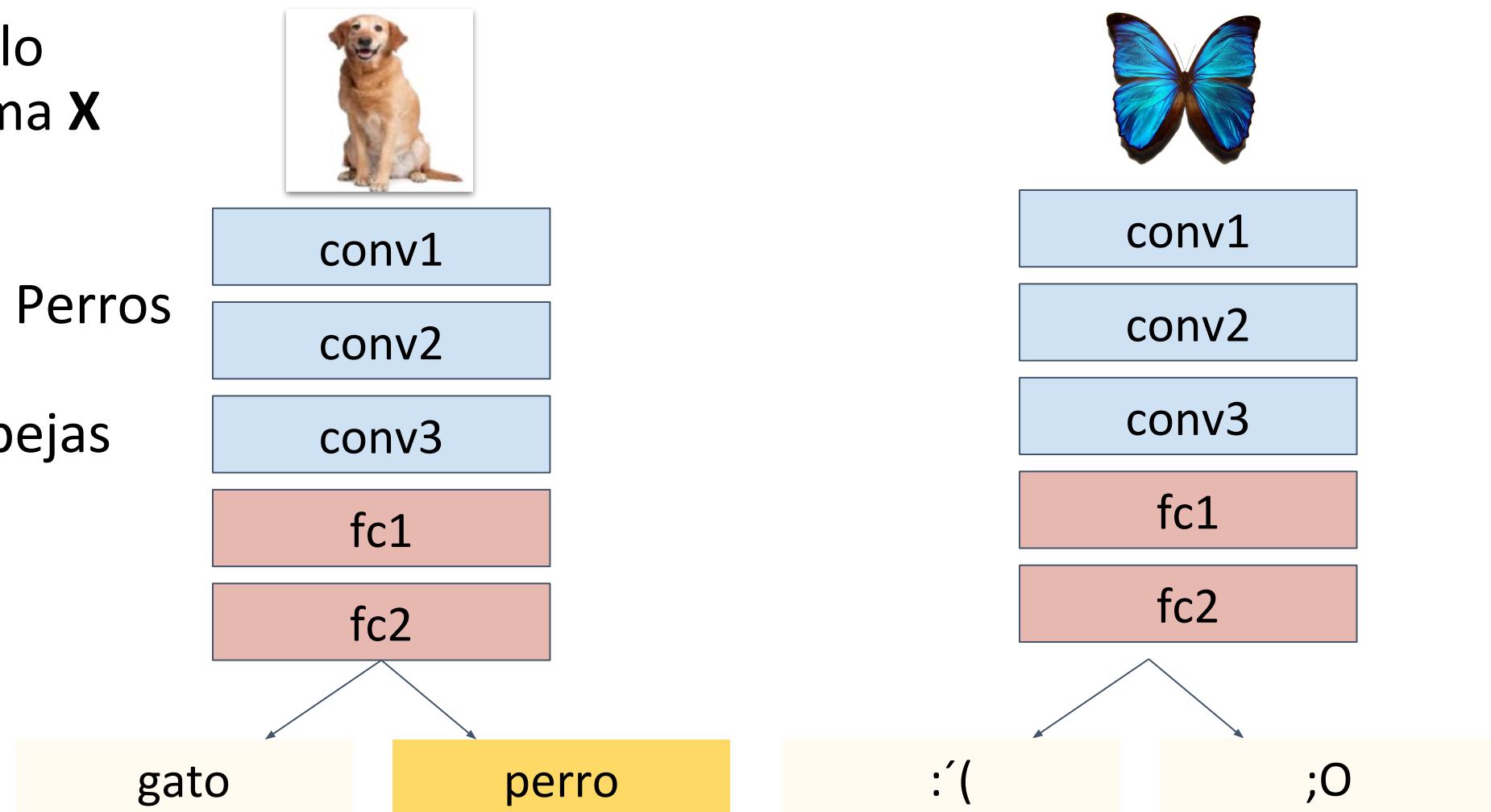
Modelos Preentrenados

- Entrenar un modelo requiere
 - Tiempo/uso de cpu
 - Optimización de hiperparámetros
 - Preparación de datos
- Usar un modelo preentrenado
 - + Fácil y rápido
 - - Arquitectura fija
 - - **Dominio fijo**

```
from keras.applications.mobilenet import  
MobileNet, preprocess_input  
model = MobileNet(weights='imagenet')  
from keras.preprocessing import image  
img = image.load_img(image_path, target_size=(224, 224))  
img = image.img_to_array(img)  
img = np.expand_dims(img, axis=0)  
img = preprocess_input(img)  
class=model.predict_classes(img)
```

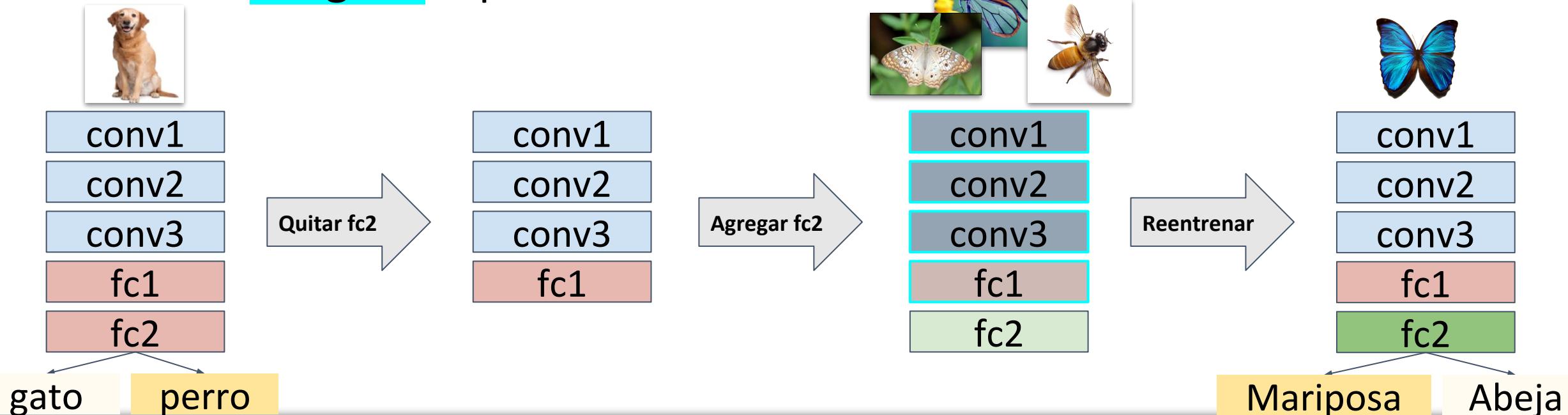
Transferencia de Aprendizaje

- Motivación
 - Tengo un modelo
 - para problema X
 - Quiero resolver
 - problema Y
- Modelo de Gatos y Perros
 - No sirve para
Mariposas vs Abejas



Transferencia de Aprendizaje

- ¿Empiezo desde 0? No
 - **Reusar Filtros Convolucionales**
 - Relativamente independientes del dominio
 - Reemplazar última capa Dense o GlobalAveragePooling
 - Reentrenar red
 - **Congelar** capas anteriores
- **Congelar**
 - No entrenar los pesos



Transferencia de Aprendizaje ([kaggle](#),[notebook](#))

- En Kaggle: Settings => Internet => On para poder bajar el modelo

```
base_model=MobileNet(input_shape=(32,32,3),weights='imagenet',include_top=False)
```

```
for layer in base_model.layers:  
    layer.trainable=False # capas "congeladas" no se entranan
```

Utilizar salida del modelo como entrada a capa GAP y Dense de 128

```
output = GlobalAveragePooling2D()(base_model.output)  
output = Dense(128,activation='relu')(output)
```

Nueva capa de salida

```
output = Dense(classes,activation='softmax')(output)  
model=Model(inputs=base_model.input,outputs=output)
```

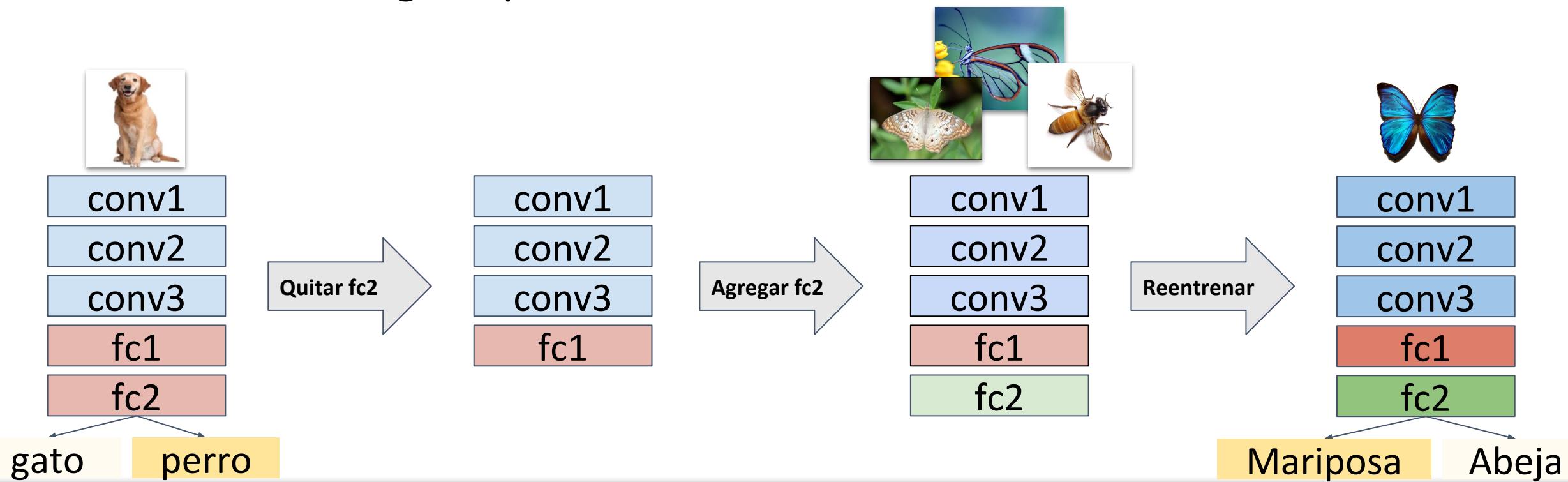
#Entrenar con nuevos datos

```
model.compile(optimizer='Adam',loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

```
model.fit(x,y,epochs=20)
```

Finetuning

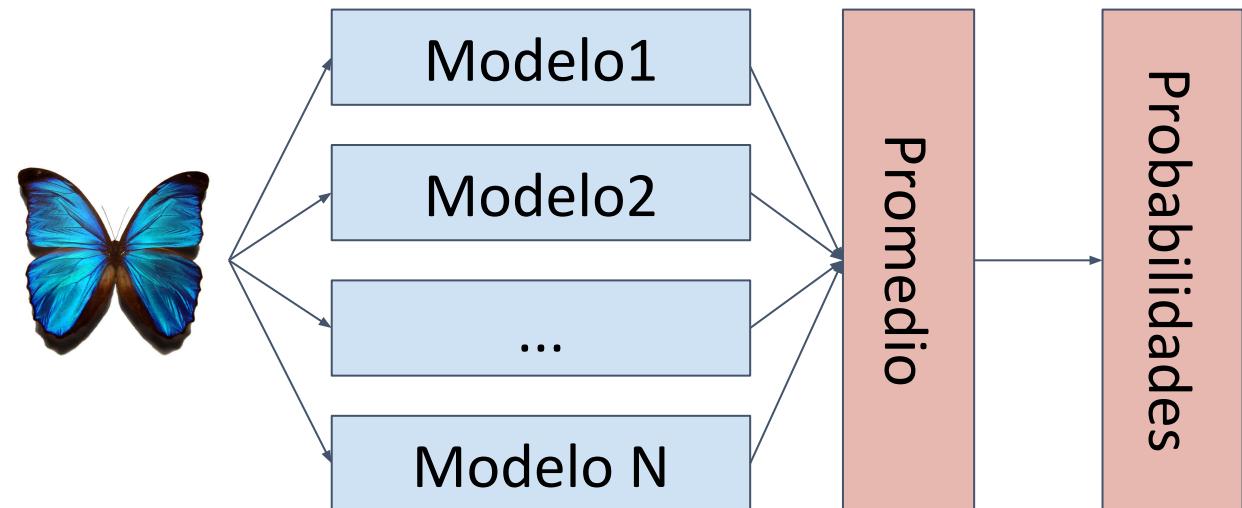
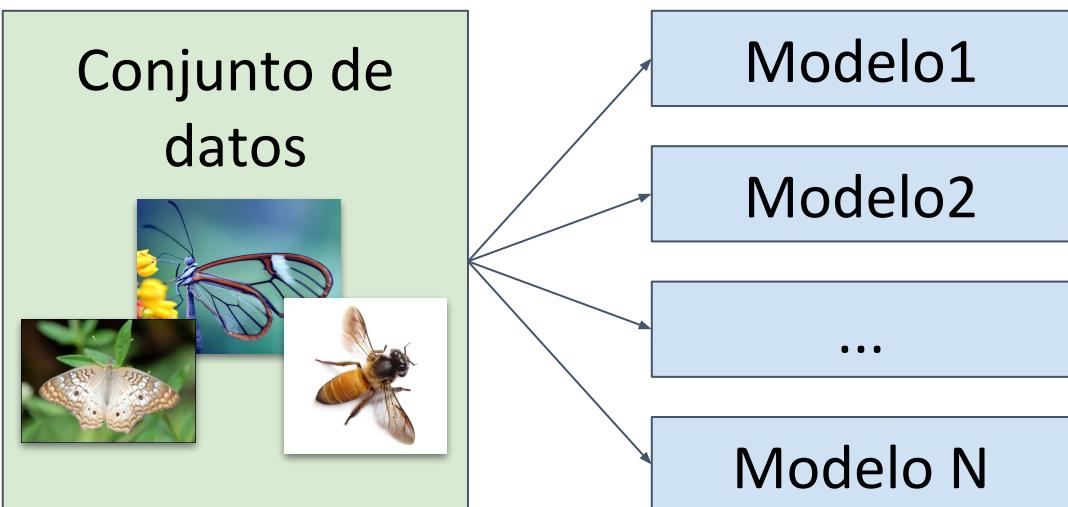
- Reentrenar todo el modelo
 - Más tiempo de entrenamiento
 - Resultados ligeramente mejores (1% a 5%)
- Código: Igual que el anterior
 - Pero sin congelar pesos



Ensemble de modelos

Ensemble de modelos

- El entrenamiento de una red comienza de un estado aleatorio
 - Un modelo sólo puede “salir malo”
 - Usar varios y que “voten” => mejora 1-3% el accuracy
- Entrenamiento
 - Entrenar varios modelos con el mismo conjunto de datos
- Predicción
 - Promediar la salida de todos los modelos p/ una imagen



Ensemble de modelos

- Entrenar `n_models` modelos

```
for i in range(n_models):
    model = ... # definir modelo
    model.fit(..) # entrenar
    model.save(f“modelo{i}.h5”) # guardar
```

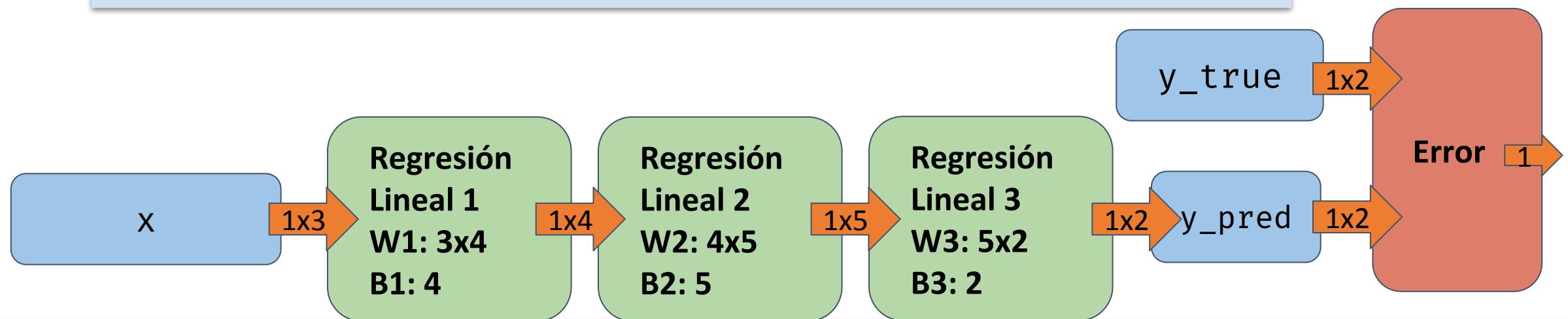
- Cargarlos, predecir con todos y promediar:

```
models = [load_model(f”modelo{i}”) for i in range(n_models)]
probabilities = [model.predict(x) for model in models]
average_probabilities=probabilities[0] #inicializo con el 1ro
for i in range(1,n_models):
    average_probabilities+=probabilities[i]
average_probabilities/=n_models
predictions=probabilities2labels(average_probabilities)
```

Optimización

¿Qué optimizamos?

- Al entrenar
 - **Bajamos el error**
 - **Cambiando los parámetros**
 - Matrices W1, W2, W3, B1, B2, B3
 - Cantidad de parámetros por matriz:
 - W1: $3 \times 4 = 12$, W2: $4 \times 5 = 20$, W3: $5 \times 2 = 10$
 - B1: 4, B2: 5, B3=2
 - $12+20+10+4+5+2 = 53$ parámetros



¿Qué optimizamos?

- # de parámetros en Keras

Layer (type)	Output Shape	Param #
=====		
c1 (Conv2D)	(None, 28, 28, 32)	320
mp1 (MaxPooling2D)	(None, 14, 14, 32)	0
c2 (Conv2D)	(None, 7, 7, 64)	18496
mp2 (MaxPooling2D)	(None, 3, 3, 64)	0
flatten_7 (Flatten)	(None, 576)	0
fc1 (Dense)	(None, 512)	295424
fc2 (Dense)	(None, 10)	5130
=====		

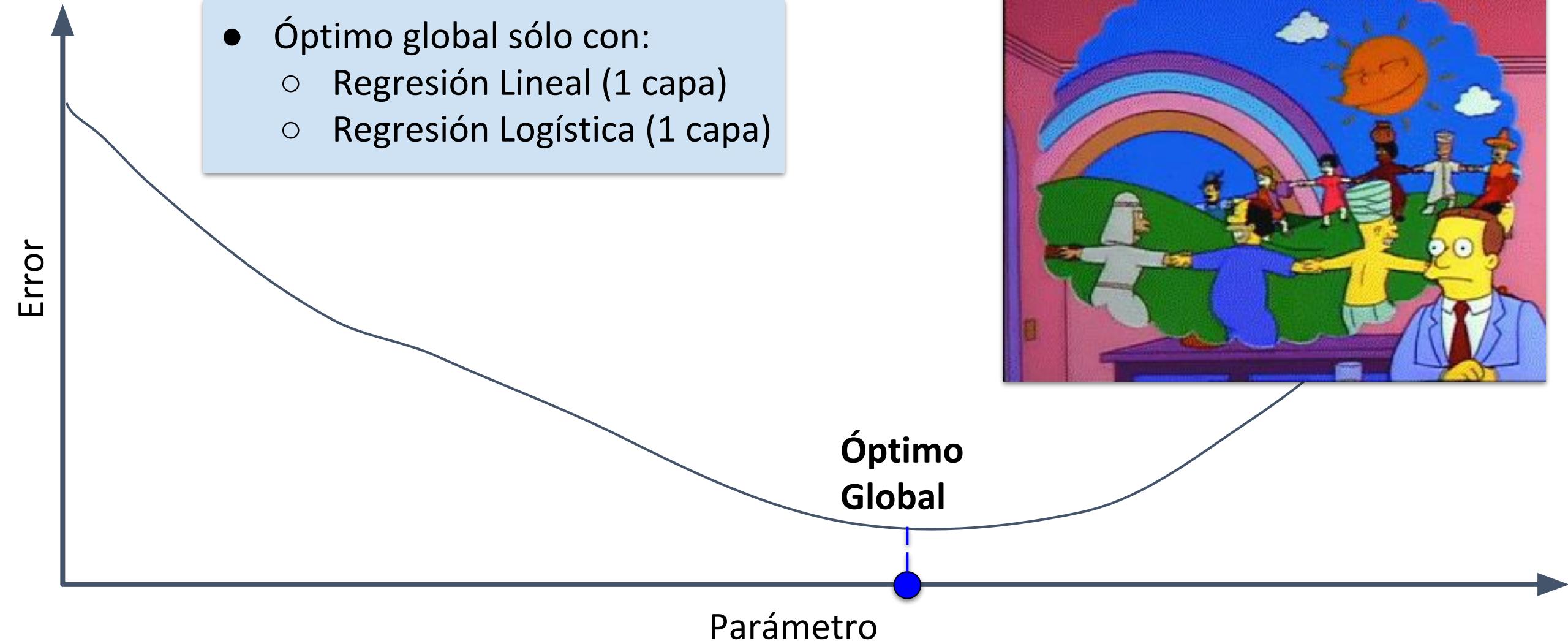
Total params: 319,370

Trainable params: 319,370

Non-trainable params: 0

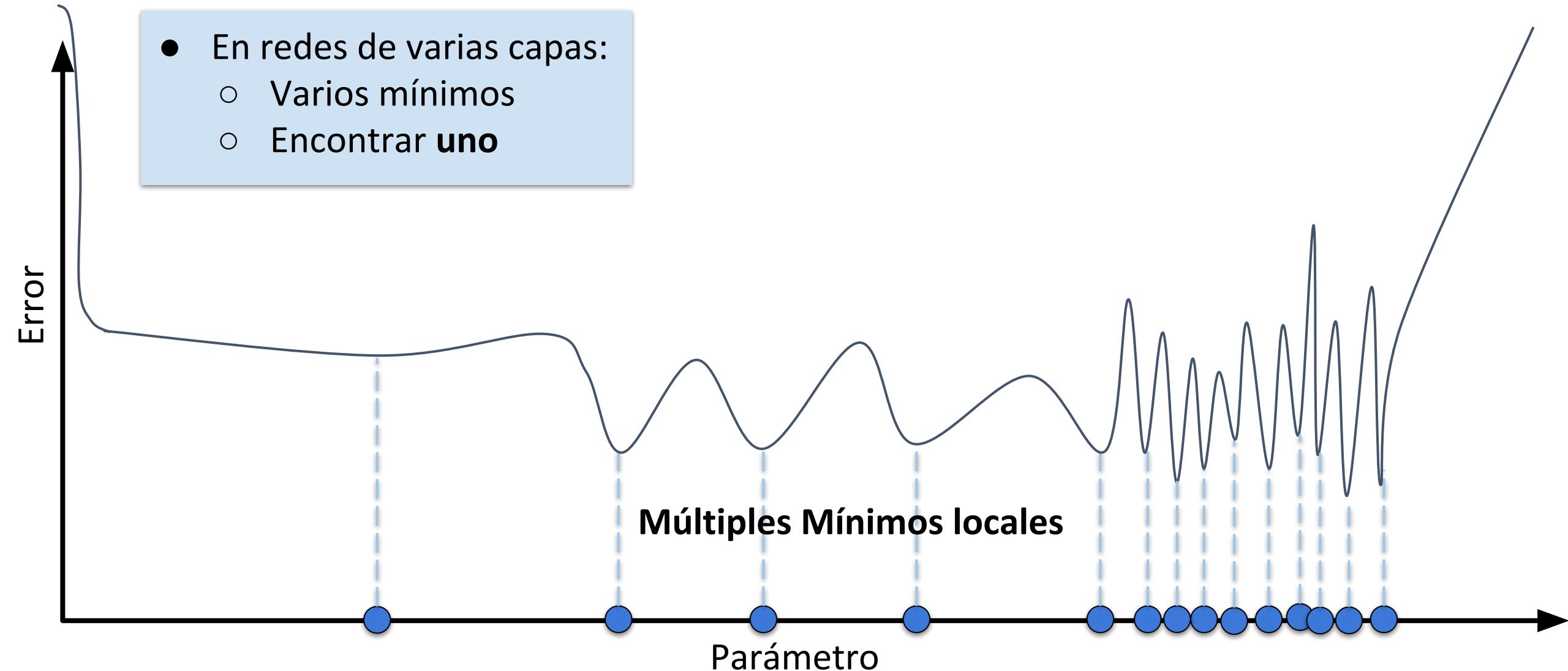
Optimización

- Óptimo global sólo con:
 - Regresión Lineal (1 capa)
 - Regresión Logística (1 capa)



Optimización

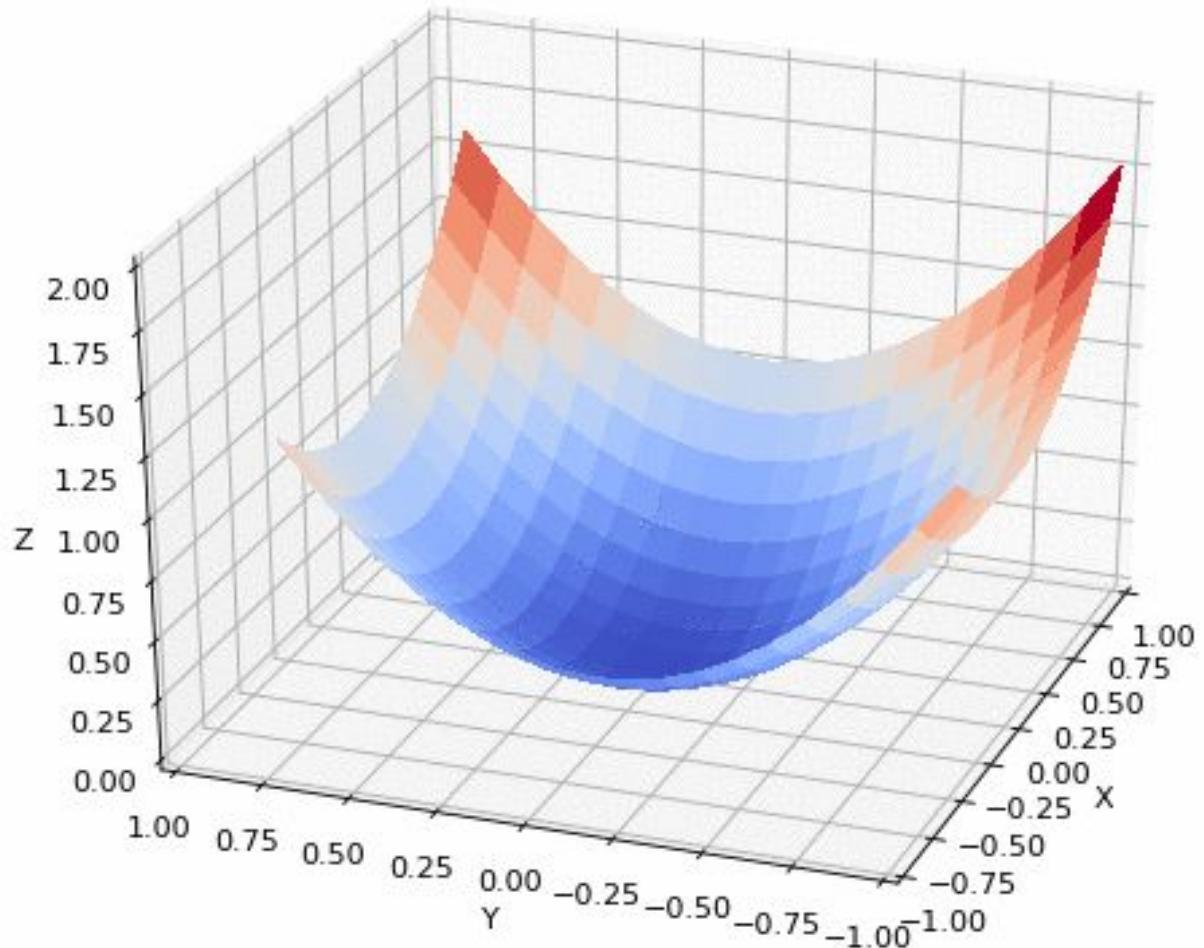
- En redes de varias capas:
 - Varios mínimos
 - Encontrar **uno**



Optimización

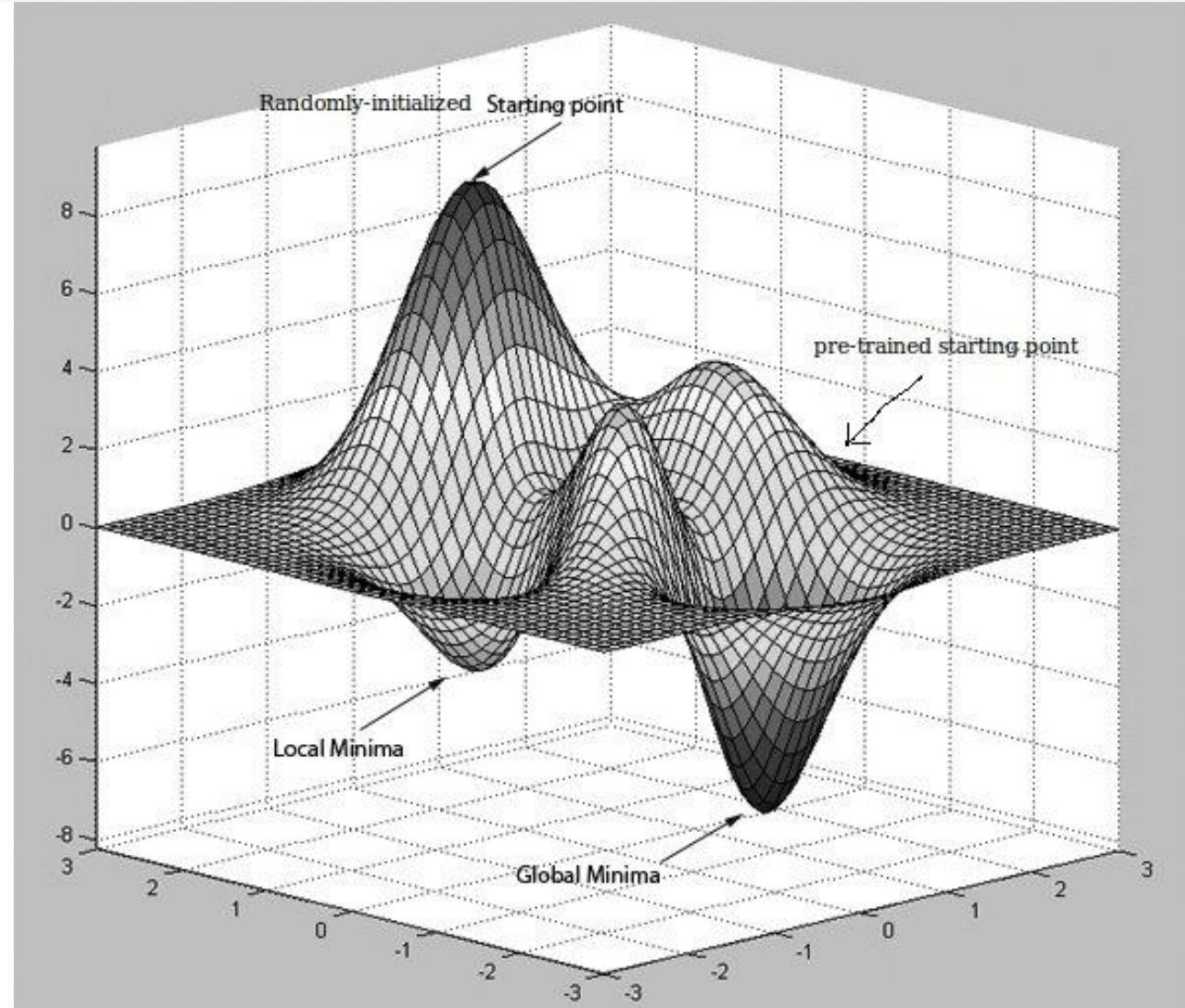
- Con 2 parámetros
 - **Superficie** de error
 - Z: valor del error
 - X e Y: parámetros
 - Caso Convexo
 - 1 solo mínimo (global)

- ¿Con **n** parámetros?
 - No podemos visualizar
 - Herramientas **analíticas**



Optimización

- Con 2 parámetros
 - **Superficie** de error
 - Z: valor del error
 - X e Y: parámetros
 - Caso NO convexo
 - Varios mínimos
 - Puede existir global (o no)
- ¿Con n parámetros?
 - No podemos visualizar
 - Herramientas **analíticas**



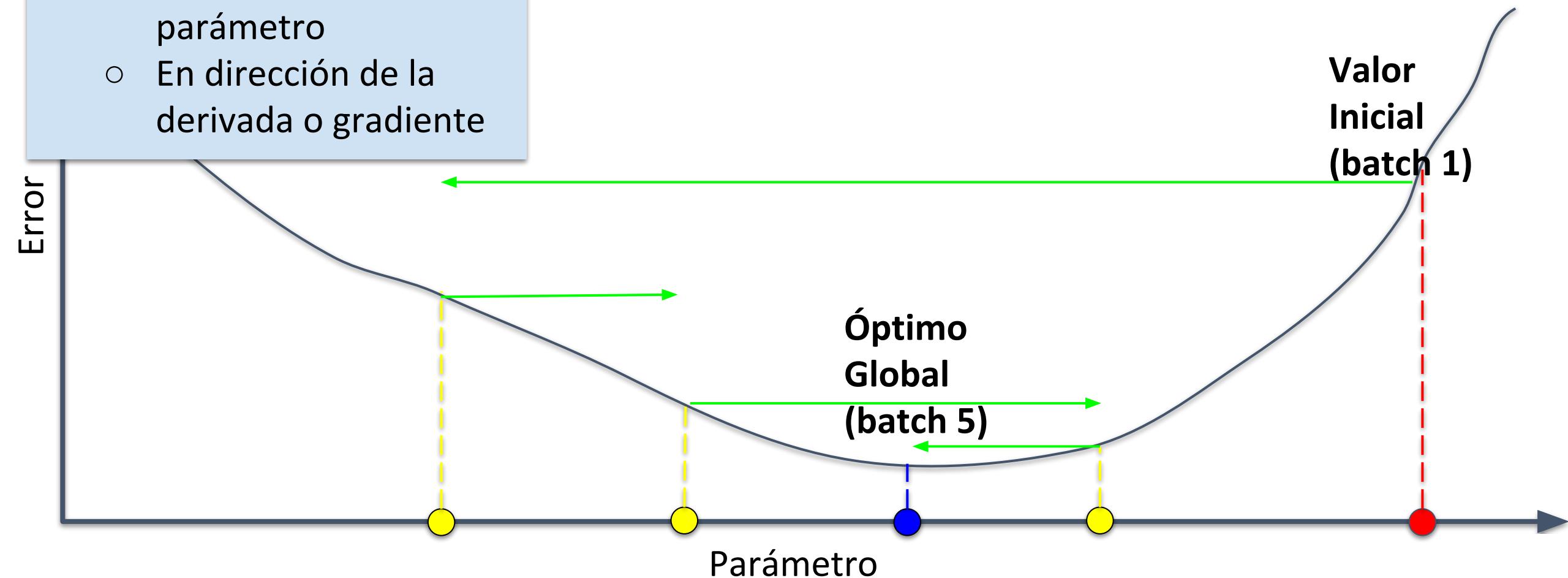
Optimización

- ¿Con n parámetros?
 - No podemos visualizar
 - Intuiciones en 1D, 2D y herramientas matemáticas

Descenso de gradiente

Descenso de gradiente estocástico

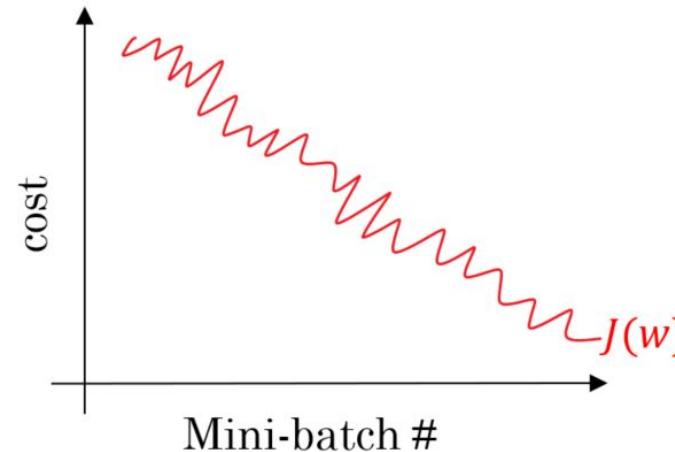
- Ejemplo con 5 batchs
 - 5 modificaciones del parámetro
 - En dirección de la derivada o gradiente



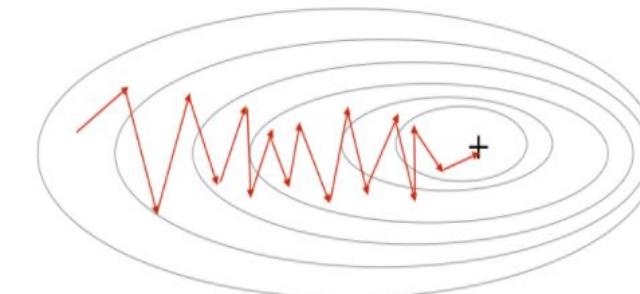
Descenso de gradiente - Versiones

- Descenso de gradiente **estocástico**
 - Por cada batch
 - Calcular derivadas
 - Actualizar parámetros
 - Batch
 - Subconjunto del total de ejemplos de entrenamiento
 - Descenso ruidoso
 - También se lo llama **minibatch gradient descent**
- Descenso de gradiente **tradicional**
 - Calcular derivadas con *todos* los ejemplos de entrenamiento
 - Por motivos históricos, se lo llama **batch gradient descent**

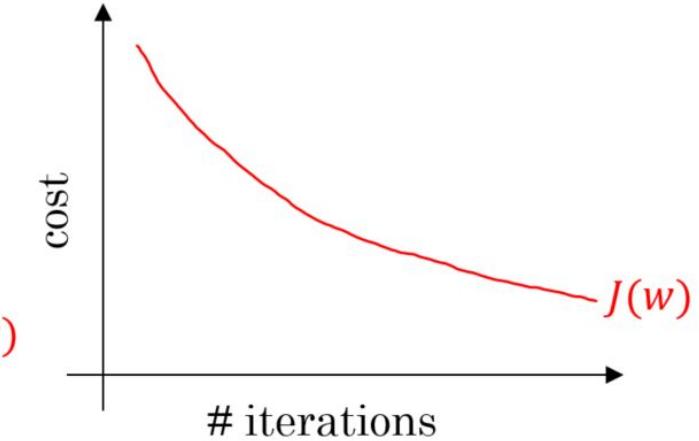
Mini-batch gradient descent



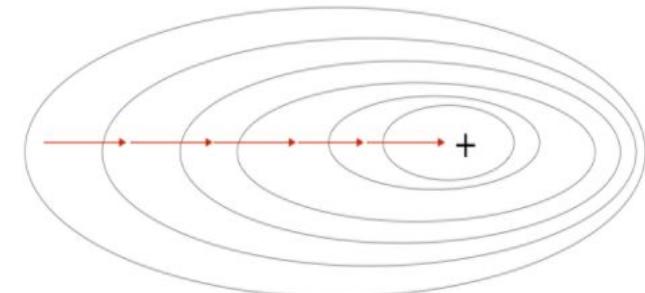
Stochastic Gradient Descent



Batch gradient descent



Gradient Descent



Descenso de gradiente - Versiones

- Descenso de gradiente tradicional
 - Simplificado:
 - `model.w` = matriz con todos los parámetros de la red
 - `dw` = matriz con la derivada de respecto a `model.w`
- Descenso de gradiente estocástico

```
x,y = load_data()

lr = 0.001 # learning rate
model = ....
epochs = 1000

for i in range(epochs):
    dw = model.derivatives(x,y)
    model.w = model.w - lr * dw
```

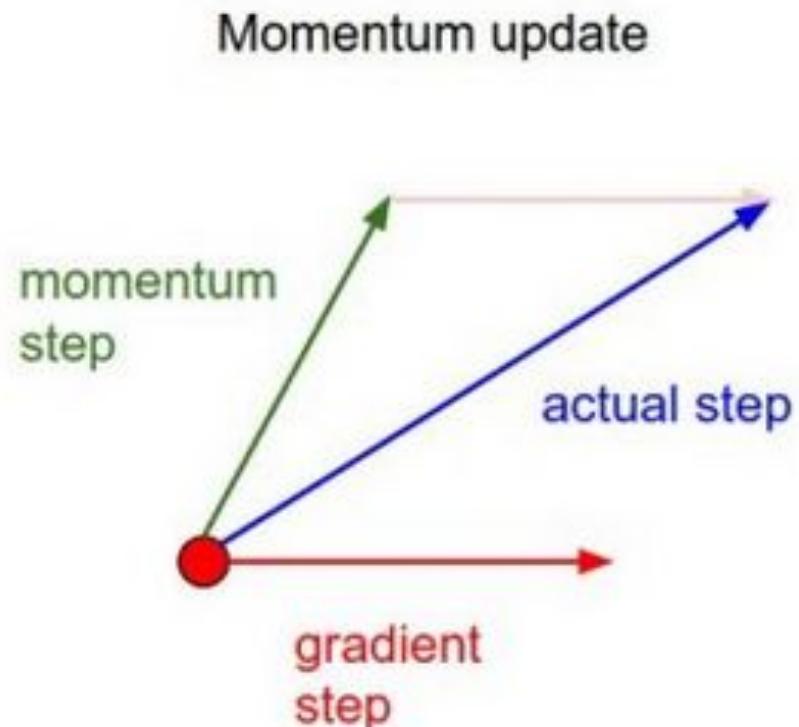
```
x,y = load_data()
lr = 0.001 # learning rate
model = ....
epochs = 1000

n = x.shape[0]
batch_size = 32
batches = n // batch_size

for i in range(epochs):
    for batch in range(batches):
        batch_x,batch_y = get_batch(x,y,batch)
        dw = model.derivatives(batch_x,batch_y)
        model.w = model.w - lr * dw
```

Descenso de gradiente con Momentum

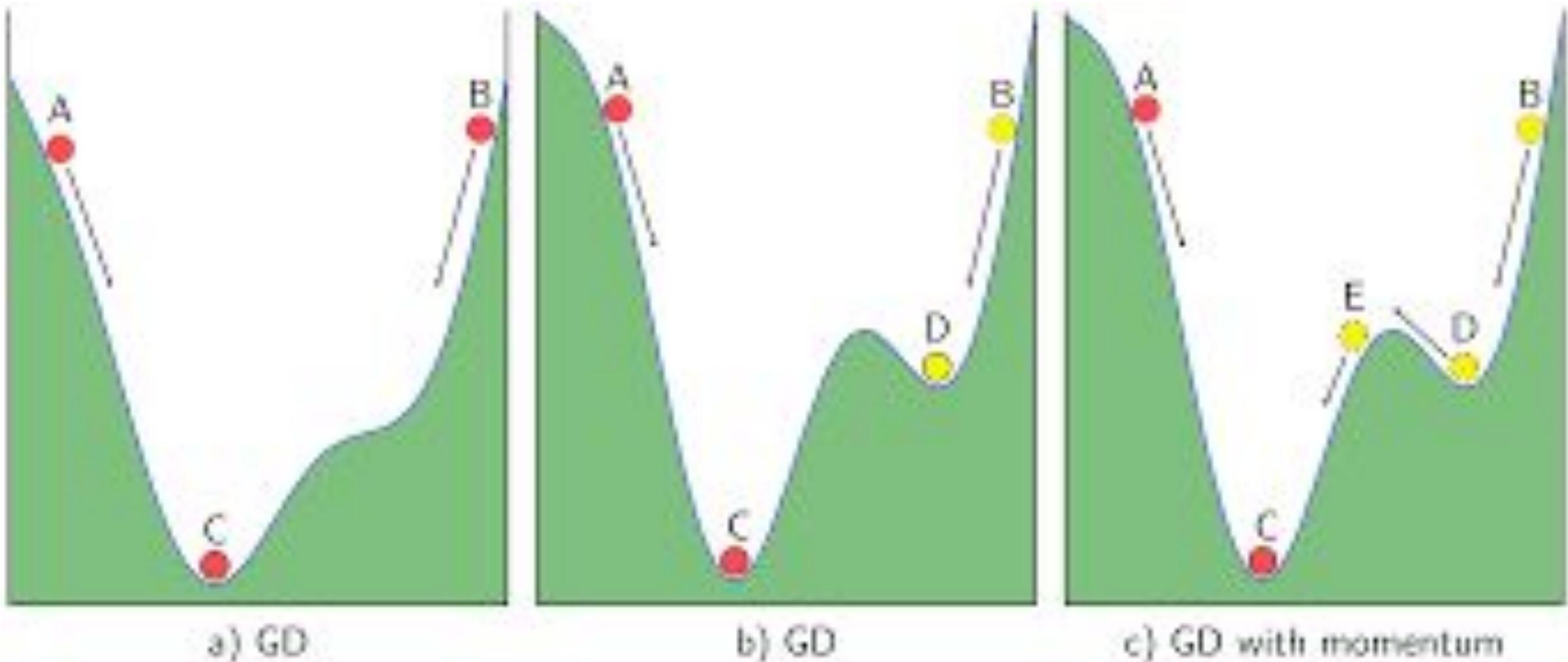
- Descenso de gradiente con velocidad o *momentum*
 - Matriz **v** de *momentum*
 - Dirección y velocidad promedio
 - Permite escapar mínimos locales “malos”



```
v=np.zeros(model.w.shape)
# mu: coef de fricción, entre 0 y 1
mu = 0.9
...
v = mu * v + alpha * dw
model.w = w - v
```

Descenso de gradiente con momentum

- Permite saltar mínimos locales *espurios* o “malos”



Optimizadores Avanzados: AdaGrad, RMSProp,

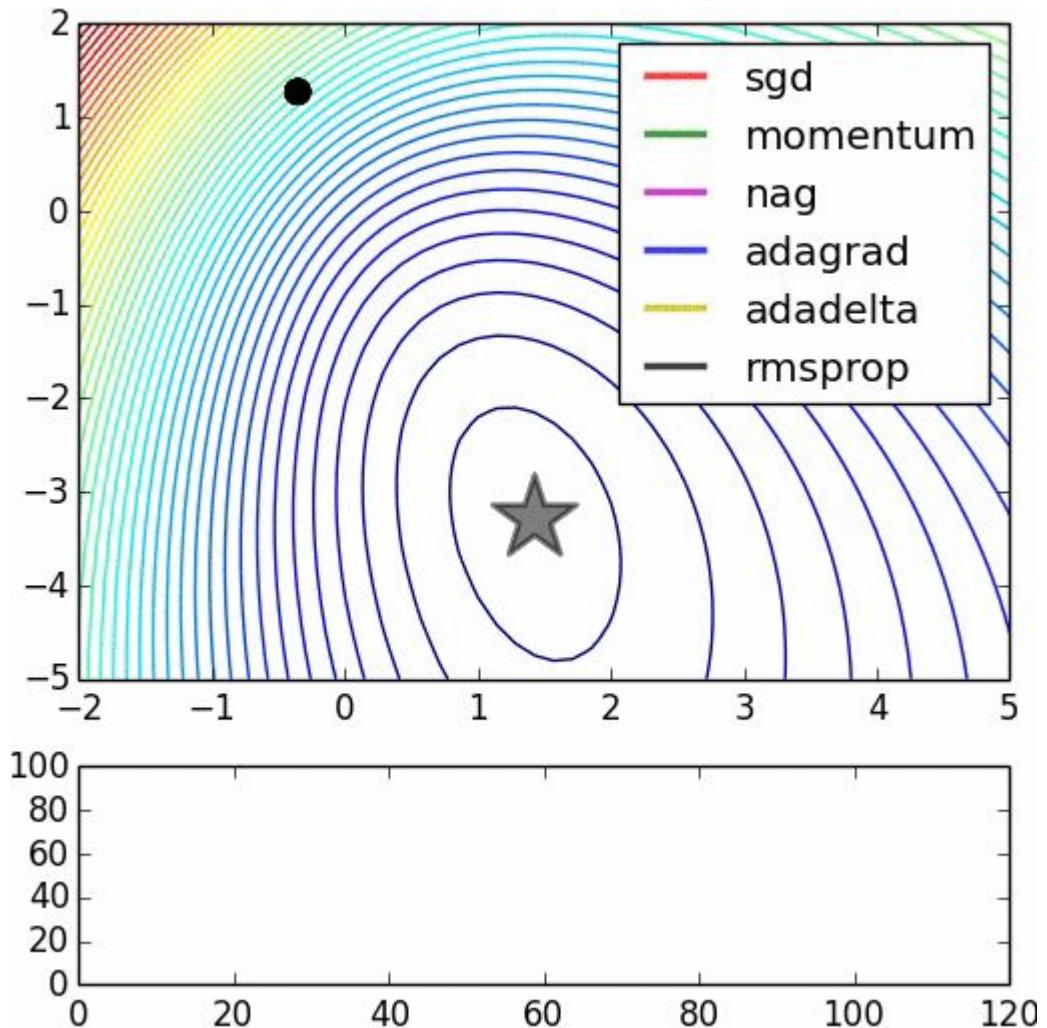
- Descenso de gradiente **AdaGrad**
 - Normaliza la magnitud de los gradientes
 - Importa más la **dirección**
 - Mejora el comportamiento con gradientes muy bajos/altos

```
gradient_history=np.zeros_like(dw)
eps = 0.0001 # para evitar division por 0
...
gradient_history = gradient_history + dw**2
normalizing_factor = (np.sqrt(gradient_history)+eps)
w = w - (dw / normalizing_factor)
```

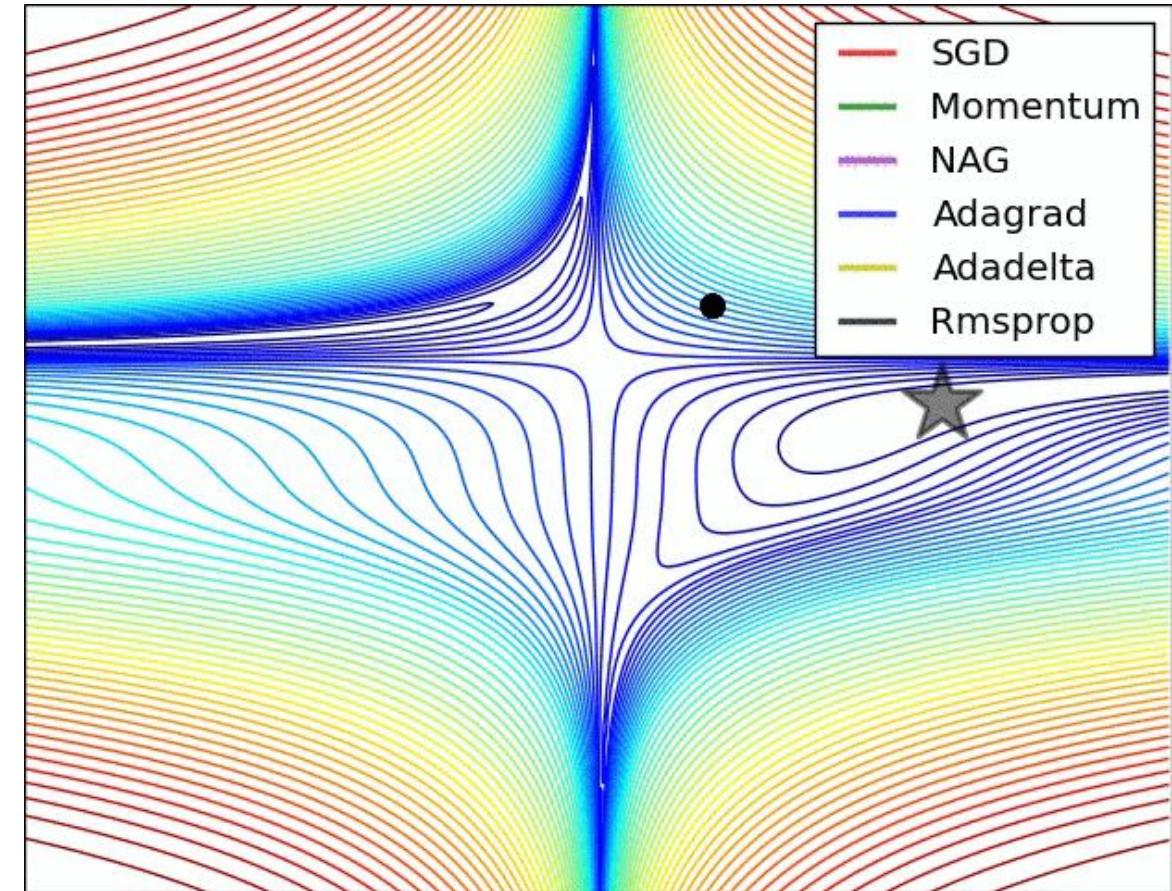
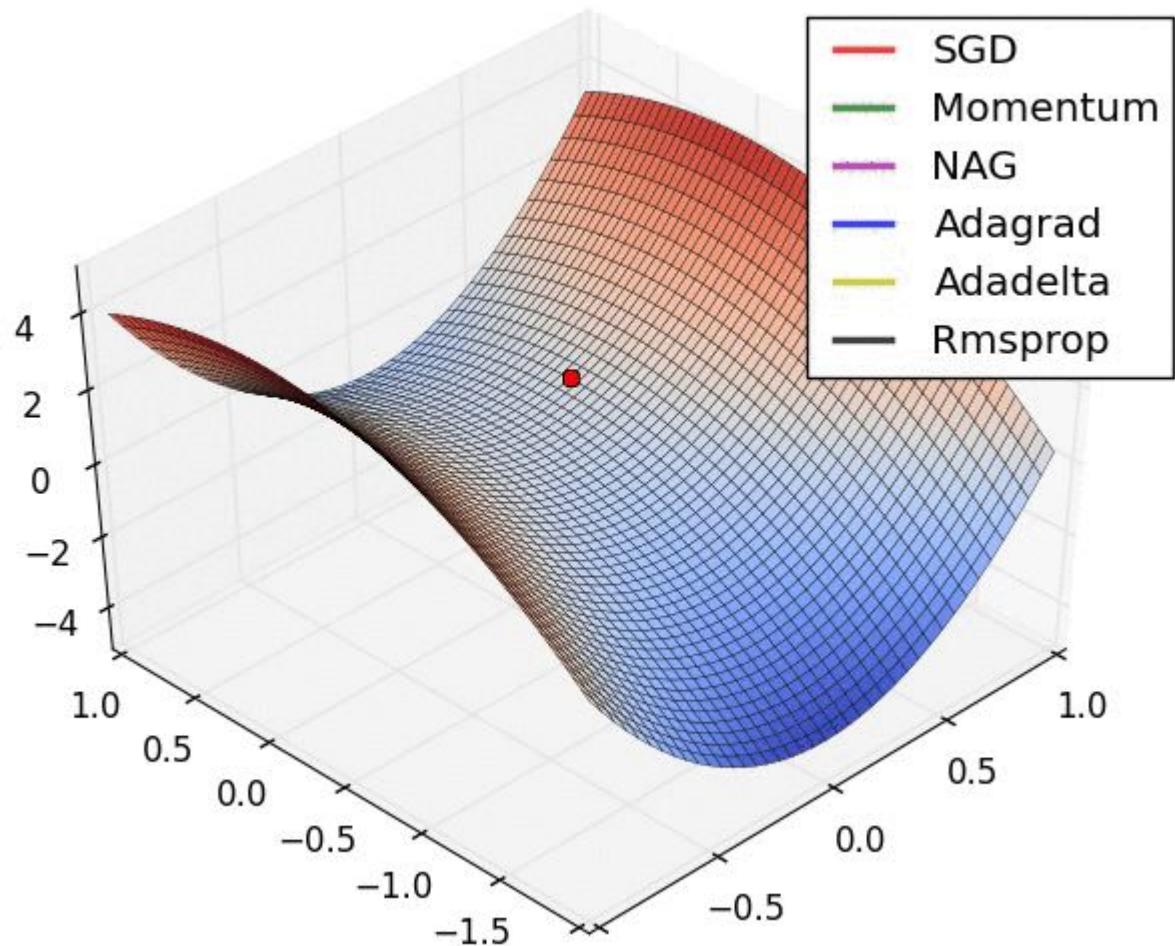
- Descenso de gradiente **Adam**: como **AdaGrad** con **Momentum**
 - No requiere encontrar un
 - Algoritmo por defecto cuando no hay tiempo

Optimizadores Avanzados

- Algoritmos más avanzados
 - Requieren menos pasos
 - Cada paso es más caro
 - más operaciones
 - más memoria
 - Adam es robusto y rápido
 - Utilizar para prototipar



Optimizadores Avanzados



Optimizadores Avanzados

- Descenso de gradiente tradicional
 - Necesita todos los ejemplos en memoria
 - Sobreajusta más (¿por qué?)
- Descenso de gradiente estocástico (**SGD**)
 - Evita los problemas del tradicional
 - Requiere ajustar la tasa de aprendizaje con cuidado (¿por qué?)
 - Obtiene los mejores resultados
- Algoritmos de optimización avanzados
 - Intentan no depender de la tasa de aprendizaje
 - Más usados:
 - **Momentum**
 - **Adagrad**
 - **Adam**
 - Los modelos entrenados son ligeramente peores que con SGD

Optimizadores Avanzados - En Keras

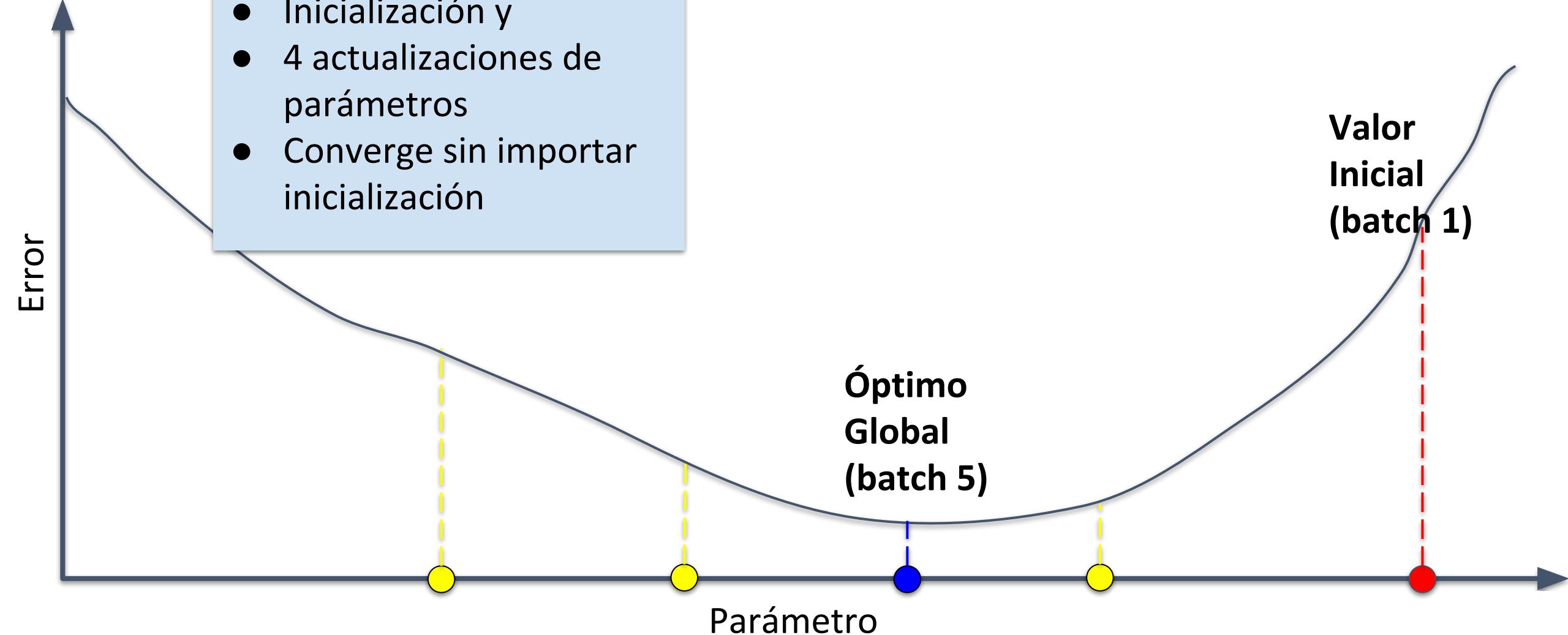
- Optimizadores modulares
 - Intercambiables
 - Desacoplados del cálculo de derivadas
 - Fácil implementar nuevos

```
opt = keras.optimizers.SGD(lr=0.01)
opt = keras.optimizers.SGD(lr=0.01, momentum=0.9)
opt = keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
opt = keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
beta_2=0.999, amsgrad=False)
...
model.compile(...,optimizer=opt)
...
#alternativamente
model.compile(...,optimizer="sgd")
model.compile(...,optimizer="rmsprop")
model.compile(...,optimizer="adam")
```

Inicialización de Parámetros

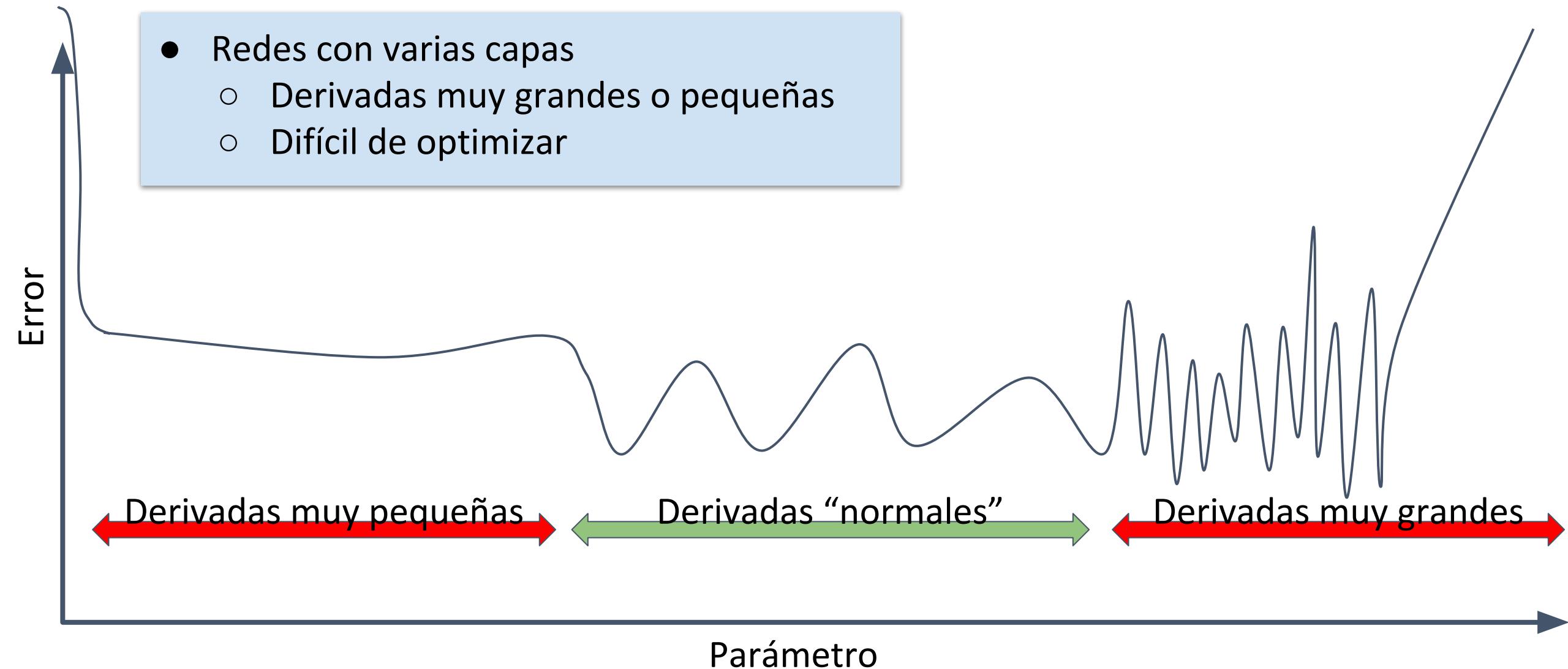
Optimización

- Inicialización y
- 4 actualizaciones de parámetros
- Converge sin importar inicialización



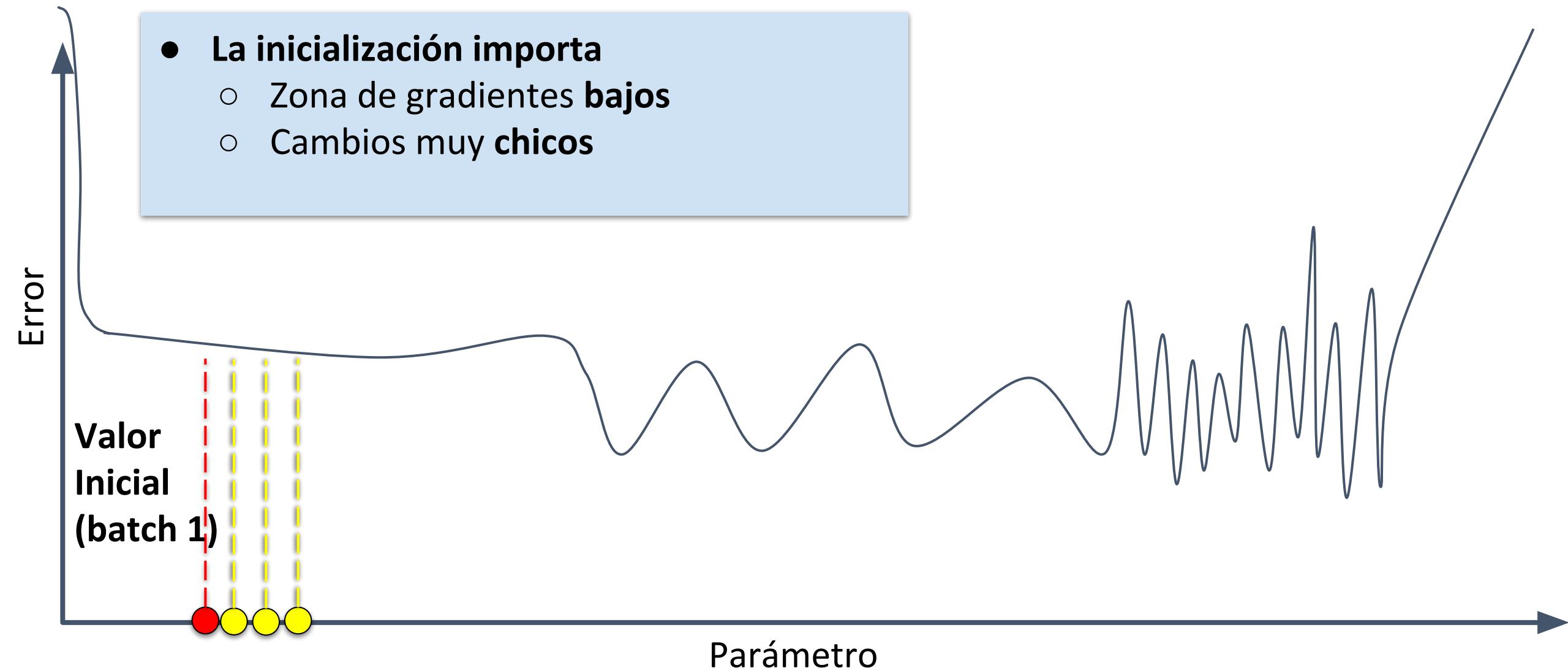
Optimización

- Redes con varias capas
 - Derivadas muy grandes o pequeñas
 - Difícil de optimizar



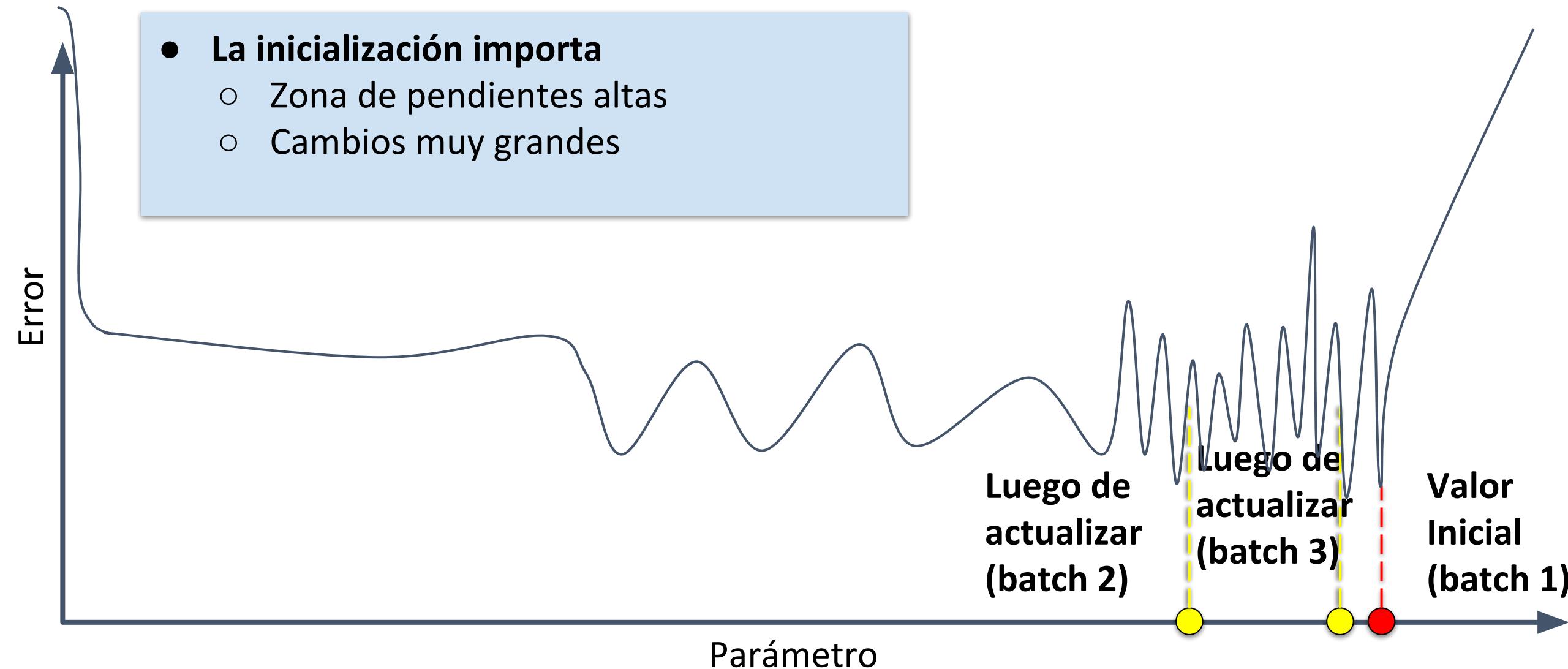
Inicialización de parámetros

- La inicialización importa
 - Zona de gradientes **bajos**
 - Cambios muy **chicos**



Inicialización de parámetros

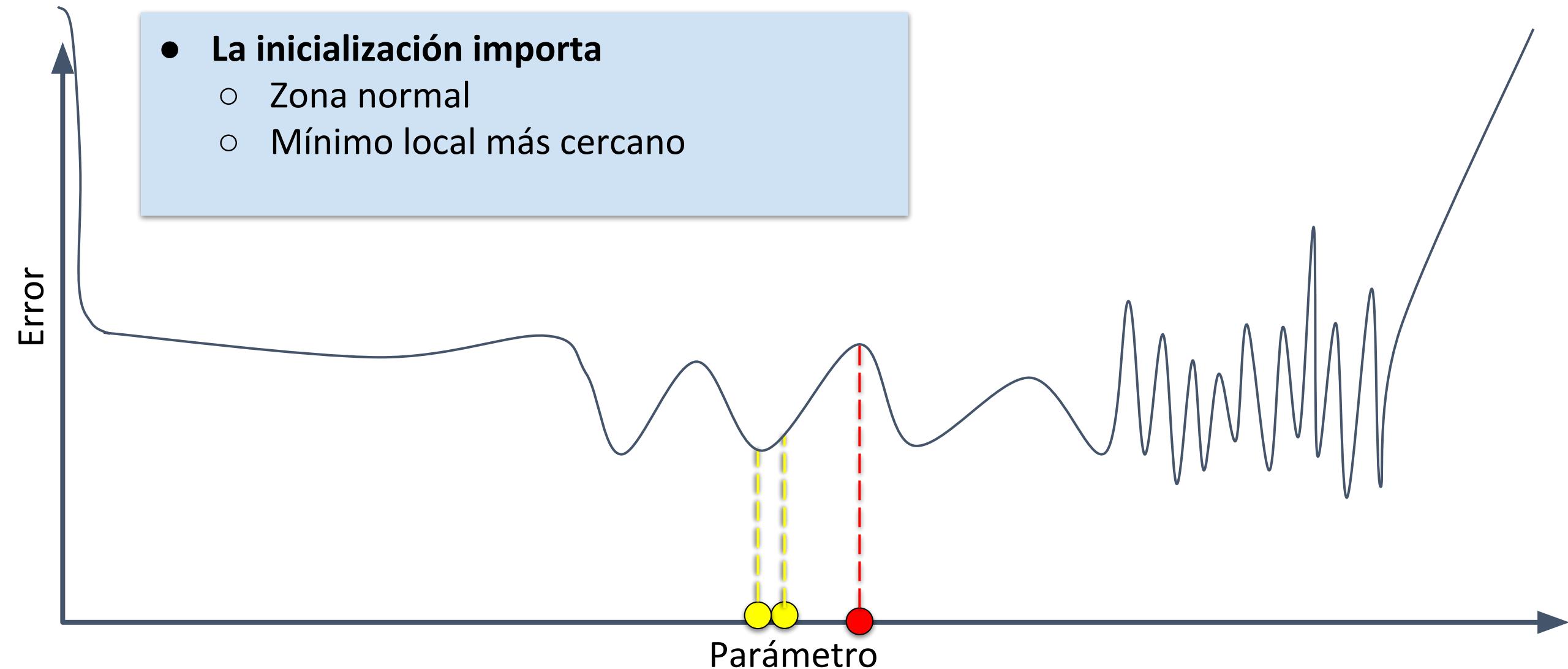
- La inicialización importa
 - Zona de pendientes altas
 - Cambios muy grandes



Inicialización de parámetros

- **La inicialización importa**

- Zona normal
- Mínimo local más cercano



Inicialización de parámetros

- Inicializar con un valor fijo

- Generalmente 0

- Funciona para sesgos B

- No funciona para matrices de pesos

- W** o filtros convolucionales

- Predice todo 0

- Las derivadas son 0

- No aprende

- Otro valor fijo (5, por ejemplo)

- Es posible que esté mal elegido

- Red no aprende

- Todas las neuronas predicen lo mismo

- **Mejor: Valor aleatorio**

$$x = (3, 2, 1)$$

$$\begin{matrix} W = 0 \\ B = 0 \\ \text{ReLU} \end{matrix}$$

$$(0, 0, 0)$$

$$\begin{matrix} W = 0 \\ B = 0 \\ \text{ReLU} \end{matrix}$$

$$(0, 0, 0)$$

$$\begin{matrix} W = 0 \\ B = 0 \\ \text{ReLU} \end{matrix}$$

$$(0, 0, 0)$$

$$x = (3, 2, 1)$$

$$\begin{matrix} W = 5 \\ B = 0 \\ \text{ReLU} \end{matrix}$$

$$(30, 30, 30)$$

$$\begin{matrix} W = 5 \\ B = 0 \\ \text{ReLU} \end{matrix}$$

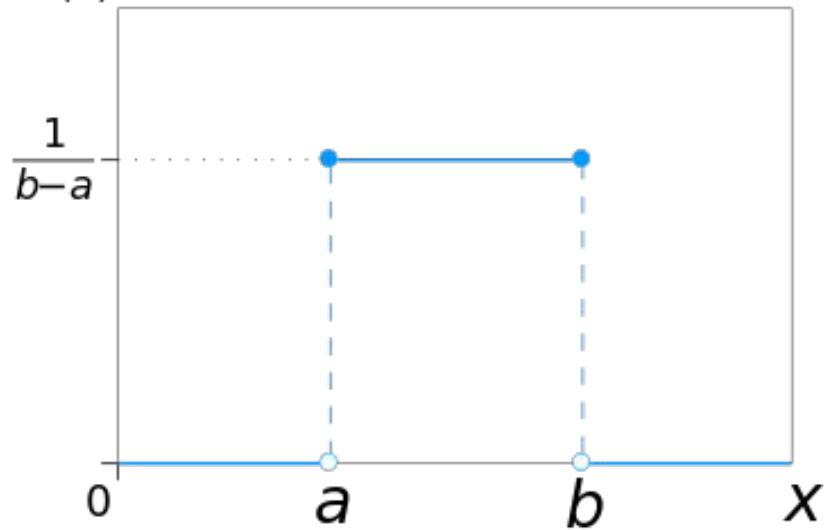
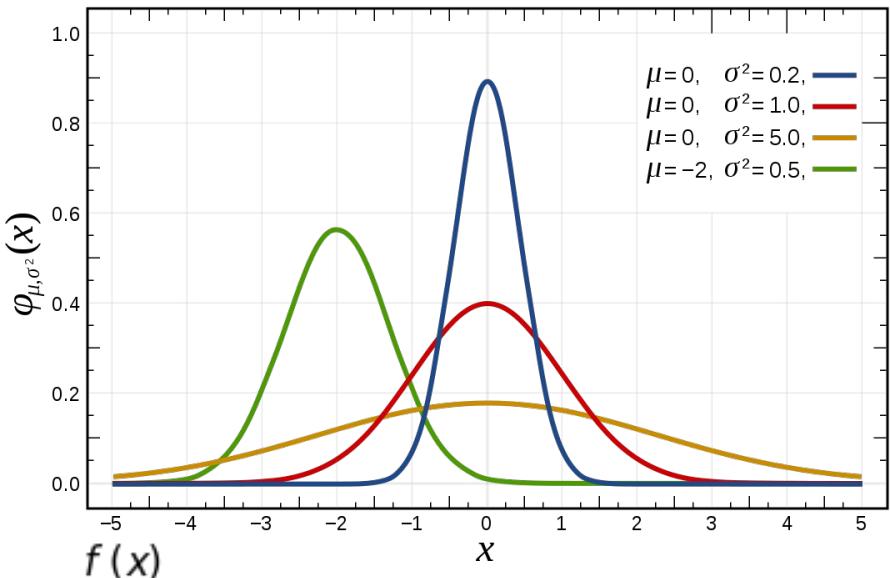
$$(450, 450, 450)$$

$$\begin{matrix} W = 5 \\ B = 0 \\ \text{ReLU} \end{matrix}$$

$$(2250, 2250, 2250)$$

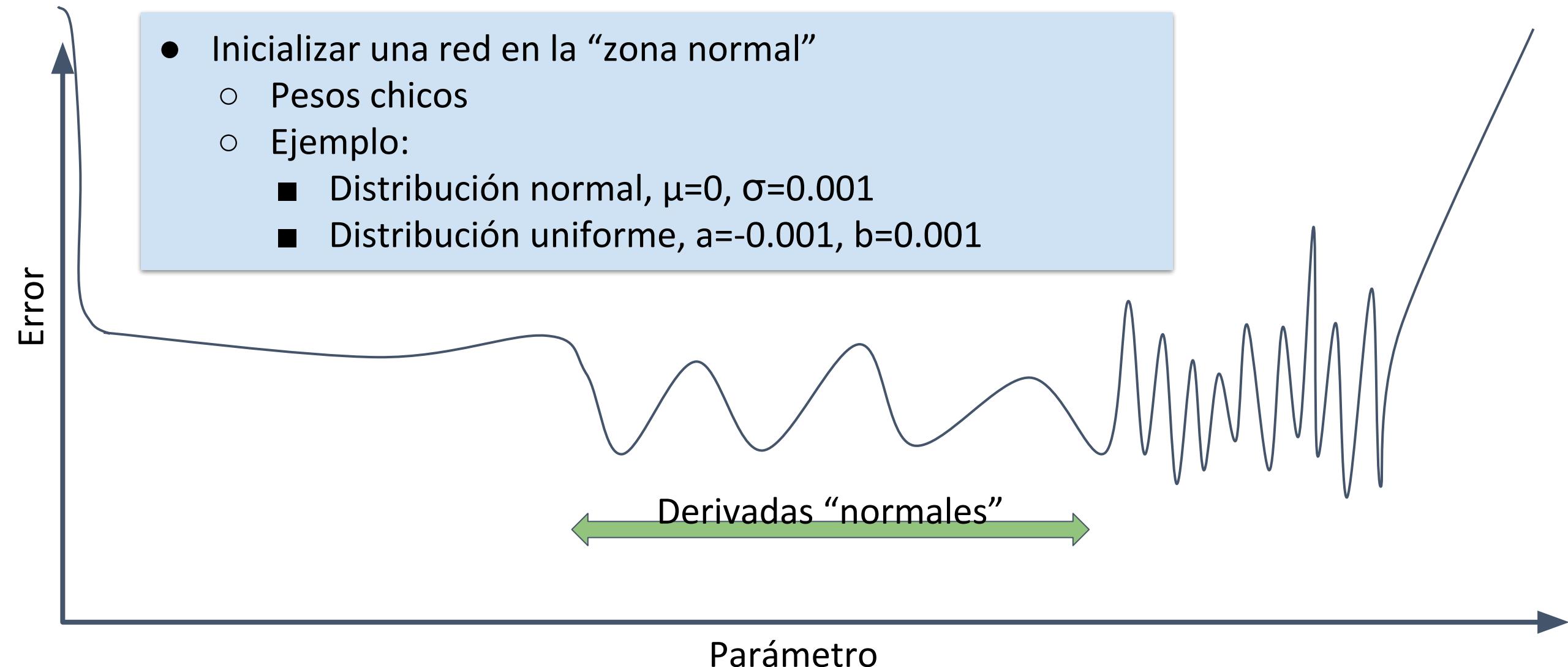
Inicialización de parámetros

- Inicializar con valor aleatorio
 - Valor aleatorio, generado con
 - Distribución normal o gaussiana
 - Elegir μ y σ^2 (hiperparámetros)
 - Distribución uniforme
 - Elegir rango del intervalo: a y b (hipérparámetros)
 - ¿El entrenamiento no funcionó?
 - Repetir inicialización
 - Volver a entrenar



Inicialización de parámetros

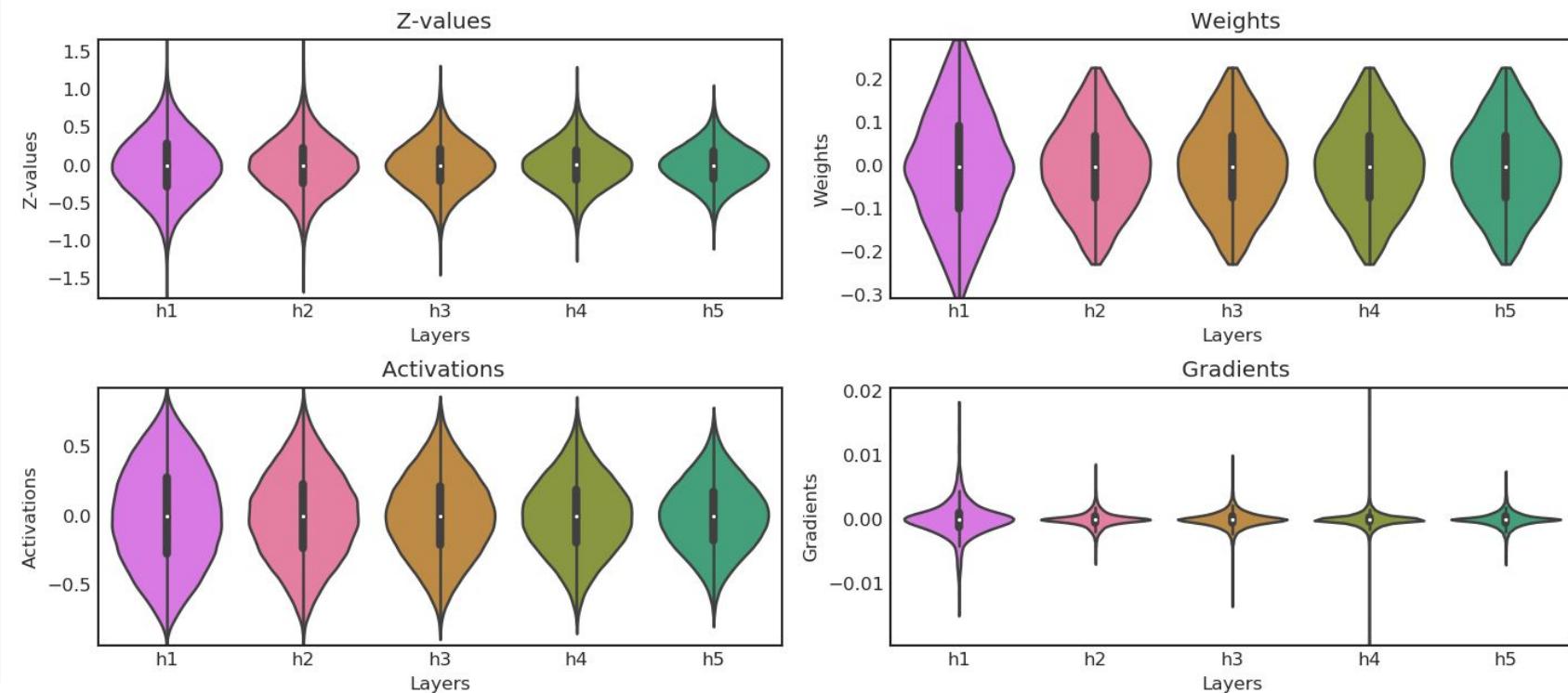
- Inicializar una red en la “zona normal”
 - Pesos chicos
 - Ejemplo:
 - Distribución normal, $\mu=0$, $\sigma=0.001$
 - Distribución uniforme, $a=-0.001$, $b=0.001$



Inicialización de parámetros

- Inicializadores avanzados
- Objetivo: magnitudes de los valores intermedios *estables*
 - *estable*:
 - No tienden a 0
 - Ni a valores muy grandes
 - => Derivadas estables
- Inic. más importantes
 - **Glorot: para capas con TanH**
 - **He: para capas con ReLu**

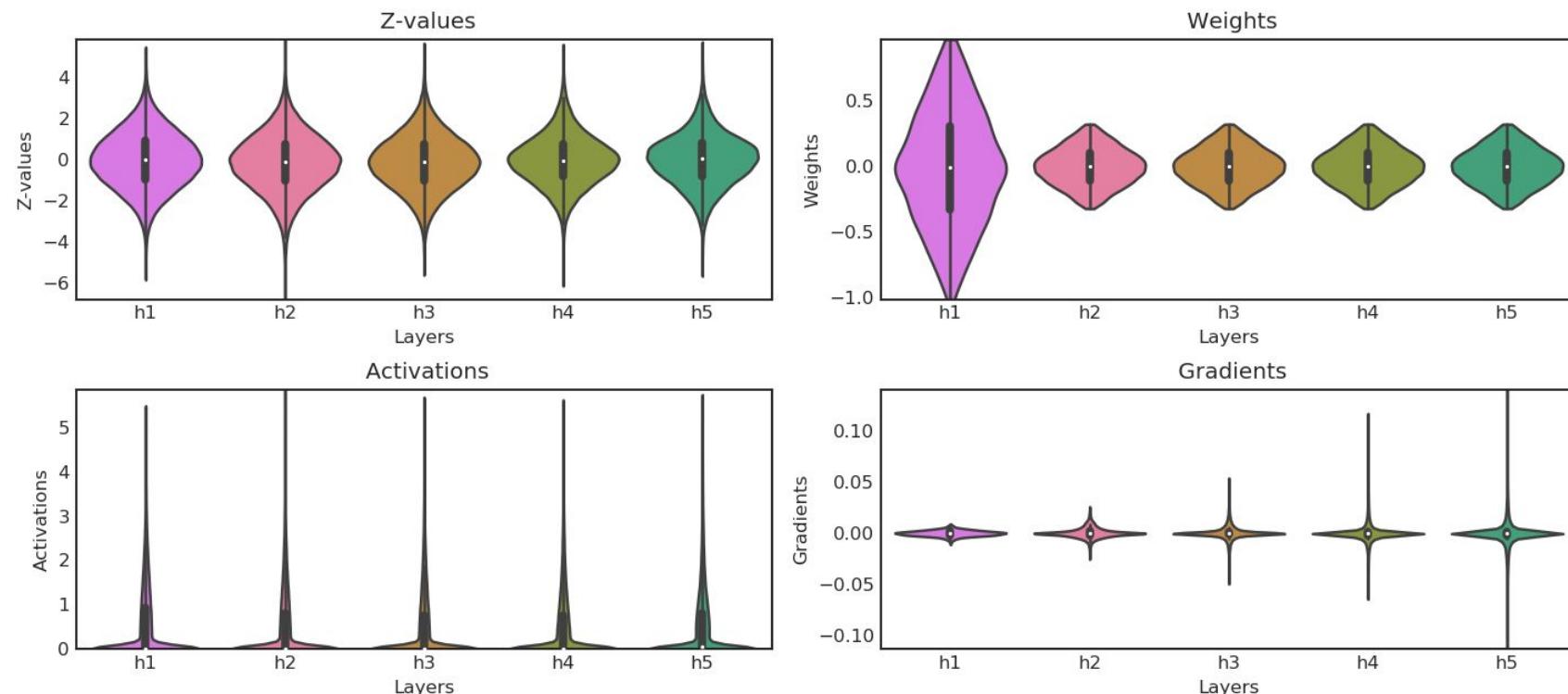
- **Glorot**: Normal con **media = 0** y
 - **Varianza = 2 / (#inputs+#outputs)**
 - **#inputs = neuronas capa anterior**
 - **#outputs = neuronas capa siguiente**



Inicialización de parámetros ([más info](#))

- Inicializadores avanzados
- Objetivo: magnitudes de los valores intermedios *estables*
 - *estable*:
 - No tienden a 0
 - Ni a valores muy grandes
 - => Derivadas estables
- Inic. más importantes
 - Glorot: para capas con TanH
 - **He: para capas con ReLu**

- He: Normal con **media = 0** y
 - **Varianza = 2 / (#inputs)**
 - **#inputs = neuronas capa anterior**



Inicialización de parámetros

- Inicializadores por defecto en keras
 - Clase Dense
 - kernel_initializer: inicializador de matriz de pesos (w)
 - bias_initializer: inicializador de vector de sesgo (b)
 - Clase Conv2D
 - kernel_initializer: inicializador de pesos de filtros convolucionales
 - bias_initializer: inicializador de vector de sesgo (b)
 - Por defecto '**glorot_uniform**' en pesos
 - Cambiar a '**he_uniform**' o '**he_normal**' si usás ReLu

```
Dense(...,kernel_initializer='glorot_uniform', bias_initializer='zeros',...)
```

```
Conv2D(...,kernel_initializer='glorot_uniform', bias_initializer='zeros',...)
```

Cronogramas de tasas de aprendizaje

Tasas de aprendizaje ([notebook](#))

- Enfoque actual:
 - α fija, valor bajo
- Problema
 - Optimización *difícil*
 - α chico
 - Optimización *fácil*
 - α grande
 - Difícil de predecir
- Soluciones
 - 1) Asumir: Al principio que es fácil (α alto), luego cada vez más difícil (disminuir α)
 - 2) Cílico: Subir y bajar constantemente

Visualización de Filtros

Visualización de Filtros

```
from keras.applications.vgg16 import VGG16  
model = VGG16()  
print(model.summary())
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	

block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000
<hr/>		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

Visualización de Filtros

```
for layer in model.layers:  
    if 'conv' not in layer.name:  
        continue  
    filters, biases = layer.get_weights()  
    print(layer.name, filters.shape)
```

Filter (h,w,cin,cout)

block1_conv1 (3, 3, 3, 64)
block1_conv2 (3, 3, 64, 64)
block2_conv1 (3, 3, 64, 128)
block2_conv2 (3, 3, 128, 128)
block3_conv1 (3, 3, 128, 256)
block3_conv2 (3, 3, 256, 256)
block3_conv3 (3, 3, 256, 256)

Filter (h,w,cin,cout)

block4_conv1 (3, 3, 256, 512)
block4_conv2 (3, 3, 512, 512)
block4_conv3 (3, 3, 512, 512)
block5_conv1 (3, 3, 512, 512)
block5_conv2 (3, 3, 512, 512)
block5_conv3 (3, 3, 512, 512)

- 64 filtros de 3x3x3
- 64 filtros de 3x3x64
- 128 filtros de 3x3x64
- 128 filtros de 3x3x128
- ...
- 512 filtros de 3x3x512

Visualización de Filtros

```
def layer_by_name(model,layer_name):
    for layer in model.layers:
        if layer.name==layer_name:
            return layer
    raise ValueError(f"Invalid layer name {layer_name}")
```

```
#Elegir capa convolucional y nro de filtro
layer_name="block1_conv2"
filter_index=0
#dibujar filtro
layer = layer_by_name(model,layer_name)
filters, biases = layer.get_weights()
plot_conv_weight(layer.name,filters[:,:,:,:,filter_index])
```

Visualización de Filtros

```
def plot_conv_weight(layer_name,filters,cols=32):
    h,w,c=filters.shape
    rows=( c // cols ) + int((c % cols)>0)
    gs = gridspec.GridSpec(rows, cols, wspace=0.1, hspace=0.1)
    # maximo y minimo para dibujar todos en la misma escala
    mi,ma=filters.min(),filters.max()
    for i in range(c):
        ax = plt.subplot(gs[i])
        ax.imshow(filters[:, :, :, i], cmap="gray", vmin=mi, vmax=ma)
        ax.set_xticks([])
        ax.set_yticks([])
    # poner en blanco los ax que sobran
    for i in range(c,cols*rows):
        ax = plt.subplot(gs[i])
        ax.axis("off")
```

[Ejemplo Completo](#)

Visualización de Filtros

- Verificación de aprendizaje correcto

- Capa `block1_conv2`
- `block1_conv2 (3, 3, 64, 64)`
- Primer filtro (`filter_index=0`)
- `shape = (3,3,64)`
- 64 cuadrados de 3x3
- c/u: pesos sobre el Feature Map anterior



Importante: No son todos 0 (filtros muertos)

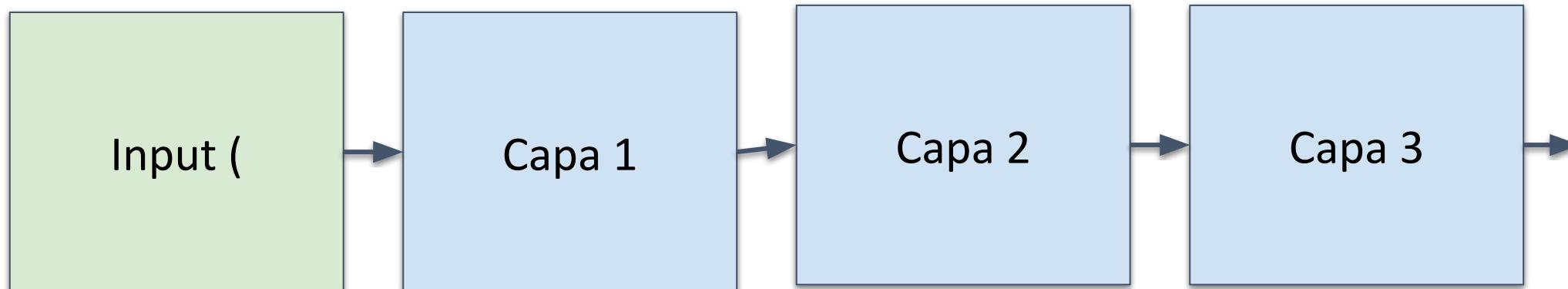
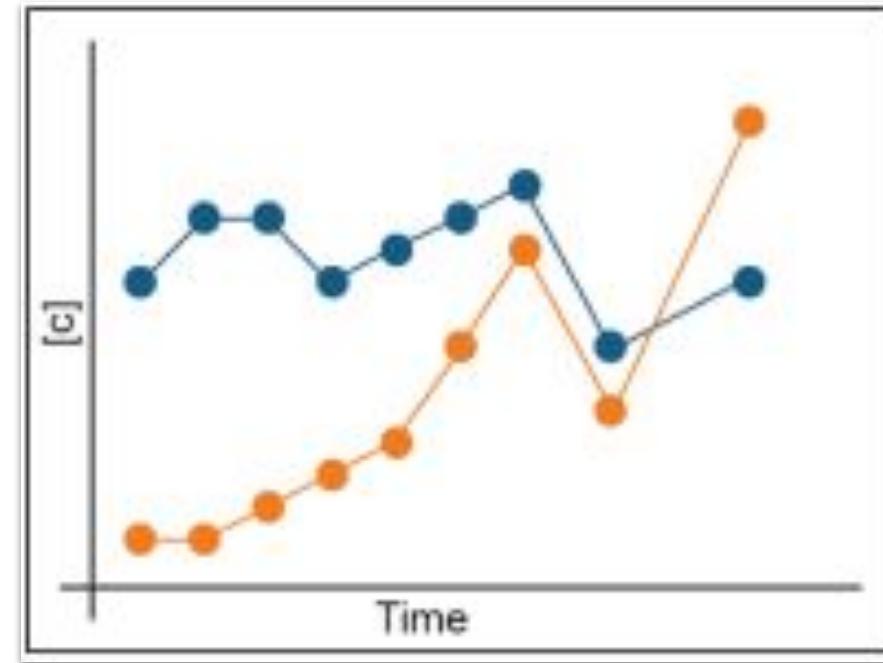
Aplicaciones y Extensiones de Redes Neuronales

Recurrent Neural Networks (RNNs)

Redes Neuronales Recurrentes

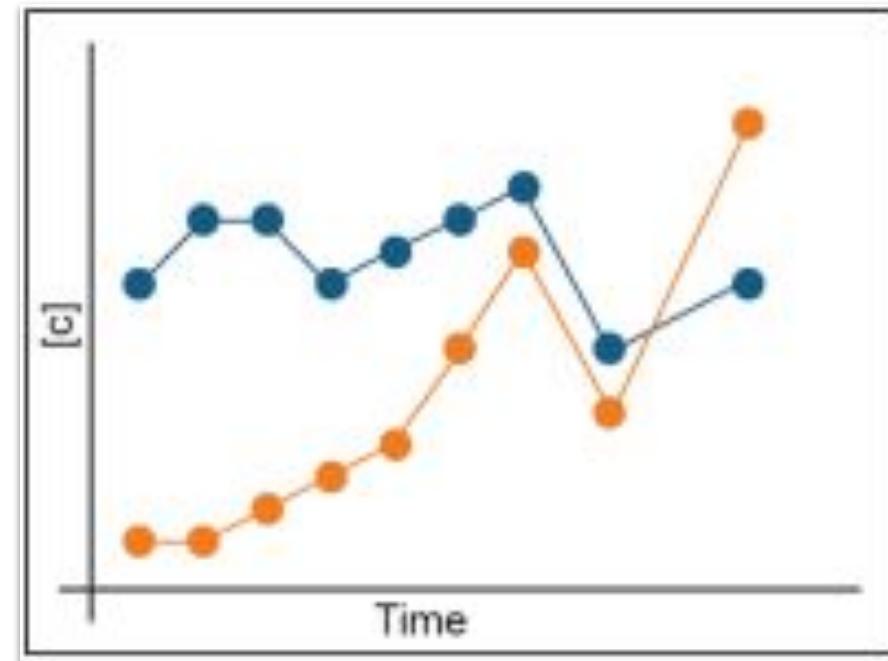
Redes Recurrentes - Motivación

- Hasta ahora
 - Problemas “estáticos”
 - Imágenes, tablas
 - Modelos “estáticos”
 - Tamaños de entrada y de salida
 - Fijos al diseñar la red



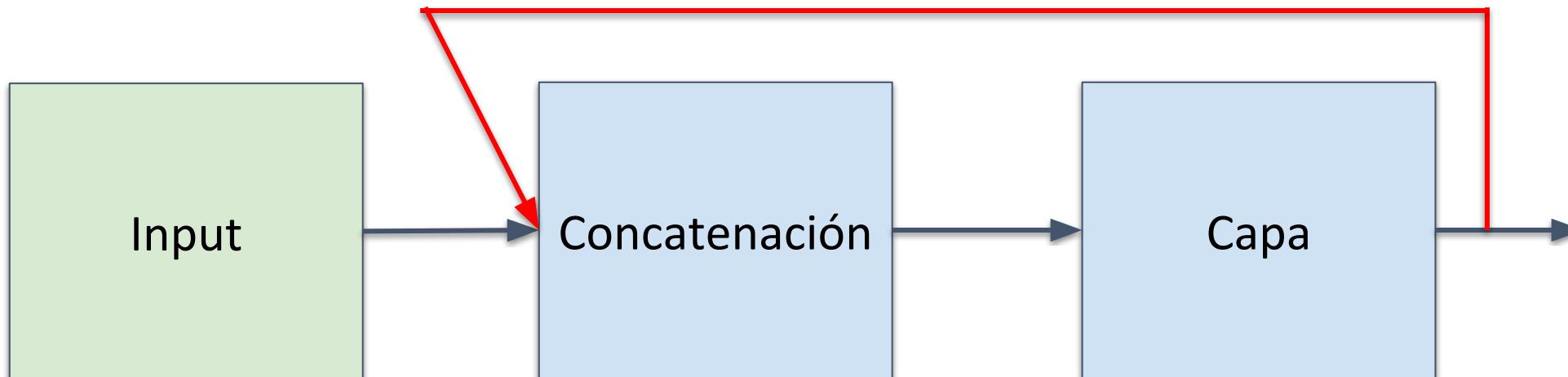
Redes Recurrentes - Motivación

- Problemas dinámicos
 - Series temporales: t muestras
 - Texto: p palabras
 - Videos: n imágenes o *frames*
 - n, t, p son variables => Longitud variable
 - Difícil de procesar con redes comunes de capas fijas
- Modelos dinámicos
 - **Redes neuronales recurrentes**



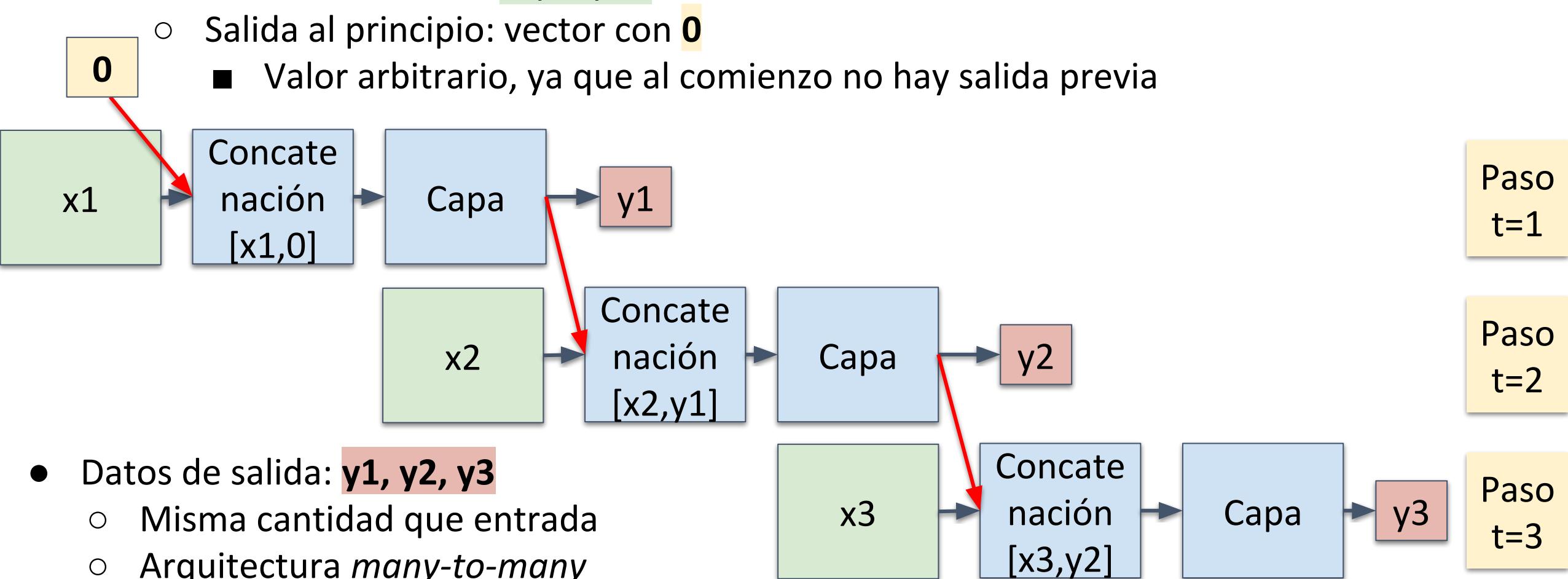
Redes Recurrentes - Definición

- Red Recurrente
 - Alguna **salida** de capa de red se **conecta** a la **entrada** de una capa anterior
 - La salida en el tiempo **t** es la entrada en el tiempo **t+1**
 - Ejecución por pasos de tiempo!
- Estado
 - Permite codificar el estado en la salida
 - Procesar datos



Redes Recurrentes - Ejecución

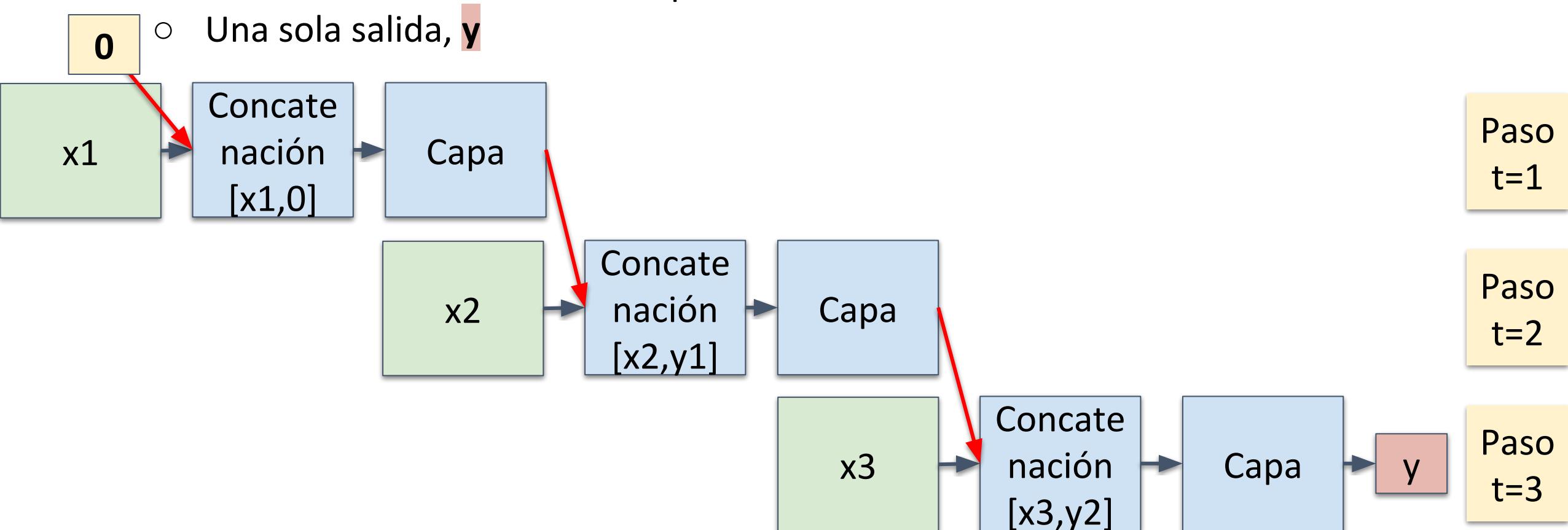
- Ejecución en pasos
 - Datos de entrada: x_1, x_2, x_3
 - Salida al principio: vector con 0
 - Valor arbitrario, ya que al comienzo no hay salida previa



- Datos de salida: y_1, y_2, y_3
 - Misma cantidad que entrada
 - Arquitectura *many-to-many*

Redes Recurrentes - Ejecución

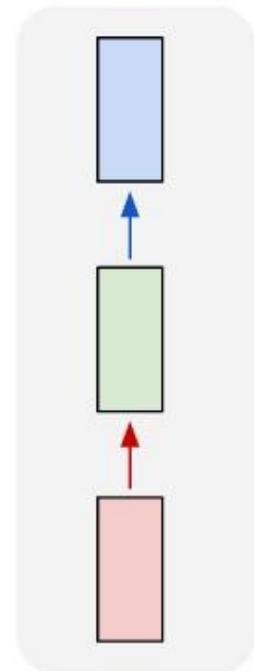
- Arquitectura *many-to-one*
 - Ignorar salidas intermedias
 - Usarlas como entrada en paso **t+1**
 - Una sola salida, **y**



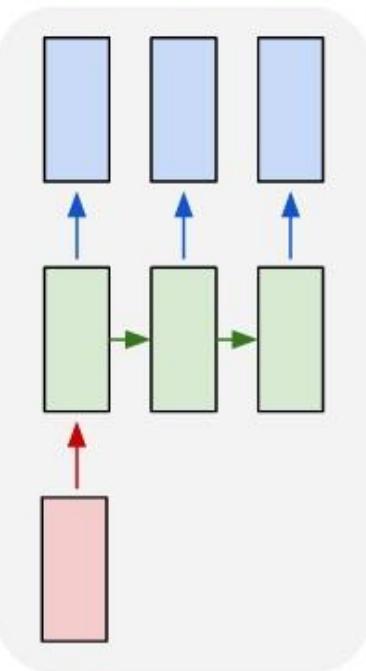
Redes Recurrentes

- Arquitecturas de redes recurrentes [1]

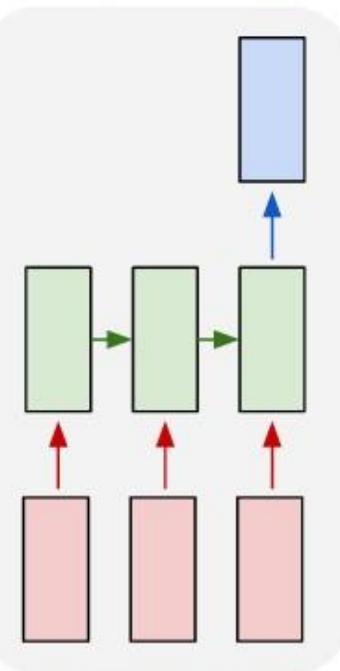
one to one



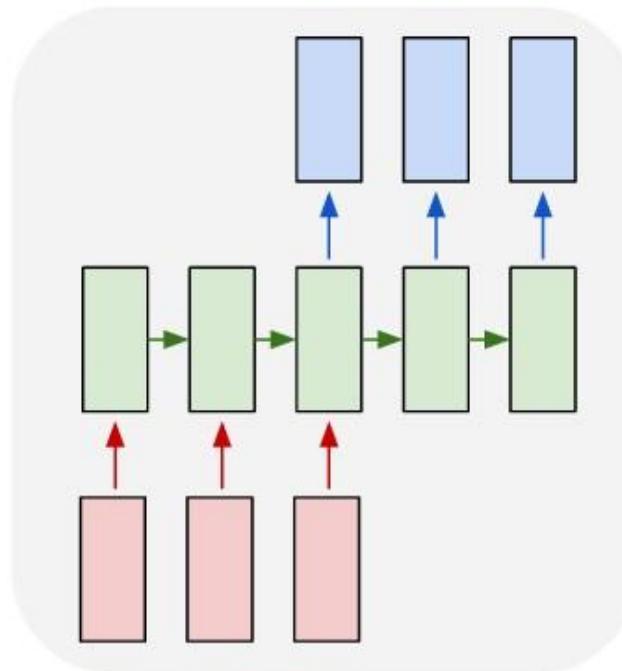
one to many



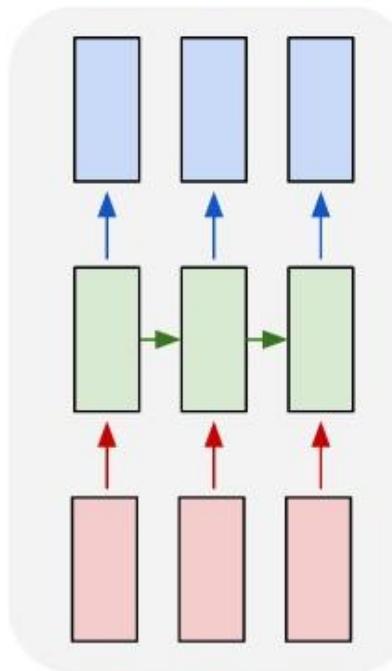
many to one



many to many



many to many



Redes comunes
(sin
recurrencia)

Generación de
series
temporales

Clasificación de
series
temporales

Transformación
de series
temporales

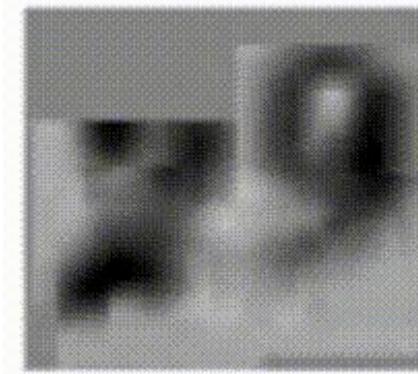
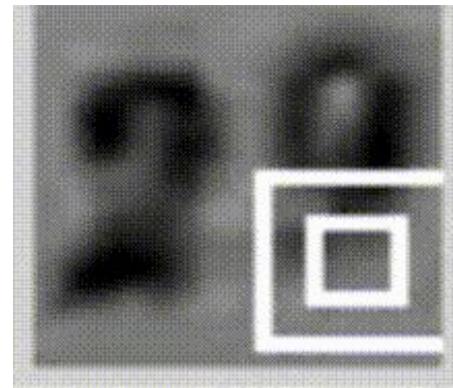
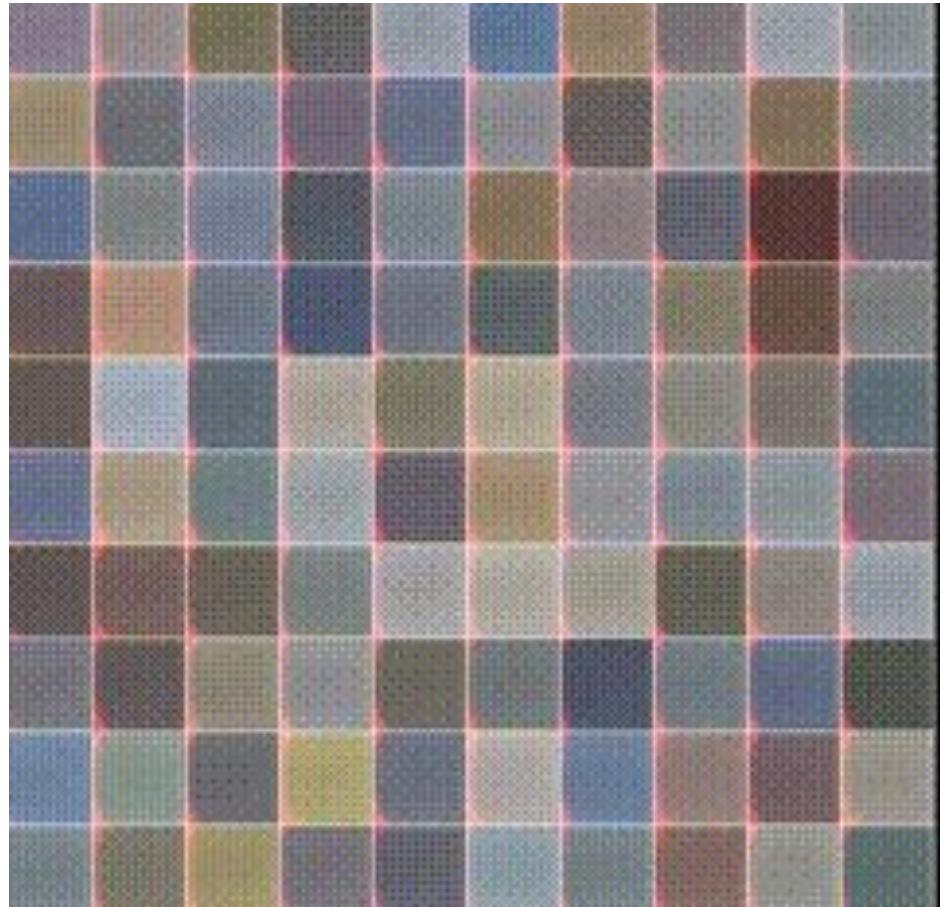
Transformación
de series
temporales

Redes Recurrentes - Propiedades

- Redes comunes
 - Aprenden **funciones**
- Redes recurrentes
 - Aprenden **programas**
 - Son turing-completas
 - Pueden simular programas arbitrarios
- Problema principal
 - Difícil de entrenar
 - Dependencias en el tiempo
- Muchas variantes para mejorar entrenamiento
 - Destacadas [1]
 - Long-Short Term Memory (LSTM)
 - GRU

Redes Recurrentes - Aplicaciones

- “Dibujo” de números y letras [1] [2]



Redes Recurrentes - Aplicaciones

- Generación de texto
 - Red que predice el próximo carácter
 - Generador de textos de Shakespeare [1]

PANDARUS:

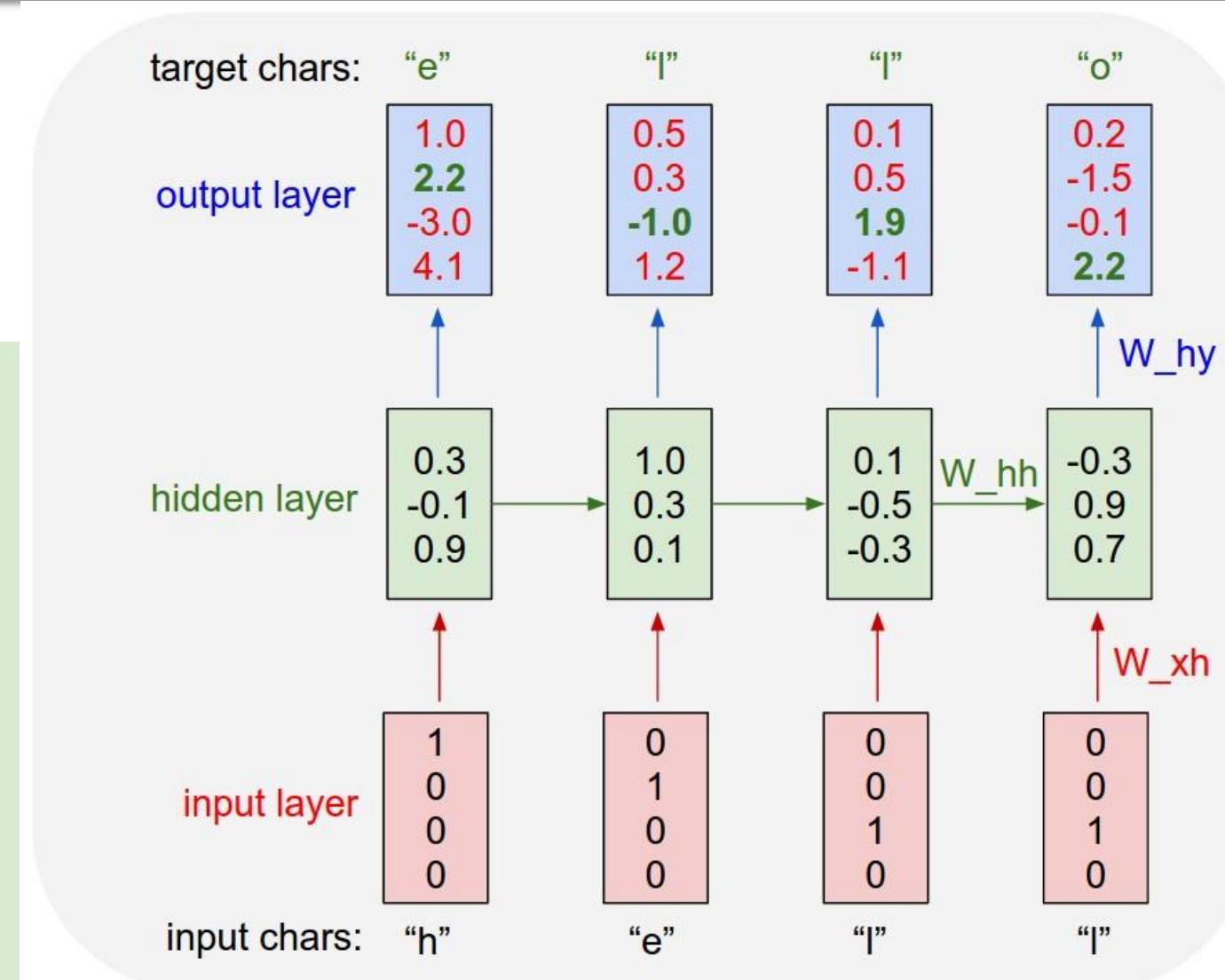
Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.



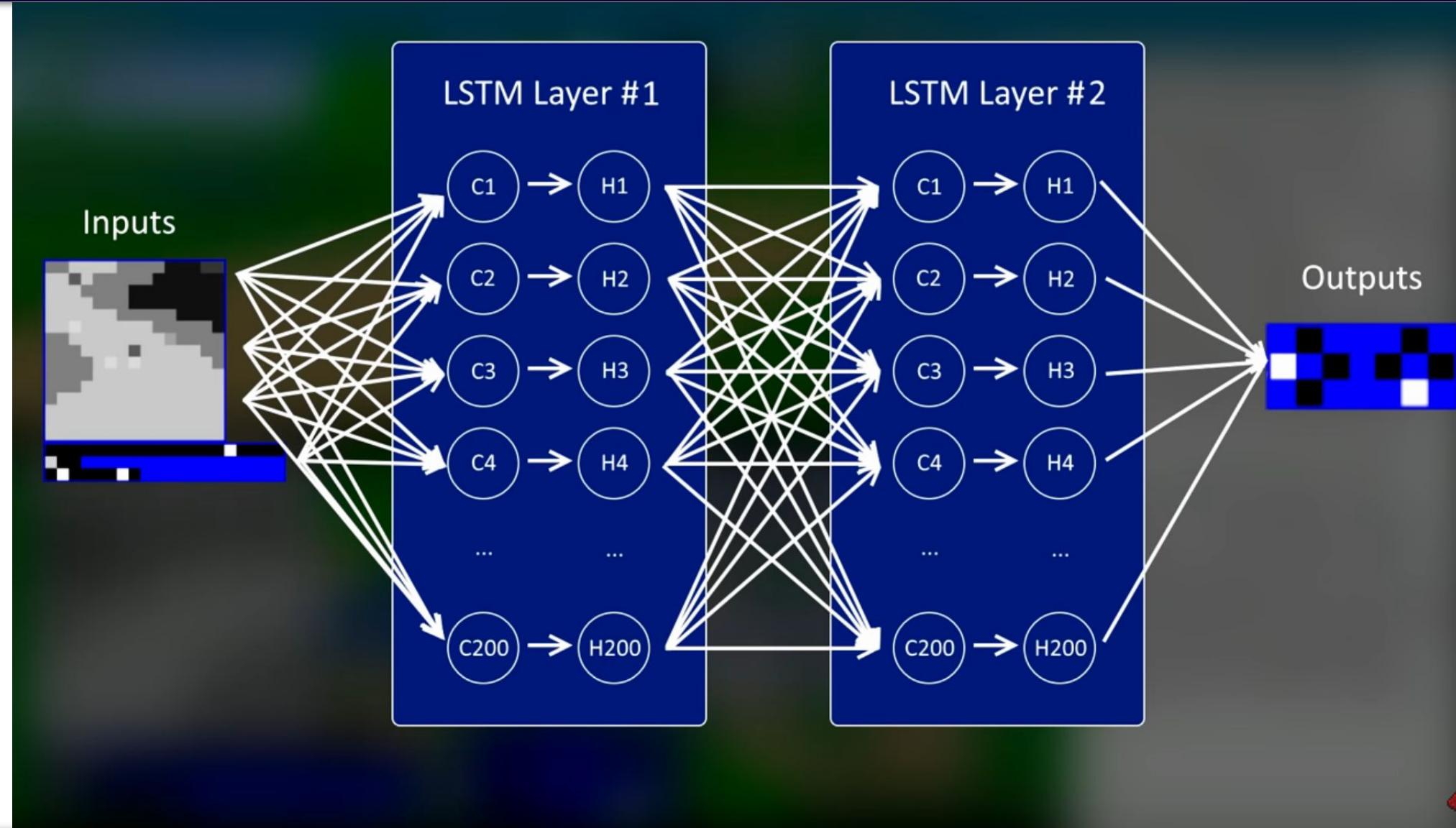
Redes Recurrentes - Aplicaciones

- [MariFlow](#)
- Controladores con RNNs



Redes Recurrentes - Aplicaciones

- [MariFlow](#)
- Entrada:
 - Imagen
- Salida
 - Botones
- 2 capas de LSTM



Generative Neural Networks

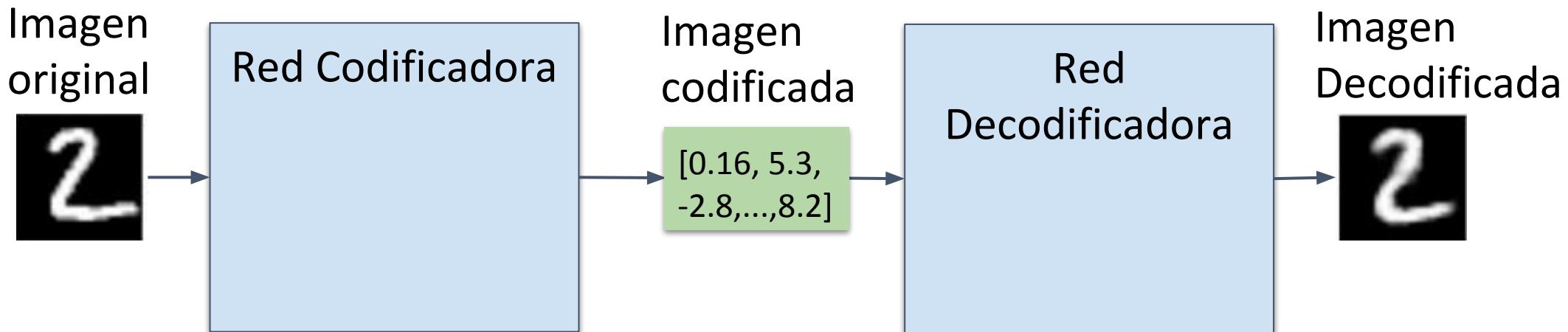
Redes Neuronales Generadoras

Autoencoders

Auto codificadores

Autoencoders (Auto codificadores)

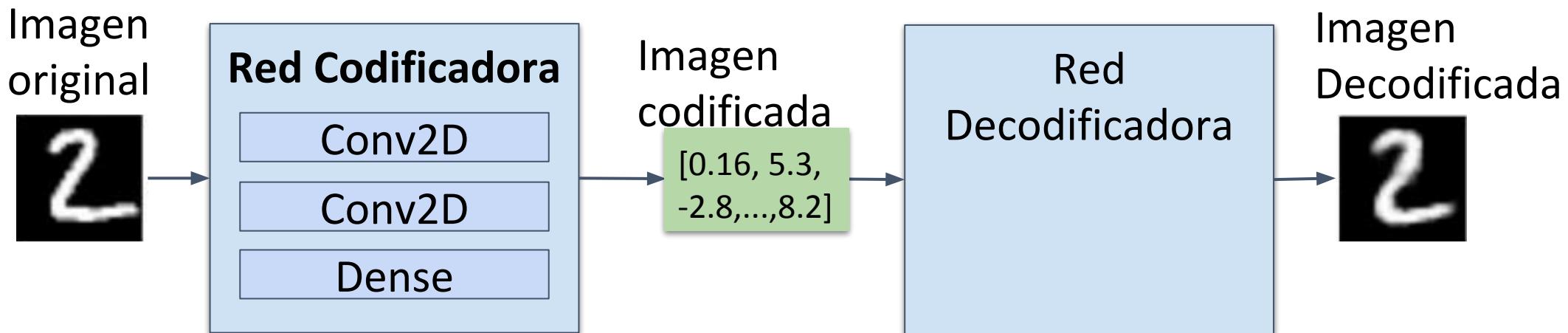
- Entrenamiento de AutoEncoder
 - Input: Imagen
 - Output: *Misma* imagen!
 - Modelo **NO** supervisado
 - No utiliza etiquetas
 - Aunque utiliza *mecanismos* supervisados
 - Tres modelos: encoder, decoder, autoencoder = decoder(encoder(x))



Autoencoders (Auto codificadores)

- **Red codificadora**

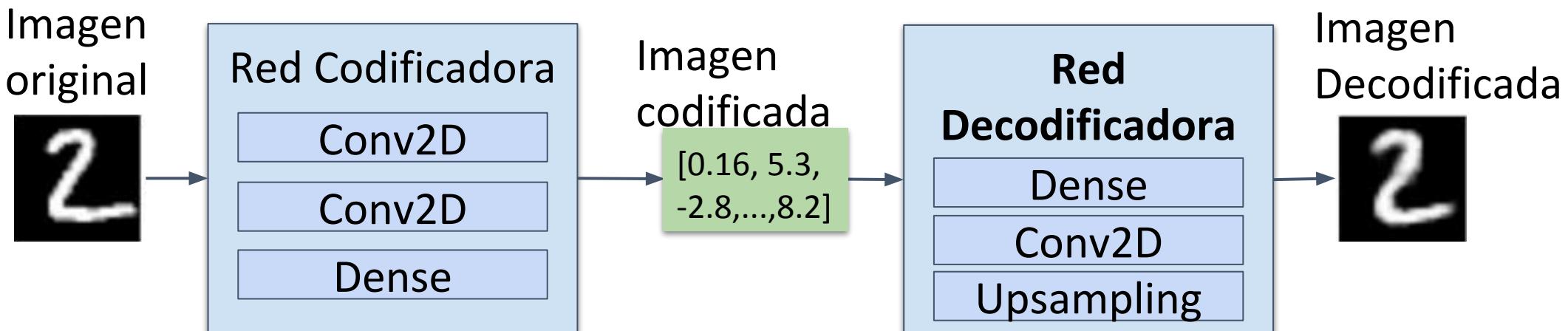
- Genera un vector con K elementos
 - K es arbitrario
- $K \ll$ dimensión de la imagen original
- Vector comprime la imagen
 - Autoencoders para compresión
- Es una red común (CNN o Dense)



Autoencoders (Auto codificadores)

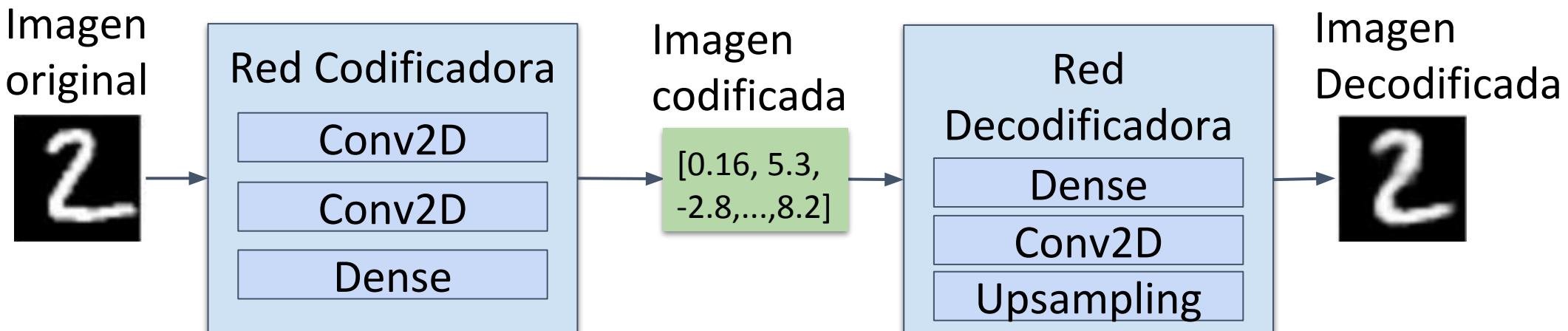
- Red **decodificadora**

- Genera una imagen
 - Mismo tamaño que imagen de entrada
- *Descomprime* la imagen
 - Entrada: vector de tamaño K
 - Salida: Imagen
- Es una red común (CNN o Dense)
 - Diseño espejo a la codificadora



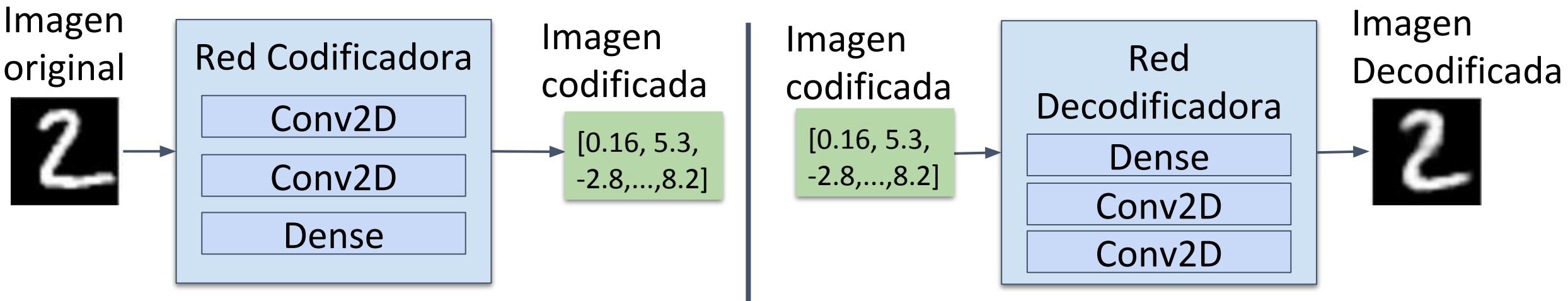
Autoencoders (Auto codificadores)

- Imagen codificada
 - Se llama vector latente (histórico)
 - Vector de tamaño K
 - K arbitrario:
 - más grande
 - mayor poder de representación
 - menor compresión



Autoencoders (Auto codificadores)

- Entrenamiento
 - `model.fit(x_train,x_train)`
 - Error: mse o entropía cruzada
 - Entre *imágenes*!
- Compresión con red codificadora
 - `codigo = encoder.predict(imagen)`
- Descompresión con red decodificadora
 - `imagen_restaurada = decoder.predict(codigo)`



Auto Codificadores para Comprimir ([notebook](#))

- Utiliza el vector latente como compresión de la imagen
- Código solo sirve para el tipo de datos de entrenamiento
 - No es un compresor universal como JPEG o PNG

```
def load_data():
    (x_train, _), (x_test, _) = mnist.load_data()
    input_dim = 28*28
    x_train = np.reshape(x_train, [-1, input_dim])/255.0
    x_test = np.reshape(x_test, [-1, input_dim])/255.0

    return x_train,x_test,input_dim
```

Auto Codificadores para Comprimir ([notebook](#))

- Utiliza el vector latente como compresión de la imagen
- Código solo sirve para el tipo de datos de entrenamiento
 - No es un compresor universal como JPEG o PNG
- Primera versión con modelos tradicionales (capas Dense)

```
def load_data():

    (x_train, _), (x_test, _) = mnist.load_data()

    input_dim = 28*28

    x_train = np.reshape(x_train, [-1, input_dim])/255.0

    x_test = np.reshape(x_test, [-1, input_dim])/255.0

    return x_train,x_test,input_dim
```



Auto Codificadores para Comprimir ([notebook](#))

```
def DenseAutoencoder(input_dim,latent_dim):
    def generate_encoder():
        encoder_input = Input(shape=(input_dim,), name='encoder_input')
        code = Dense(latent_dim, name='latent_vector')(encoder_input)
        encoder = Model(encoder_input, code, name='encoder')
        return encoder,encoder_input

    def generate_decoder():
        latent_input = Input(shape=(latent_dim,), name='decoder_input')
        decoded_image = Dense(input_dim,activation="sigmoid",name='decoder_output')(latent_input)
        decoder = Model(latent_input, decoded_image, name='decoder')
        return decoder

    encoder,encoder_input = generate_encoder()
    decoder = generate_decoder()
    autoencoder = Model(encoder_input, decoder(encoder(encoder_input)), name='autoencoder')
    return autoencoder,encoder,decoder
```

Auto Codificadores para Comprimir ([notebook](#))

```
x_train,x_test,input_dim = load_data()
latent_dim = 64
autoencoder,encoder,decoder=DenseAutoencoder(input_dim,latent_dim)
autoencoder.compile(loss='binary_crossentropy', optimizer='adam')

autoencoder.fit(x_train,x_train,
                 validation_data=(x_test, x_test),
                 epochs=10, batch_size=128)

x_decoded = autoencoder.predict(x_test)
compare_images(x_test,x_decoded)
```



AC Convolucionales para Comprimir ([notebook](#))

- **Codificador** usa capas Conv2D en lugar de Dense
 - Vector latente tiene tamaño (K, L, M)

```
encoder_input = Input(shape=input_shape, name='encoder_input')
```

```
x = Conv2D(16, (3, 3), activation='relu', padding='same')(encoder_input)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
encoded_shape = K.int_shape(x)[1:]
```

```
encoder = Model(encoder_input, x, name='encoder')
```



AC Convolucionales para Comprimir ([notebook](#))

- **Decodificador** usa capas *Upsample*
 - Capa final con 1 filtro genera imagen de 1 canal

```
latent_input = Input(shape=encoded_shape, name='decoder_input')
```

```
x = Conv2D(8, (3, 3), activation='relu', padding='same')(latent_input)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
```

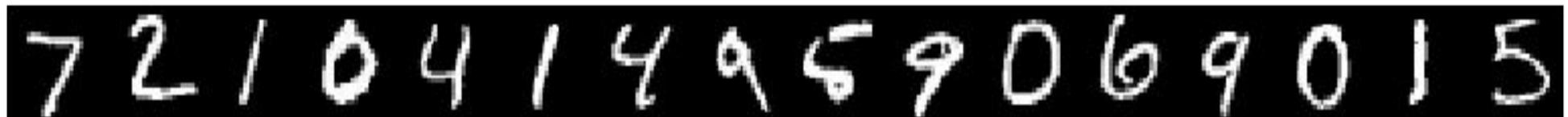
```
decoder = Model(latent_input, x, name='decoder')
```



Denoising Autoencoder ([notebook](#))

- Autoencoder para quitar ruido
 - `x_noise = x + np.random.noise(...)`
 - **model.fit(x_noise,x)**
 - Aprende a convertir imágenes con ruido en imágenes sin ruido

Originales



Con ruido



Restauradas



Denoising Autoencoder ([notebook](#))

- Carga de datos y generación de muestras con ruido
- Se modifica a las originales con ruido gaussiano
 - valores de una normal con media 0.5 y desviación 0.1

```
def load_data(noise_location,noise_strength):  
    (x_train, _), (x_test, _) = mnist.load_data()  
  
    image_size = x_train.shape[1]  
    x_train = np.reshape(x_train, [-1, 28,28, 1])/255.0  
    x_test = np.reshape(x_test, [-1, 28,28, 1])/255.0  
    input_shape = (28,28, 1)  
  
    # Genera muestras de MNIST corrompidas por el ruido  
    # centrado en 0.5 y con fuerza 0.1  
    noise = np.random.normal(loc=0.5, scale=0.1, size=x_train.shape)  
    x_train_noisy = x_train + noise  
    noise = np.random.normal(loc=0.5, scale=0.1, size=x_test.shape)  
    x_test_noisy = x_test + noise  
  
    x_train_noisy = np.clip(x_train_noisy, 0., 1.) # restrinjo al rango 0-1  
    x_test_noisy = np.clip(x_test_noisy, 0., 1.) # restrinjo al rango 0-1  
    return x_train,x_train_noisy,x_test,x_test_noisy,input_shape
```

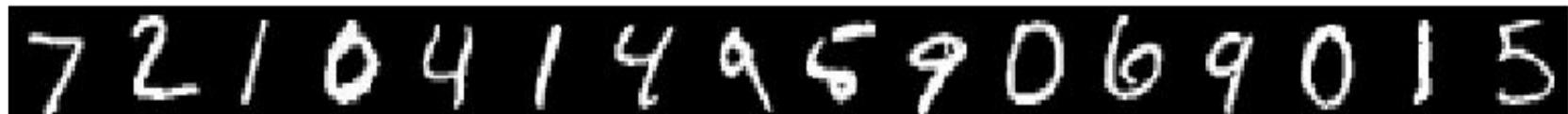
Denoising Autoencoder ([notebook](#))

```
autoencoder.fit(x_train_noisy,  
                 x_train,  
                 validation_data=(x_test_noisy,x_test),  
                 epochs=30,  
                 batch_size=batch_size)
```

- Entrenamiento del modelo con valores ruidosos como input y originales como output

- Evaluar el modelo con una imagen ruidosa para quitar ruido
- `x_restaurado = autoencoder.predict(x_ruidoso)`

Originales



Con ruido



Restauradas



Generative Adversarial Networks (GANs)

Redes Generadoras Adversariales

Redes Generativas Adversariales

Click on the person who is real.



Redes Generativas Adversariales

You are **correct**. The image on the left is real.

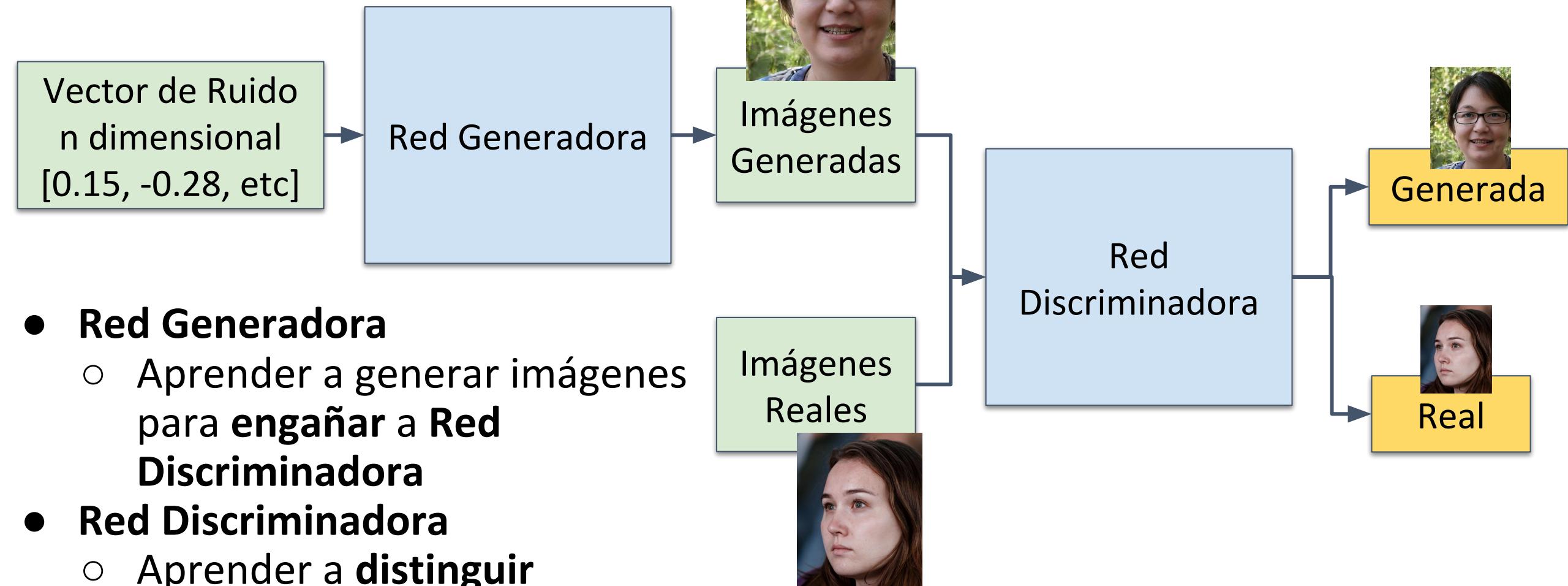
[Play again.](#)



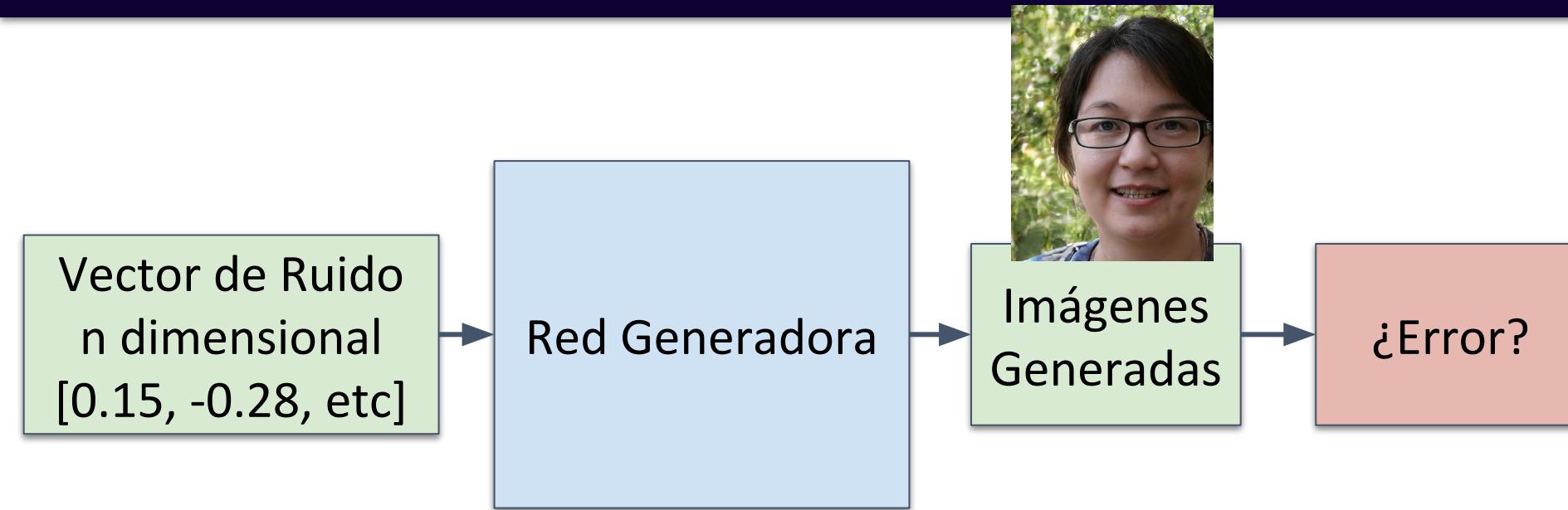
Redes Generativas Adversariales



Redes Generativas Adversariales



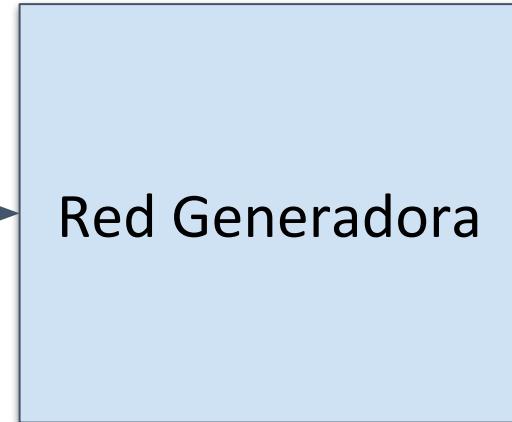
Redes Generativas Adversariales



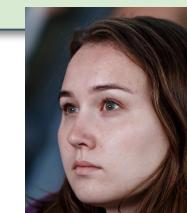
- **Red Generadora sola**
 - Difícil de entrenar
 - Red Discriminadora provee gradiente de error útil para la Red Generadora

Redes Generativas Adversariales

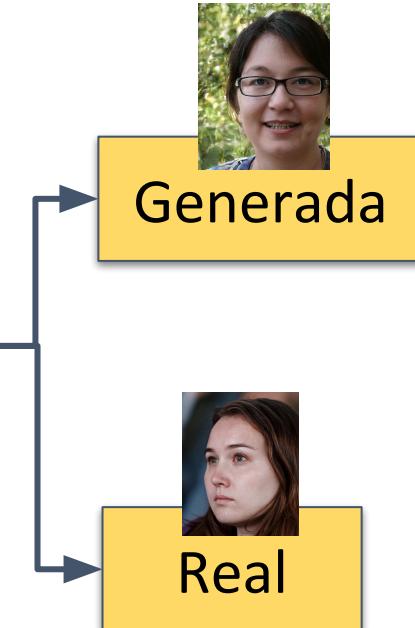
Vector de Ruido
n dimensional
[0.15, -0.28, etc]



Imágenes Generadas



Red Discriminadora



Generada



Real

Entrenamiento, repetir N épocas:

1. Generar K imágenes falsas con **Red Generadora**
2. Seleccionar K imágenes reales
3. Entrenar **Red Discriminadora** para distinguir las K+K imágenes
4. Entrenar **Red Generadora** para engañar a **Red Discriminadora**

Redes Generativas Adversariales ([notebook](#))

- Pseudocódigo entrenamiento

```
generadora = ...  
discriminadora = ...  
GAN = discriminadora(generadora(x))
```

```
for i in range(epochs):  
    Generar K vectores de ruido  
    Generar K imágenes falsas con generadora y el ruido  
    Seleccionar K imágenes verdaderas del training set  
    Entrenar discriminadora con los K+K ejemplos  
    Congelar los pesos de discriminadora  
    Entrenar red GAN (entera) para engañar discriminadora  
        (solo entrena pesos de generadora)  
    Descongelar los pesos de discriminadora
```