

Computer Vision

Deep Learning en Computer Vision

Marcos Zuñiga

Departamento de Electronica

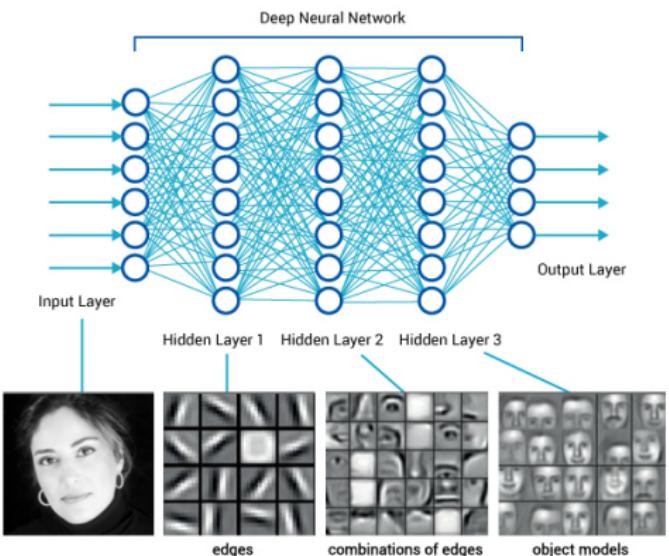
27 de Mayo 2020



UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Deep Learning

- **Deep learning:** clase de algoritmos de machine learning que utiliza múltiples capas para extraer de forma progresiva características de más alto nivel desde datos de entrada.
- Se basan en las redes neuronales artificiales. El apellido *deep* viene de la cantidad de capas que consideran.
- Cada capa va transformando los datos en una representación cada vez más abstracta y compuesta (e.g. imagen (pixels) → bordes → conjuntos de bordes → nariz/ojos → cara).
 - Un proceso de deep learning puede aprender qué características posicionar de forma óptima en qué capa.
 - En adelante hablaremos de Deep Neural Networks (DNN).



Aprendizaje Supervisado VS No Supervisado

- **Supervisado:**

- Dados X (entradas), Y (objetivos - etiquetas)
- Aprender a predecir el objetivo dada la entrada.

- **No Supervisado:**

- Dado sólo X (entradas)
- Aprender estructura de los datos, estimación de densidad ($p(X)$), clustering.



Aprendizaje Supervisado: Clasificación VS Regresión

- **Clasificación:**

- Etiquetas son discretas (enteros de 0 a K-1, con K el número de clases).
- E.g.: Llueve o no llueve mañana.

- **Regresión:**

- Etiquetas son números reales.
- E.g.: Predecir los mm de lluvia caída mañana.



Aprendizaje Supervisado: Generalización y Overfitting

Generalización

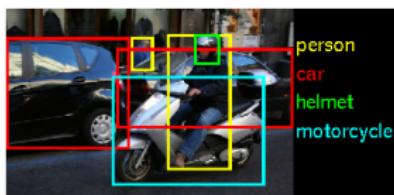
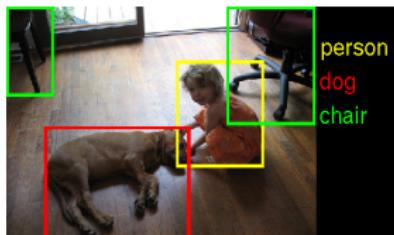
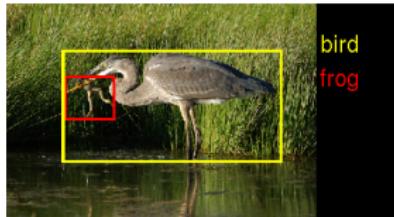
Capacidad de un modelo de predecir con precisión no sólo los datos con los que fue entrenado, sino que también los datos que no le habían sido presentados antes.

- Usualmente se separan los datos en conjuntos de entrenamiento y testing, para evaluar la capacidad de generalización del modelo.
- Si se tiene 100% de precisión con el conjunto de entrenamiento, pero sólo 60% con el de testing, significa que **generaliza** pobremente y sufre de **overfitting**.
- Overfitting se considera un error de modelado, que ocurre cuando éste ajusta con demasiada precisión un conjunto limitado de datos.



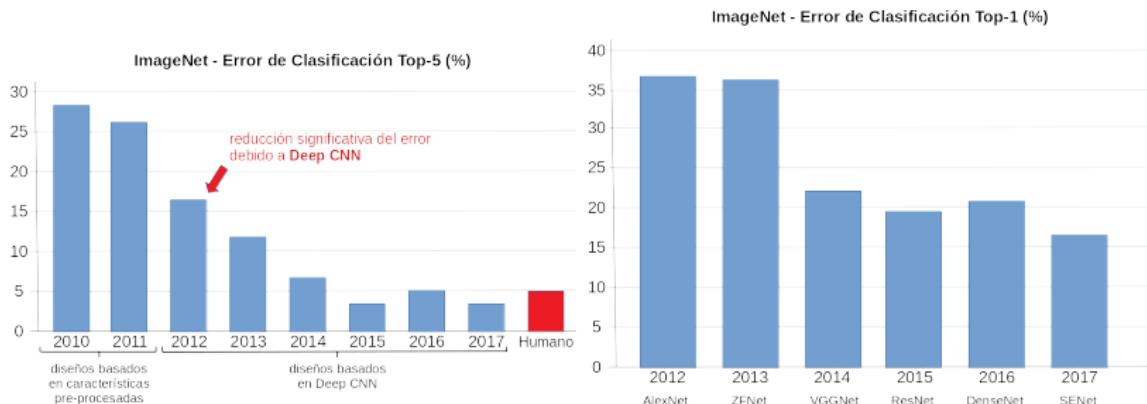
Deep Learning en Computer Vision: Avances - Clasificación

- Existen diversos datasets que se han utilizado como benchmark para la evaluación de modelos de DNN: MNIST, CIFAR-10 e ImageNet [Russakovsky et al., 2015].
- ImageNet: Competición desde 2010. Objetivo es estimar el contenido de fotografías usando un subset del dataset anotado (10,000,000+ imágenes con 10,000+ categorías de objetos) como entrenamiento. Imágenes de prueba presentadas sin anotación inicial (etiquetas). Algoritmos tendrán que producir etiquetas de los objetos en las imágenes.



Deep Learning: Clasificación - Imagenet

- En la categoría clasificación de Imagenet (igual del 2014), con 200 clases, 500k imágenes de entrenamiento, 20K validación, 40k testing.
- Las métricas Top-1 y Top-5 se han usado en la literatura para comparar la evolución de los algoritmos de clasificación.



Componentes para Deep Learning

- **Bibliotecas:**

- Caffe - (Berkeley Vision Lab) - casi descontinuado.
- **TensorFlow** - (Google)
- CNTK - (Microsoft) - descontinuado.
- Torch - (Facebook) - descontinuado.
- **PyTorch** - (Facebook)
- Theano - (MILA) - descontinuado.
- MXNet - Apache Software Foundation
- **Keras** - (Sobre TensorFlow, iniciativa individual + respaldo de Google)

- **Modelos:** Sistema completo end-to-end para desarrollar una tarea de visión bien definida.

- FRCNN, Mask-RCNN; SSD, YOLO, RetinaNet (detección/segmentación).
- FCNN (Fully Convolutional, segmentación).
- RNN, GRU, LSTM.

- **Redes/Arquitecturas:** Una red neuronal con capas convolucionales, recurrentes o ambas, para extraer características de una imagen.

- VGG16, Alexnet, Siamese, ResNet, Inception, Inception-Resnet, DenseNet.

Datasets para Deep Learning

Imágenes:

- MNIST Handwritten Digits [LeCun et al., 1998] - 28x28 gs [1998 - 2012]
10 clases / 60K imágenes entrenamiento / 10K imágenes testing
- CIFAR-10 - CIFAR-100 [Krizhevsky, 2009] - 32x32 color
CIFAR-10 - 10 clases/ 50K entrenamiento / 10K testing
CIFAR-100 - 100 clases/ 50K entrenamiento / 10K testing
- Imagenet [Russakovsky et al., 2015] [2009 - 2020]
22K clases / 15M imágenes
- Pascal VOC - challenge [Everingham et al., 2012] [2005 - 2012]
20 clases / 11.5K imágenes
- MS COCO [Tsung-Yi et al., 2014] [2014 - 2019]
90 clases / 183K imágenes / detección, segmentación, keypoints
- OpenImages [Krasin et al., 2017] - débilmente anotado [2017 - 2019]
600 clases / 1.9M imágenes / detección
- Fashion MNIST [Xiao et al., 2017] [2017 - 2020]
10 clases / 60K imágenes entrenamiento / 10K imágenes testing
Si no funciona con este dataset, simplemente no funciona

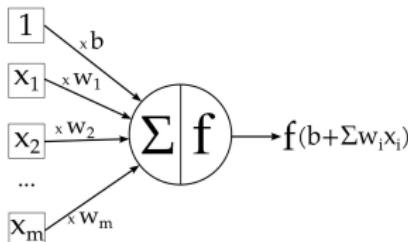
Datasets para Deep Learning

Video (reconocimiento de actividades):

- Kinetics [Carreira et al., 2019] [2017-2019]
400-600-700 clases de acciones / 325-650K video clips
- ActivityNet-200 [Caba H. et al., 2015]
200 clases de acciones / 20K videos / 648h / 31K instancias de acción
- MSRDailyActivity3D [Wang et al., 2012] - anotado con Kinect (esqueleto) [2012]
16 clases de acciones / 320 video clips
- NTU RGB+D [Shahroudy et al., 2016] [2016 - 2019]
60 clases de acciones / 57K videos (2016)
→ 120 clases de acciones / 120K videos (2019)
- Toyota Smarthome [Das et al., 2019] - RGBD + esqueleto [2019 - 2020]
31 clases de acciones / 16K videos (2019)
→ 53 clases de acciones / 536 videos / 41K instancias de acción (2020)

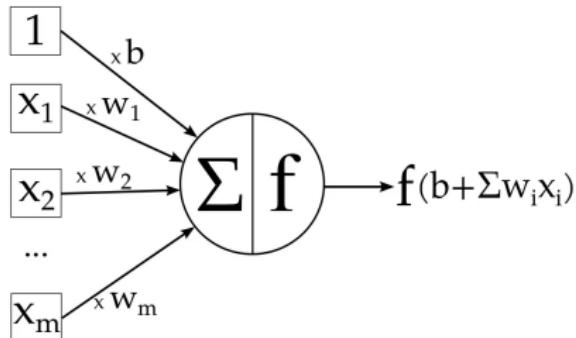
Conceptos Básicos: Redes Neuronales

- Las redes neuronales son uno de los algoritmos más populares de machine learning en el presente: supera a otros algoritmos en velocidad y precisión.
- Son esencialmente modelos matemáticos para resolver un problema de optimización.
- Se componen de **neuronas**, su unidad básica de computación.
- La funcionalidad básica es similar a una neurona humana, es decir, toma cierta entrada y activa una salida.
- Una red neuronal organiza sus neuronas en capas, conectadas por las entradas y salidas de sus neuronas.
- El conjunto de todos los parámetros (b y w) para todas las capas, son los que la red busca optimizar para obtener la respuesta adecuada para diferentes entradas.



Conceptos Básicos: Neurona

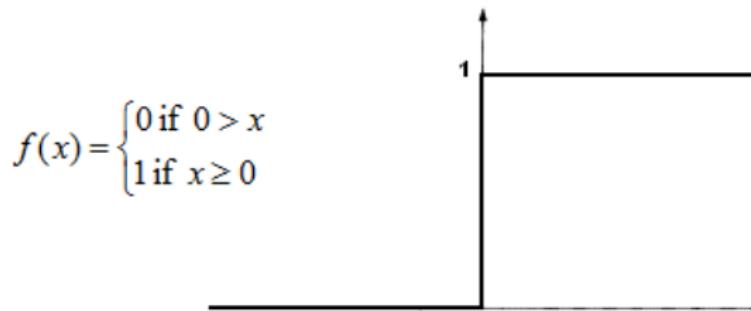
- Una neurona toma como entrada (x),
- realiza alguna computación a ella (e.g.: multiplicarla con una variable w y sumarle otra variable b),
- para producir un valor (e.g.: $z = wx + b$).
- El valor resultante se le pasa a una función no lineal, llamada función de activación (f), para producir la salida final (activación) de la neurona.



Funciones de activación: Step

- Función *step* definida como:

Unit step (threshold)

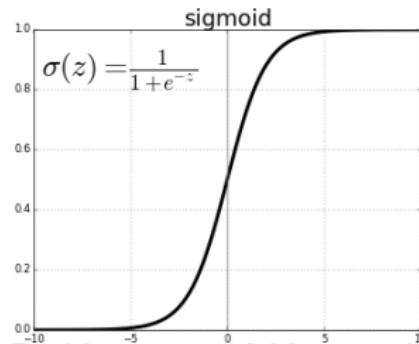


- La salida es 1 si $x \geq 0$, y 0 si $x < 0$.
- Problema: No-diferenciable en cero - en la actualidad, una red neuronal usa el método back-propagation junto con gradient-descent para calcular los pesos de diferentes capas; como esta función no es diferenciable en cero la optimización no es capaz de progresar con gradient-descent y falla en actualizar los pesos.

Funciones de activación: Sigmoidal

- Para superar el problema de *step*, se introdujeron las funciones *sigmoidales*.
- La salida tiende a 0 cuando $z \rightarrow -\infty$, y tiende a 1 cuando $z \rightarrow \infty$.
- Por qué se usa:**

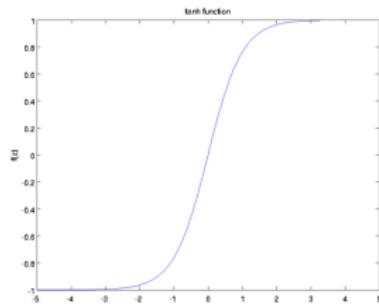
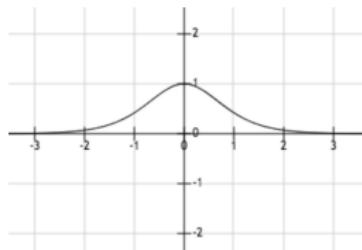
- no lineal*: permite capturar relaciones no lineales en los datos;
- diferenciable*: permite uso de gradient-descent y backpropagation;
- aproxima muy bien la Gaussiana acumulada*: si entrada sigue distribución Gaussiana, se ajusta muy bien, lo que aplica a muchos eventos aleatorios.



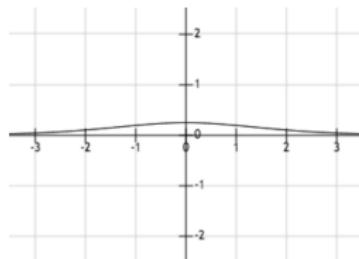
- Problema - vanishing gradients:** aplasta entrada a rango de salida muy pequeño [0,1] y tiene gradientes muy agresivos. Entonces, grandes regiones del espacio de entrada producen un cambio insignificante en la salida. Problema se incrementa con el número de capas y hace que se estanque el aprendizaje en cierto punto.

Funciones de activación: Tanh

- La función $tanh(z)$ es una versión reescalada de la *sigmoidal*, con un rango de salida en $[-1, 1]$, en vez de $[0, 1]$.
- Por qué se usa:** dado que los datos se centran normalmente en torno a 0, las derivativas son mayores. Un gradiente mayor ayuda a un mejor proceso de aprendizaje (precisión).
- Tanh*, para entrada en $[-1, 1]$, se obtiene derivativa en $[0.42, 1]$.



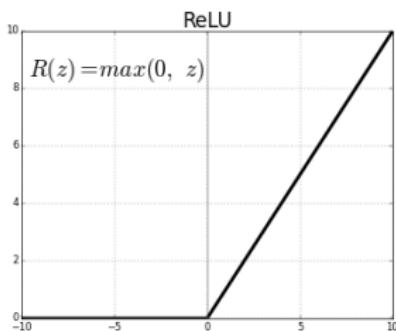
- sigmoidal*, para entrada en $[0,1]$, se obtiene derivativa en $[0.2, 0.25]$.



- Problema de *vanishing gradients* persiste.

Funciones de activación: ReLU

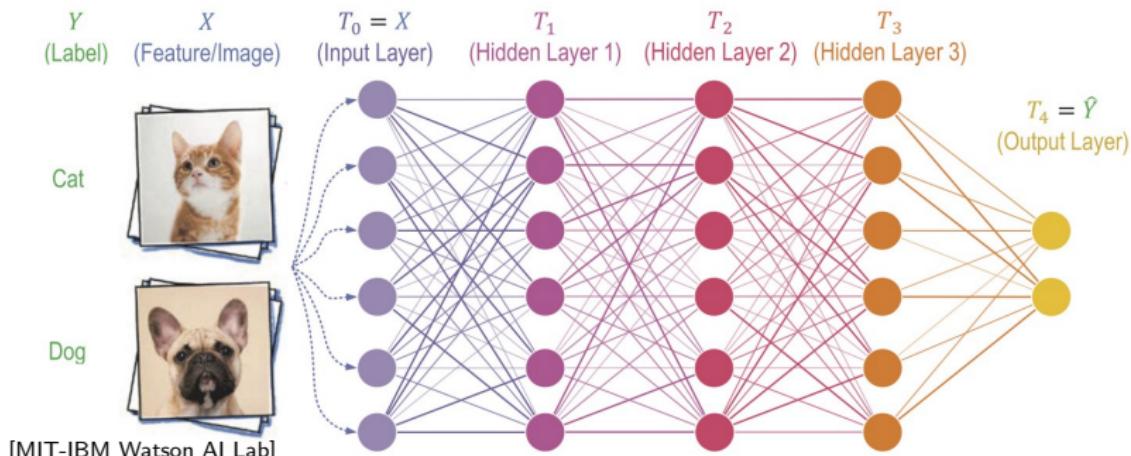
- La *Rectified Linear Unit (ReLU)* es la función de activación más comúnmente utilizada en modelos de deep learning.
- Retorna 0 si recibe entrada negativa. Para cualquier valor positivo de entrada x , retorna el mismo valor. En resumen:
 $f(x) = \max(0, x)$.
- Una variante es *Leaky ReLU*: Igual a ReLU para $x \geq 0$, pero para valores negativos tiene una pendiente constante (< 1). A la pendiente frecuentemente se le llama α . E.g., si el usuario toma $\alpha = 0.3$, la ReLU será $f(x) = \max(0.3 * x, x)$. Ventaja teórica de, al involucrar siempre a entrada x , usaría de forma más completa la información contenida en x .



- Hay otras alternativas, pero usuarios e investigadores no han encontrado suficiente beneficio de usarlas. En la práctica, en general, ReLU ha funcionado mejor que la sigmoidal o Tanh.

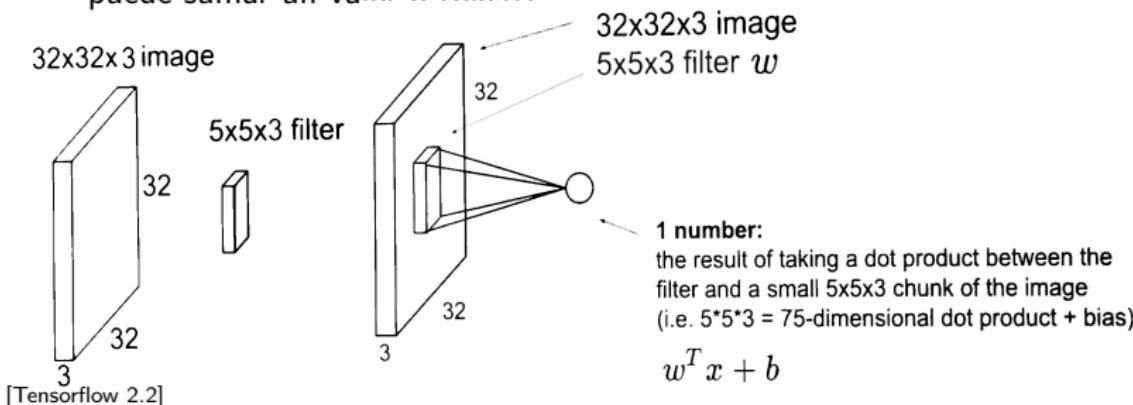
Redes neuronales: Capas (Layers)

- Redes neuronales se organizan en capas (conjunto de neuronas).
- Primera capa recibe datos de entrada, siguientes procesan (capas escondidas) y última es la capa de salida.
- Redes con muchas capas escondidas tienden a ser más precisas, llamadas *deep networks* → algoritmos de machine learning que usan deep networks clasifican dentro de deep learning.



Redes neuronales: Tipos de Capas - Capa Convolucional

- En una capa convolucional, todas las neuronas aplican una operación de convolución a las entradas - neuronas convolucionales.
- La presencia de una de estas capas hace al modelo una Convolutional Neural Network (CNN).
- El parámetro más importante es el tamaño del filtro.
- E.g.: Tamaño del filtro 5*5*3. Imagen de entrada: 32*32 con 3 canales.
- Tomando un trozo de la imagen de 5*5*3 (3 canales) y aplicando convolución con filtro (w), resulta en un solo valor de salida (al cual se le puede sumar un valor b (bias))



Redes neuronales: Tipos de Capas - Capa Convolucional

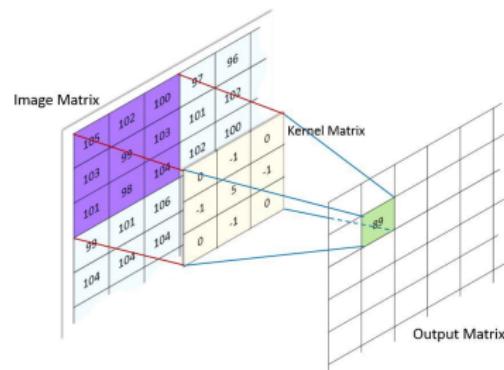
- Es mandatorio que la tercera dimensión del filtro sea igual al número de canales de la entrada.
- Luego, cada capa de entrada debiese convolucionarse completa por los filtros.
- En este caso, se desliza la ventana 1 pixel cada vez. Se puede elegir saltar en más de un pixel (parámetro llamado *stride*).

1	1	1	0	0
0	1	1	0	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

4	3	4
2		

Image Convolved Feature

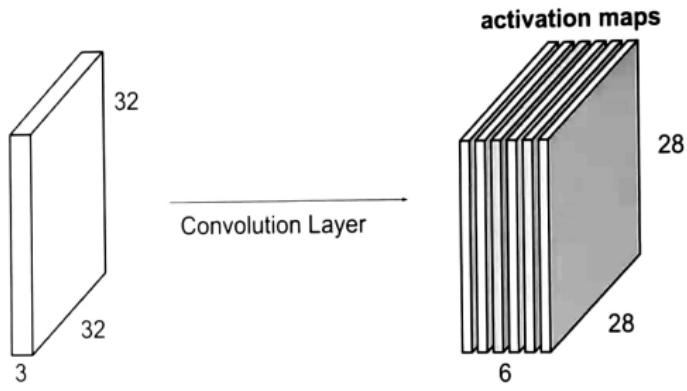
[Tensorflow 2.2, Machine Learning Guru]



- Si se concatenan todas las salidas en 2D, se obtiene un *mapa de activación* de 28×28 , para un stride de 1. *Por qué?*

Redes neuronales: Capa Convolucional - mapas de activación

- Típicamente, se usa más de un filtro en una capa de convolución.
- Si se tienen 6 filtros en el ejemplo anterior, se tendrá una salida de $28*28*6$.

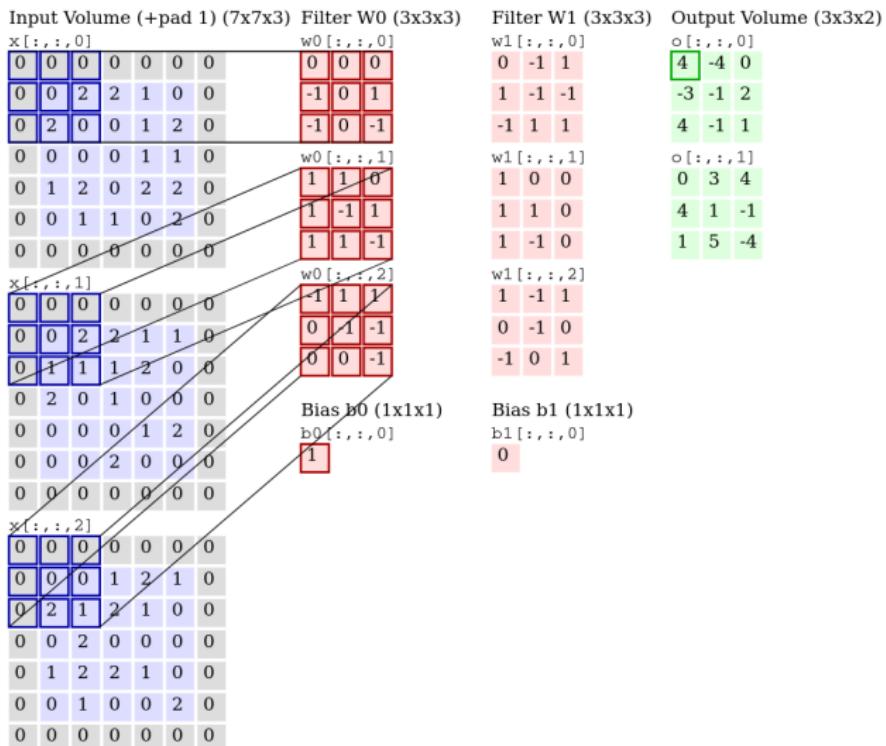


[Tensorflow 2.2]

Redes neuronales: Capa Convolucional - mapas de activación

- La salida se reduce por el tamaño del filtro (dónde cabe para ser aplicado). Práctica estandar es *zero-padding*: agregar ceros a los bordes de la capa de entrada para que la salida quede del mismo tamaño que la entrada.
- Si en el ejemplo anterior se agrega *padding* de tamaño 2, la salida será de $32*32*6$.
- Generalizando, para entrada de $N*N$, filtro de tamaño $F*F$, stride S y padding de tamaño P, la dimension de salida será $(N - F + 2P)/S + 1$
- En el ejemplo (sin padding): $(32 - 5 + 2 * 0)/1 + 1 = 28$
- En el ejemplo (con padding de 2): $(32 - 5 + 2 * 2)/1 + 1 = 32$

Redes neuronales: Capa Convolucional - Ejemplo

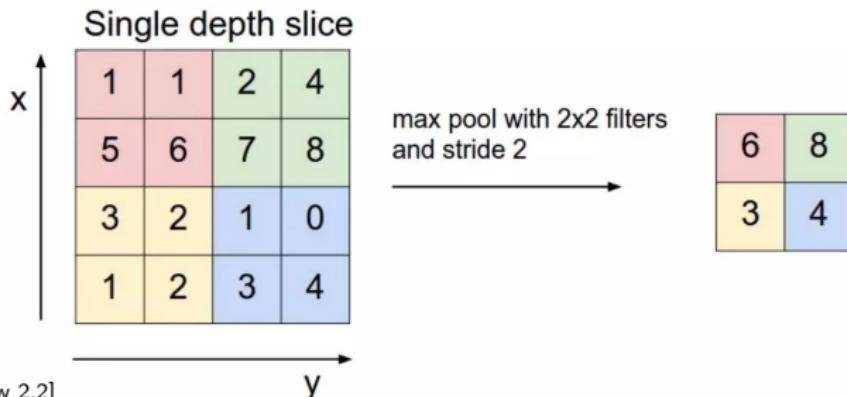


[CS231n Stanford]

Se consideró un stride de 2.

Redes neuronales: Tipos de Capas - Capa de Pooling

- La capa de *pooling* es preferentemente utilizada inmediatamente después de una capa convolucional.
- Se usa para reducir el tamaño (sólo ancho y alto, no profundidad).
- Reduce número de parámetros, overfitting y la computación.
- El más común es *Max pooling* que aplica a una ventana $F \times F$ la operación máximo. El parámetro *stride* se aplica igual que en convolucional.



Redes neuronales: Tipos de Capas - Capa de Pooling

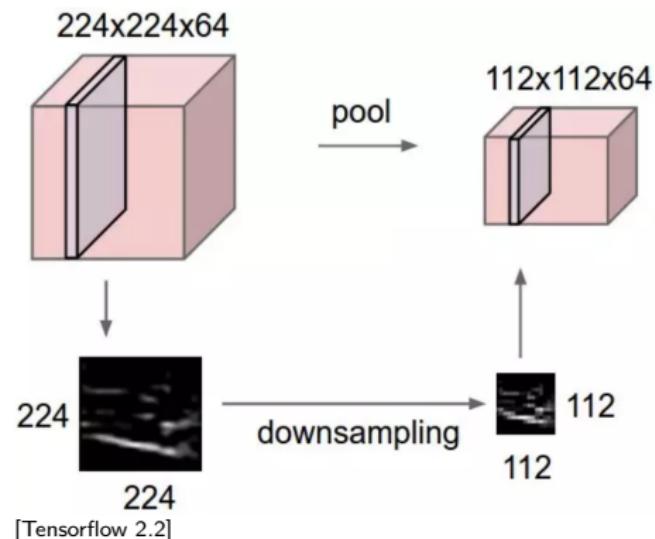
- Si la entrada es de tamaño $w_1 * h_1 * d_1$ y el tamaño del filtro de pooling es $f * f$ con stride S , las dimensiones de la salida $w_2 * h_2 * d_2$ serán:

$$w_2 = (w_1 - f)/S + 1$$

$$h_2 = (h_1 - f)/S + 1$$

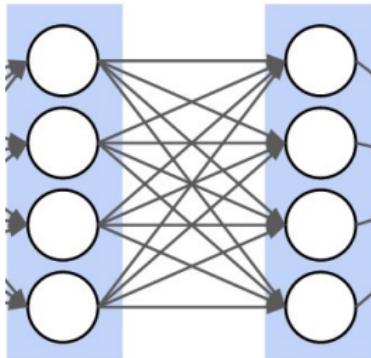
$$d_2 = d_1$$

- El pooling más común se realiza con un filtro de $2*2$ con stride de 2 (reduce el tamaño a la mitad) →
- Si se toma el promedio en vez del máximo, se llama *average pooling*, pero no es muy popular.



Redes neuronales: Tipos de Capas - Capa FC

- En la capa *Fully Connected (FC)*, cada neurona en la capa recibe como entrada todas las neuronas de la capa previa.
- Se le llama también capa *densa* (Dense).
- La salida es la multiplicación de los pesos por entrada, más un bias.
- Las capas de convolución y pooling se utilizan para extraer características genéricas relevantes de las imágenes.
- Finalmente, la capa FC clasifica la información de alto nivel de abstracción obtenida por las capas convolucionales.



[abhishek.mishra Numahub]

Redes neuronales: Proceso de aprendizaje

- Decidir la arquitectura de la red: ordenamiento de las capas, números de neuronas por capa.
- Existen muchas arquitecturas estándar que se pueden reutilizar, útiles para problemas típicos: AlexNet, GoogleNet, Inception Resnet, VGG, etc.
- Ajustar pesos y parámetros: obtener parámetros de la red - pesos (w) y biases (b). Objetivo del entrenamiento es obtener los mejores valores posibles de todos estos parámetros, para resolver el problema de manera confiable.

Redes neuronales: Proceso de aprendizaje

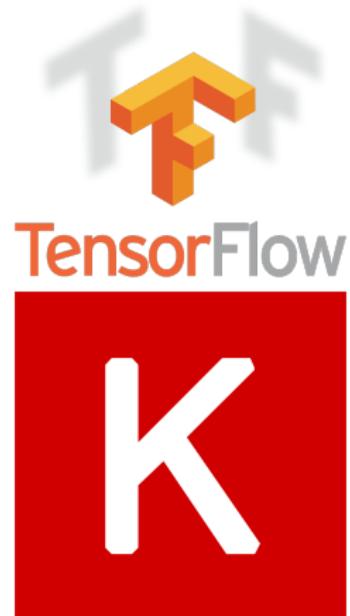
- *Back-propagation*: algoritmo que parte con conjunto aleatorio de parámetros y los actualiza hasta encontrar la salida correcta para todas las imágenes de entrenamiento. Hay muchos métodos *optimizadores* para actualizar los parámetros (más adelante).
- Para optimizar se requiere definir el *costo* que indica si el entrenamiento de la red va en la dirección correcta. La función que evalúa el ajuste de una muestra de entrenamiento a la salida esperada se llama *loss function*, y el costo se calcula normalmente como el promedio de los *loss* entre las muestras de entrenamiento. Se define de tal forma que si el costo se reduce, aumenta la precisión de la red. Hay muchas funciones de loss/costo (más adelante).
- Back-propagation computa eficientemente el gradiente de la *loss function* c/r a cada peso de la red, una capa a la vez, iterando hacia atrás desde la última capa. Luego, se usan iterativamente estos gradientes para actualizar los pesos, para minimizar el costo.

Redes neuronales: Proceso de aprendizaje

- Para mejorar el rendimiento computacional, no todos los datos de entrenamiento alimentan la red de una sola vez. E.g. se tienen 1600 imágenes de entrenamiento, y se dividen en pequeños grupos (*batches*) de tamaño 16 o 32 (*batch-size*). O sea, tomará 100 o 50 iteraciones completar el uso de todos los datos para entrenamiento.
- Se define como un *epoch*, el momento en que una red completa una iteración con todos los datos de entrenamiento. Se realizan varios *epoch* en el proceso de entrenamiento (parámetro).
- Luego, del entrenamiento, los parámetros y el modelo se almacenan en un archivo binario, el cual se podrá usar para obtener la salida para un nuevo ejemplo (*inferencia* o *predicción*).

Manos a la obra: Herramientas

- *TensorFlow*: biblioteca gratuita y open-source para el desarrollo de aplicaciones de machine learning y en particular redes neuronales. Desarrollada por equipo Google Brain inicialmente para uso interno de Google. Liberada en Noviembre 2015.
- *Keras*: biblioteca open-source de redes neuronales escrita en Python. Es capaz de correr sobre TensorFlow, Microsoft Cognitive Toolkit, R, Theano, y PlaidML. Para experimentación rápida con deep learning: amigable, modular, y extensible. Desarrollado inicialmente por un ingeniero de Google, en 2017, el equipo de TensorFlow decidió dar soporte a Keras en la biblioteca core de TensorFlow
- Keras es una abstracción de alto nivel muy intuitiva para desarrollar con facilidad modelos de deep learning.



Manos a la obra: Instalación

- Instalación para TensorFlow (viene con Keras) en:
<https://www.tensorflow.org/install/pip>
- Se recomienda uso de GPU para acelerar entrenamiento. Para testear funcionamiento:

```
import tensorflow as tf
print("Num GPUs Available: ",
      len(tf.config.experimental.list_physical_devices('GPU')))
```

- Salida esperada (algo así):

```
Num GPUs Available: 2
```

- Se recomienda también instalar:

```
pip install numpy
pip install matplotlib
```

Manos a la obra: Keras - Elementos estructurales

- Bibliotecas:

```
import tensorflow as tf
from tensorflow import keras
```

- Crear un modelo:

```
model = tf.keras.Sequential()
```

- En Keras se ensamblan *capas* para formar un modelo.

tf.keras.Sequential es el tipo de modelo más básico, que agrupa una pila lineal de capas.

- Ahora, para agregar capas se usa *add*, método de *tf.keras.Model*:

```
from tensorflow.keras import layers
# Agregar capa densa layer con 64 neuronas al modelo,
# con función de activación ReLU:
model.add(layers.Dense(64, activation='relu'))
# Agregar otra:
model.add(layers.Dense(64, activation='relu'))
# Agregar una capa de salida con 10 neuronas:
model.add(layers.Dense(10))
```

- Existen otros parámetros para capas a estudiar más adelante, como la inicialización y regularización de los pesos (kernel) y biases.

Manos a la obra: Keras - Elementos estructurales

- Habiendo terminado de construir las capas del modelo, éste debe ser compilado. Ejemplos:

- **Regresión:**

```
# Configurar un modelo para regresión con
# error medio cuadrado (MSE).
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
               loss='mse',           # mean squared error
               metrics=['mae'])     # mean absolute error
```

- En este ejemplo, se compila el modelo para un problema de regresión: cuál es el valor de variables de salida y (continuo), para un conjunto de valores de entrada x , conociendo un conjunto de ejemplos (x, y) .

- **Clasificación:**

```
# Configurar un modelo para clasificación categórica.
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.01),
               loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
               metrics=['accuracy'])
```

- En un problema de clasificación, se quiere obtener la etiqueta correcta como respuesta, conociendo pares de entrada y etiquetas (discreto).
- Más adelante: loss, optimizer y metrics.

Manos a la obra: Keras - Elementos estructurales

- Para entrenar, se utiliza el método *fit*:

```
import numpy as np
#1000 filas de 32 valores en [0.0; 1.0]
data = np.random.random((1000, 32))
#1000 filas de 10 valores en [0.0; 1.0]
labels = np.random.random((1000, 10))
model.fit(data, labels, epochs=10, batch_size=32)
```

- data* contiene la entrada, *labels* la salida, para entrenamiento. Se puede definir número de *epochs* y *batch_size*.
- También se puede agregar a *fit*, el conjunto de validación:

```
#Con set de validación
val_data = np.random.random((100, 32))
val_labels = np.random.random((100, 10))
model.fit(data, labels, epochs=10, batch_size=32,
           validation_data=(val_data, val_labels))
```

Manos a la obra: Keras - Elementos estructurales

- En vez de pasar los ejemplos y sus salidas por separado, se puede definir un Dataset, lo cual tiene la ventaja de manejar el acceso a disco de forma adecuada para sets grandes (ejemplo completo más adelante):

```
#Con dataset y batches
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32)
model.fit(dataset, epochs=10)
```

- El método *evaluate* permite obtener el rendimiento de la red para un conjunto de validación:

```
#Con datos:
model.evaluate(data, labels, batch_size=32)

#Con dataset:
model.evaluate(dataset)
```

Manos a la obra: Keras - Elementos estructurales

- Para obtener la predicción (inferencia) de un conjunto de datos, se usa el método *predict*:

```
#Predict
result = model.predict(val_data, batch_size=32)
#Retorna tupla (100,10) - 100 resultados con 10 dimensiones de y:
print(result.shape)
#Recorrer valores:
for i in result:
    i_strings = ["%.4f" % x for x in i]
    salida = ""
    for j in i_strings:
        salida += " " + j
    print(salida)
```

- Definiciones:**

- shape*: Número de elementos de cada dimensión de un tensor.
- rank*: Número de dimensiones del tensor (rango).
- axis o dimension*: Una dimensión en particular de un tensor.
- size*: Número total de items en el tensor.

TensorFlow - Tensores

- En TensorFlow un tensor es un arreglo multi-dimensional con un tipo uniforme (*dtype*). Por defecto es *tf.int32*
- En Python, los tensores son similares a los np.arrays (numpy).
- Son inmutables: no se pueden actualizar, luego de creados.
- Tipos:**

- Tensor *rango-0* o *escalar*: contiene un solo valor.

```
rank_0_tensor = tf.constant(4)  
print(rank_0_tensor)
```

```
tf.Tensor(4, shape=(), dtype=int32)
```

- Tensor *rango-1* o *vector*: es una lista de valores (unidimensional):

```
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])  
print(rank_1_tensor)
```

```
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```

TensorFlow - Tensores

- **Tipos:**

- Tensor *rango-2* o *matriz* - bidimensional:

```
rank_2_tensor = tf.constant([[1, 2],  
                            [3, 4],  
                            [5, 6]], dtype=tf.float16)  
  
print(rank_2_tensor)
```

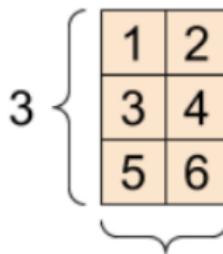
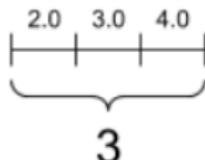
```
tf.Tensor(  
[[1. 2.]  
[3. 4.]  
[5. 6.]], shape=(3, 2), dtype=float16)
```

scalar, shape: []

vector, shape: [3]

matrix, shape: [3, 2]

4



Tensores - Operaciones

- **Operaciones con tensores:**

```
a = tf.constant([[1, 2],  
                 [3, 4]])  
b = tf.constant([[1, 1],  
                 [1, 1]]) # Lo mismo con 'tf.ones([2, 2])'  
print(tf.add(a, b), "\n")  
print(tf.multiply(a, b), "\n")  
print(tf.matmul(a, b), "\n")  
  
tf.Tensor(  
[[2 3]  
 [4 5]], shape=(2, 2), dtype=int32)  
tf.Tensor(  
[[1 2]  
 [3 4]], shape=(2, 2), dtype=int32)  
tf.Tensor(  
[[3 3]  
 [7 7]], shape=(2, 2), dtype=int32)
```

- **Equivalentes:**

```
print(a + b) # suma por elemento  
print(a * b) # multiplicacion por elemento  
print(a @ b) # multiplicacion matricial
```

Tensores - Operaciones

- Conversión a array the numpy:

```
np.array(rank_2_tensor)  
rank_2_tensor.numpy()
```

- Funciones útiles:

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])  
# Encontrar el valor mayor:  
print(tf.reduce_max(c))  
# Encontrar el indice del valor mayor  
print(tf.argmax(c))  
#Calcular softmax  
print(tf.nn.softmax(c))  
  
tf.Tensor(10.0, shape=(), dtype=float32)  
tf.Tensor([1 0], shape=(2,), dtype=int64)  
tf.Tensor([  
[[2.6894143e-01 7.3105860e-01]  
[9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

Tensores - Softmax

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])
print(tf.nn.softmax(c))

tf.Tensor
[[2.6894143e-01 7.3105860e-01]
 [9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

- En deep learning, el término *logits layer* se usa para la última capa de neuronas para una tarea de clasificación, la cual produce valores de predicción en bruto como números reales en $[-\infty, +\infty]$.
- La función *softmax* es una función de activación que transforma números (logits) en probabilidades que suman 1. Entrega un vector que representa las distribuciones de probabilidad de una lista de salidas, considerando los logits como el logaritmo de las probabilidades (ajusta negativos).

LOGITS SCORES	SOFTMAX	PROBABILITIES
$y \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$	$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$	$\begin{array}{l} \rightarrow p = 0.7 \\ \rightarrow p = 0.2 \\ \rightarrow p = 0.1 \end{array}$

Tensores - Operaciones

- Acceso a elementos (sigue reglas similares a arreglos en Python):
 - índices comienzan en 0.
 - índices negativos cuentan hacia atrás desde el final.
 - funciona estructura start:stop:step
 - Para tensor multidimensional, coma separa índices.

```
rank_1_tensor = tf.constant([5, 3, 1, 2, 4, 6, 8, 13, 21, 34])
print("Primero:", rank_1_tensor[0].numpy())
print("Segundo:", rank_1_tensor[1].numpy())
print("Ultimo:", rank_1_tensor[-1].numpy())
```

Primero: 5
Segundo: 3
Ultimo: 34

```
print(rank_2_tensor.numpy())
print("Valor:", rank_2_tensor[1, 1].numpy())
```

[[1. 2.]
 [3. 4.]
 [5. 6.]]
Valor: 4.0

Tensores - Operaciones

- Acceso a elementos:

```
print("Todo:", rank_1_tensor[:].numpy())
print("Antes de 4:", rank_1_tensor[:4].numpy())
print("Desde 4 al final:", rank_1_tensor[4:].numpy())
print("Desde 2, antes de 7:", rank_1_tensor[2:7].numpy())
print("Intercalados:", rank_1_tensor[::-2].numpy())
print("Reversados:", rank_1_tensor[::-1].numpy())
```

Todo: [0 1 1 2 3 5 8 13 21 34]

Antes de 4: [0 1 1 2]

Desde 4 al final: [3 5 8 13 21 34]

Desde 2, antes de 7: [1 2 3 5 8]

Intercalados: [0 1 3 8 21]

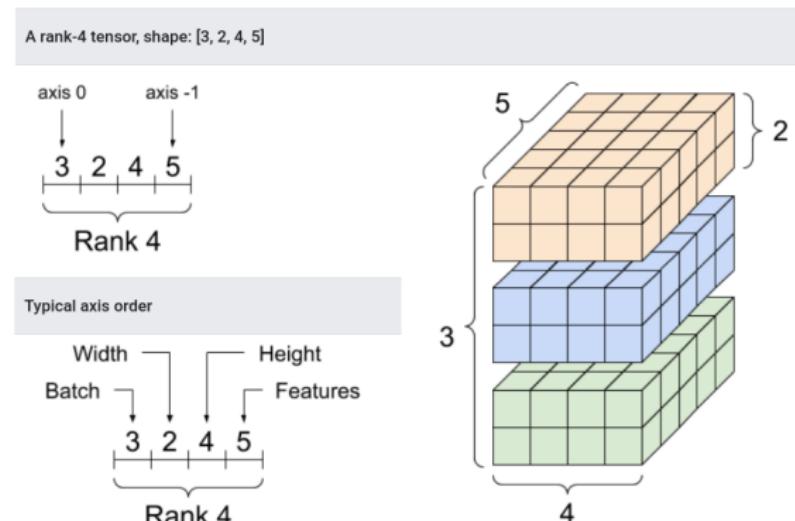
Reversados: [34 21 13 8 5 3 2 1 1 0]

- Más en:

<https://www.tensorflow.org/guide/tensor>

Tensores en Deep Learning

- Si bien las dimensiones son referidas por sus índices, se debe considerar el orden típico considerado en deep learning.
- En general, se ordenan de global a local: El eje de *batch* primero, luego las dimensiones espaciales, y al final las características (features) - o canales en el caso de una imagen.



Funciones Loss

- Uno de los parámetros relevantes para definir un modelo es establecer la función de costo o *loss*.
- Las redes neuronales en deep learning se entranan con el método de optimización stochastic gradient descent, el cual requiere el error de estimación del modelo en cada iteración.
- Requiere la elección de una función de error: la función *loss*.
- La función *loss* se usa para estimar la pérdida del modelo, tal que los pesos del modelo se actualicen para ir reduciendo esta pérdida en la siguiente evaluación.

Funciones Loss

- Existen funciones loss diferentes para distintos problemas.
E.g. clasificación, regresión.
- Además, la configuración de la capa de salida debe ser apropiada para la función *loss* escogida.
- Combinaciones más importantes *loss*-problema:
 - Función *mean squared error* (y variantes) para problemas de **regresión**.
 - Funciones *cross-entropy* y *hinge* para **clasificación binaria**.
 - Funciones *cross-entropy* y *KL divergence* para **clasificación multi-clase**.

Funciones Loss - problemas de regresión

- Regresión: predecir una cantidad en los reales.
- Se usará para ejemplos la biblioteca scikit-learn de Python (análisis predictivo de datos, construida sobre NumPy, SciPy y matplotlib; licencia BSD) - **sklearn**.
- Para regresión, la función *make_regression* de *sklearn.datasets*:

```
from sklearn.datasets import make_regression
X, y = make_regression(n_samples=1000, n_features=20,
                       n_informative=10, noise=0.1, random_state=1)
```

- Ejemplo: problema con 20 características de entrada (10 significativos y 10 irrelevantes); 1,000 ejemplos aleatorios. Se usa *random_state* para hacer experimento aleatorio repetible.
- Las redes en general tienen mejor rendimiento cuando las variables de entrada y salida de valores reales son escaladas a un rango sensible. En imágenes no es tan necesario pues los canales están estandarizados.
- *sklearn* usa *StandardScaler*: resta la media y normaliza por desviación estándar, para cada variable.

```
from sklearn.preprocessing import StandardScaler
X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y), 1))[:, 0]
```

Funciones Loss - problemas de regresión

- Normalmente, se usa sólo conjunto de entrenamiento para parámetros de escala.
Luego, se divide conjunto en entrenamiento y testing:

```
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

- Red de ejemplo para explorar diferentes funciones loss: 20 entradas, una capa escondida (25 nodos) con ReLU, salida con 1 nodo (valor real a predecir (con activación lineal)):

```
import tensorflow as tf
from tensorflow.keras import layers
model = tf.keras.Sequential()
model.add(layers.Dense(25, input_dim=20, activation='relu',
                      kernel_initializer='he_uniform'))
model.add(layers.Dense(1, activation='linear'))
```

- De optimizador usar stochastic gradient descent (SGD) con *learning rate* de 0.01 y momentum de 0.9 (explicación más adelante).

```
from tensorflow.keras.optimizers import SGD
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='LA FUNCION LOSS', optimizer=opt)
```

- Entrenamiento (fit) con 100 epochs y conjunto de testing evaluado al final de cada epoch:

```
history = model.fit(trainX, trainy, validation_data=(testX, testy),
                     epochs=100, verbose=0)
```

- A continuación, diferentes loss para regresión.

Regresión - Función Loss Mean Squared Error

- Mean Squared Error (MSE), loss por defecto en problemas de regresión.
- Matemáticamente, se prefiere si la variable objetivo es Gaussiana (en muchos problemas es así; cambiar sólo si hay una buena razón).

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

con Y_i valor observado, y \hat{Y}_i valor de predicción.

- Es el promedio de las diferencias al cuadrado, entre valores observados y predicciones. Siempre positivo y valor perfecto es 0.0. El cuadrado castiga errores más grandes.
- En Keras se usa especificando '`mse`' o '`mean_squared_error`' como función `loss` al compilar el modelo:

```
model.compile(loss='mean_squared_error')
```

- Se recomienda usar función de activación lineal en la salida (salida real continua):

```
model.add(layers.Dense(1, activation='linear'))
```

- Ejemplo en `regression1.py`

Regresión - Función Loss Mean Squared Logaritmic Error

- Para problemas donde valor objetivo tiene valores dispersos y no se desea castigar tan fuertemente los errores como en *mse*.
- Mean Squared Logaritmic Error (MSLE) se enfoca en diferencia porcentual: MSLE tratará diferencias pequeñas entre pequeños valores observados y predicciones, aproximadamente similar a grandes diferencias entre valores grandes.

$$MSLE = \frac{1}{N} \sum_{i=0}^N (\log(Y_i + 1) - \log(\hat{Y}_i + 1))^2$$

● Ejemplo:

	observado	predicción	MSE	MSLE
	30	20	100	0.02861
	30000	20000	100000000	0.03100
			<i>gran diferencia</i>	<i>pequeña diferencia</i>

- En Keras se usa especificando '*mean_squared_logarithmic_error*' y dejando todo el resto igual:

```
model.compile(loss='mean_squared_logarithmic_error',
              optimizer=opt, metrics=['mse'])
```

- Ejemplo en *regression2.py*

Regresión - Función Loss Mean Absolute Error

- Mean Absolute Error (MAE) no es sensible a outliers. Desventaja: magnitud del gradiente no depende del tamaño del error (gradiente grande con error pequeño), lo que puede generar problemas de convergencia.

$$MSE = \frac{1}{n} \sum_{i=1}^n |Y_i - \hat{Y}_i|$$

- Es el promedio de las diferencias absolutas, entre valores observados y predicciones.
- Útil para control de outliers o si se sabe que la distribución es multimodal (predicciones en las modas y no en su promedio).
- En Keras se usa especificando '*mean_absolute_error*' como función loss al compilar el modelo:

```
model.compile(loss='mean_absolute_error',  
              optimizer=opt, metrics=['mse'])
```

- Ejemplo en *regression3.py*

Funciones Loss - Clasificación Binaria

- Clasificación binaria: ejemplos con una de dos etiquetas de salida - predecir un 0 o 1, para una o otra clase; a menudo implementado como predecir probabilidad de que ejemplo pertenezca a clase 1.
- Ejemplo: problema de círculos de scikit-learn: muestras para dos círculos concéntricos en un plano 2D - puntos de círculo externo clase 0, puntos de círculo interno clase 1.
- 1000 ejemplos con 10% de ruido estadístico.

```
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
```

- Separación de muestras igual a ejemplos anteriores.
- Se usa el mismo modelo, salvo por activación de capa de salida. La función de activación de ella dependerá de la función loss.

```
...
model.add(Dense(1, activation='...'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='...', optimizer=opt, metrics=['accuracy'])
history = model.fit(trainX, trainy,
                     validation_data=(testX, testy),
                     epochs=200, verbose=0)
```

- Se agrega métrica *accuracy* (predicciones correctas VS total).

Clasificación - Función Binary Cross Entropy Loss

- Binary Cross-entropy es la función loss por defecto a usar en clasificación binaria. Los valores de salida son 0 o 1.
- Calcula promedio entre distribuciones de probabilidad observada y de predicción - Cross-entropy perfecta con valor 0.

$$BCE = -\frac{1}{n} \sum_{i=1}^n Y_i \log(p(\hat{Y}_i)) + (1 - Y_i) \log(1 - p(\hat{Y}_i))$$

- Y_i es valor observado (0 o 1), y $p(\hat{Y}_i)$ es la probabilidad del valor de predicción (entregado por sigmoidal).
- En Keras se usa especificando '*binary_crossentropy*' como función loss al compilar el modelo:

```
model.compile(loss='binary_crossentropy', optimizer=opt,
               metrics=['accuracy'])
```

- La función requiere que la capa de salida tenga un solo nodo y activación '*sigmoid*' para predecir la probabilidad de la clase 1.

```
model.add(Dense(1, activation='sigmoid'))
```

- Ejemplo en *binary1.py*

Funciones Loss - Clasificación Multi-clase

- Clasificación multi-clase: a ejemplo se le asigna una de dos o más clases. Predecir un entero (clase), entre 0 y (*num_clases* - 1). A menudo implementado como predecir probabilidad de que el ejemplo pertenezca a cada clase conocida.

- Ejemplo: problema de blobs *make_blobs()* de scikit-learn - 1,000 ejemplos de problema de clasificación de 3 clases con 2 variables de entrada.

```
X, y = make_blobs(n_samples=1000, centers=3, n_features=2,  
                   cluster_std=2, random_state=2)
```

- Separación de muestras igual a ejemplos anteriores.

- Se usa el mismo modelo, salvo por el número de nodos de salida (igual a número de clases) y activación de capa de salida. La función de activación de ella dependerá de la función loss.

```
...  
model.add(Dense(..., activation='...'))  
opt = SGD(lr=0.01, momentum=0.9)  
model.compile(loss='...', optimizer=opt, metrics=['accuracy'])  
history = model.fit(trainX, trainy,  
                     validation_data=(testX, testy),  
                     epochs=100, verbose=0)
```

- 100 epochs. Se agrega métrica *accuracy* (predicciones correctas VS total).

Clasificación - Función Categorical Cross Entropy Loss

- Categorical Cross-entropy es la función loss por defecto a usar en clasificación binaria. Los valores objetivo son $\{0, 1, \dots, n - 1\}$, donde cada clase asume un entero diferente.
- Calcula el promedio entre las distribuciones de probabilidad observada y de predicción para todas las clases del problema. Puntaje perfecto es 0.

$$CCE = \sum_{i=1}^n -Y_i \log\left(\frac{e^{\hat{Y}_i}}{\sum_{j=1}^n e^{\hat{Y}_j}}\right)$$

- Y_i es valor de probabilidad observado para la clase i (normalmente una clase tiene 1, y el resto 0) e \hat{Y}_i es el valor de predicción en formato *logit* (luego ajustado por softmax).
- En Keras se usa especificando '*categorical_crossentropy*' como función loss al compilar el modelo:

```
model.compile(loss='categorical_crossentropy', optimizer=opt,
               metrics=['accuracy'])
```
- La función requiere que la capa de salida tenga n nodos (para n clases) y activación '*softmax*' para predecir la probabilidad de cada clase.

```
model.add(Dense(3, activation='softmax'))
```

Categorical Cross Entropy Loss - One-hot Encoding

- Se requiere que la salida de los ejemplos se encuentre en formato *one-hot encoding*.
- Para un ejemplo de clase i , el formato sería $[0, 0, \dots, Y_i = 1, \dots, 0, 0]$, o sea, una lista con sólo 0s, salvo en la posición i , que contiene un 1.
- Representa las probabilidades de los ejemplos.
- Ejemplo:

```
y = [2, 2, 0, 1, 0, 1]
# one-hot encoding:
from tensorflow.keras.utils import to_categorical
y = to_categorical(y)
print(y)
[ [0, 0, 1], [0, 0, 1], [1, 0, 0],
  [0, 1, 0], [1, 0, 0], [0, 1, 0] ]
```

- Ejemplo en *multiclass1.py*

Categorical Cross Entropy Loss - SoftMax

- La activación 'softmax' entrega valores de probabilidad, desde valores que pueden ser negativos y positivos.

$$L_i = \frac{e^{\hat{Y}_i}}{\sum_{j=1} e^{\hat{Y}_i}}$$

- Luego, la cross-entropy loss suma log negativos.
- Ejemplo:



	Scoring Function	Unnormalized Probabilities	Normalized Probabilities	Negative Log Loss
Dog	-3.44	0.0321	0.0006	
Cat	1.16	3.1899	0.0596	
Boat	-0.81	0.4449	0.0083	
Airplane	3.91	49.8990	0.9315	0.0709

- En este caso, el clasificador reportaría correctamente que la imagen es un avión con un 93.15% de confianza.

Clasificación - Sparse Categorical Cross Entropy Loss

- Para problemas de clasificación con un gran número de clases.
- One-hot encoding poco práctico/eficiente en este caso.
- Sparse Cross-Entropy realiza el cálculo de cross-entropy, sin requerir codificación one-hot en el entrenamiento.
- En Keras se usa especificando '*sparse_categorical_crossentropy*' como función loss al compilar el modelo:

```
model.compile(loss='sparse_categorical_crossentropy',  
              optimizer=opt, metrics=['accuracy'])
```

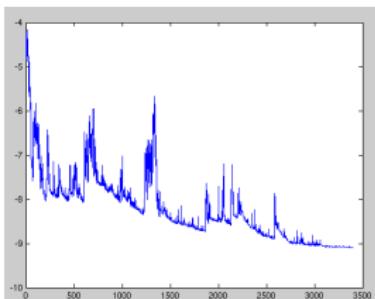
- La entrada y es entonces sólo un vector con los números de la clase correspondiente (no requiere one-hot).

Keras - Optimizers

- Entrenar es equivalente a minimizar la función loss.
- Gradient descent es la forma más común de optimizar redes neuronales.
- Minimiza una función loss $L(\theta)$ parametrizada por los parámetros de un modelo $\theta \in \mathbb{R}^d$ actualizando los parámetros en la dirección opuesta del gradiente de la función loss $\nabla_{\theta} L(\theta)$.
- El learning rate η determina el tamaño de los pasos para alcanzar el mínimo.
- O sea, se sigue la dirección de la pendiente de la superficie que representa a la función objetivo bajando hasta llegar al valle.

Keras - Optimizers - SGD

- El algoritmo más simple es Stochastic Gradient Descent (SGD): realiza actualización de parámetros para cada ejemplo de entrenamiento $x^{(i)}$ y etiqueta $y^{(i)}$.
$$\theta = \theta - \eta \cdot \nabla_{\theta} L(\theta; x^{(i)}; y^{(i)})$$
- SGD realiza una actualización a la vez: es rápido y se puede usar para aprender online; alta frecuencia de actualización genera alta varianza que causa que función loss fluctúe mucho.



- Permite saltar a nuevos y potencialmente mejores mínimos locales, pero complica convergencia al mínimo exacto (oscila mucho).
- Se ha demostrado que si se reduce lentamente η , SGD converge cada vez más suavemente, convergiendo al mínimo.

Keras - Optimizers - Mini-batch Gradient Descent

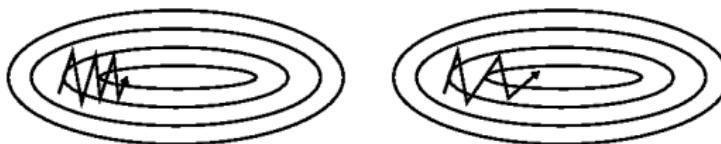
- Realiza actualización por cada mini-batch de n ejemplos de entrenamiento:

$$\theta = \theta - \eta \cdot \nabla_{\theta} L(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

- De esta forma:
 - Reduce la varianza de las actualizaciones (convergencia más estable).
 - Puede usar operaciones de matrices altamente optimizadas (muy eficiente).
- Se usa el mismo término SGD en este caso.

Gradient Descent Optimization - Momentum

- SGD tiene problema de navegar barrancos: áreas donde las curvas de la superficie son mucho abruptas en una dimensión que en la otra (común cerca de un óptimo local) - SGD oscila en las pendientes del barranco con bajo progreso en descenso hacia el óptimo local.



izquierda: sin momentum; derecha: con momentum.

<https://ruder.io/optimizing-gradient-descent/>

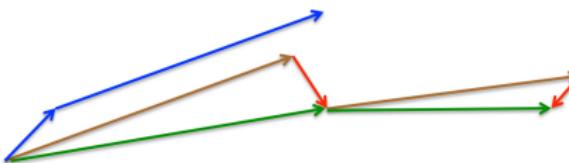
- Momentum [Qian, 1999] es un método que ayuda a acelerar SGD en la dirección relevante, reduciendo las oscilaciones: agrega una fracción γ (usualmente 0.9 o similar) del vector de actualización del paso temporal pasado al vector de actualización actual:
 $v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\theta} L(\theta)$
 $\theta = \theta - v_t$
- Esencialmente con momentum, se empuja la bola por la colina; la bola acumula momentum mientras rueda cerro abajo, cada vez más rápido - parámetros: momentum incrementa dimensiones cuyos gradientes apuntan en la misma dirección y reduce la actualización a gradientes de dirección cambiante (converge más rápido y reduce oscilación).

Gradiente acelerado de Nesterov

- Tampoco es deseable que la bola siga ciegamente la pendiente (bola inteligente: se frene cuando la colina se vuelva cuesta arriba, para que no se pase).
- Nesterov accelerated gradient (NAG) [Nesterov, 1983] hace que el momentum considere la idea anterior: dado que se usará el término de momentum γv_{t-1} para mover θ , se computa $\theta - \gamma v_{t-1}$ que da aproximación de la siguiente posición de los parámetros (idea general de donde estarán los parámetros).
- Se calcula entonces el gradiente con respecto a la posición aproximada futura de los parámetros (no en relación a los parámetros actuales):

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\theta} L(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$



- Momentum primero computa gradiente actual (vector azul pequeño) y luego hace gran salto en la dirección del gradiente actualizado acumulado (vector azul grande).
- NAG primero hace gran salto en la dirección del gradiente previamente acumulado (vector café), mide el gradiente y luego hace una corrección (vector rojo): actualización de NAG completa (vector verde).
- Esta actualización anticipatoria evita ir demasiado rápido y resulta en mayor responsividad.

Optimizers - Adagrad

- Adagrad [Duchi et al., 2011] es un algoritmo que adapta el learning rate a los parámetros, realizando actualizaciones más pequeñas para parámetros asociados a características que ocurren frecuentemente y más grandes para parámetros asociados con características infrecuentes: se ajusta bien para datos sparse (datasets grandes). Usa learning rates diferentes para cada parámetro θ_i en cada paso de tiempo t :

$$g_{t,i} = \nabla_{\theta_i} L(\theta_{t,i})$$

con $g_{t,i}$ derivativa parcial de la función loss relativa a parámetro θ_i al tiempo t .

- Entonces, la actualización SGD para cada parámetro θ_i al tiempo t es:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i,i} + \epsilon}} \cdot g_{t,i}$$

- Adagrad modifica el learning rate general η en cada paso t para cada θ_i , basado en sus gradientes pasados, con $G_t \in \mathbb{R}^{d \times d}$ matriz diagonal con cada elemento diagonal i, i la suma de los cuadrados de los gradientes de θ_i hasta t ; ϵ es un término de suavizado para evitar división por cero (usualmente $\sim 1e-8$).
 - Como G_t contiene los gradientes pasados de todos los θ en su diagonal, se puede vectorizar realizando el producto matriz-vector \odot entre G_t y g_t :
- $$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t, \text{ con } \eta \text{ usualmente } 0.01.$$
- Beneficio:** elimina necesidad de sintonización manual de η .
 - Debilidad principal:** acumulación en el denominador, hace que learning rate se encoja hasta ser infinitesimalmente pequeño, momento en el que se deja de aprender.

Optimizers - Adadelta

- Adadelta [Zeiler, 2012] - extensión de Adagrad para reducir agresividad en decrecimiento del learning rate: en vez de acumular todos los gradientes cuadrados pasados, restringirse a una ventana de gradientes pasados de tamaño fijo w .
- Por eficiencia, la suma de gradientes se define recursivamente como promedio atenuado de los gradientes cuadrados pasados (running average). El promedio $E[g^2]_t$ en tiempo t entonces depende sólo de promedio previo (como fracción γ , similar a momentum; seteada usualmente en 0.9) y el gradiente actual:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

- La actualización toma la forma:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t = -\frac{\eta}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \nabla\theta_t$$

dado que el denominador corresponde al criterio de error raíz de la media cuadrada (RMS) de un gradiente.

Optimizers - Adadelta

- Sin embargo, los autores notaron que las unidades en la actualización no se corresponden (η), vale decir, el paso de actualización debiese estar en unidades relacionadas con el parámetro que actualiza. Para esto, primero definieron un nuevo promedio, ahora de las actualizaciones de parámetros al cuadrado:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta_t^2$$

- El RMS de la actualización de parámetros es entonces:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

- Dado que $RMS[\Delta\theta]_t$ es desconocido, se aproxima por $RMS[\Delta\theta]_{t-1}$ que reemplaza al learning rate η :

$$\Delta\theta_t = -\frac{RMS[g]_{t-1}}{RMS[g]_t} g_t$$

- **Beneficio:** ni siquiera se requiere un learning rate por defecto.

Optimizers - RMSProp

- RMSprop y Adadelta se desarrollaron independientemente, más o menos al mismo tiempo, para resolver el problema de disminución de learning rates de Adagrad. De hecho, RMSprop es idéntico en el primer vector de actualización de Adadelta presentado previamente:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- RMSprop también divide el learning rate por el promedio atenuado de gradientes cuadrados.
- Su autor, Geoff Hinton, sugiere $\gamma = 0.9$ y $\eta = 0.001$.

Optimizers - Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015], además de almacenar promedios atenuados de gradientes cuadrados como Adadelta y RMSprop, también calcula un promedio de gradientes pasados m_t , similar al momentum.
- Sin embargo, mientras momentum puede verse como una bola rodando en una pendiente, en Adam el comportamiento es como una bola pesada con fricción.
- Se calculan:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t ; \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

con m_t el primer momento (media) y v_t el segundo momento (varianza no centrada) de los gradientes.

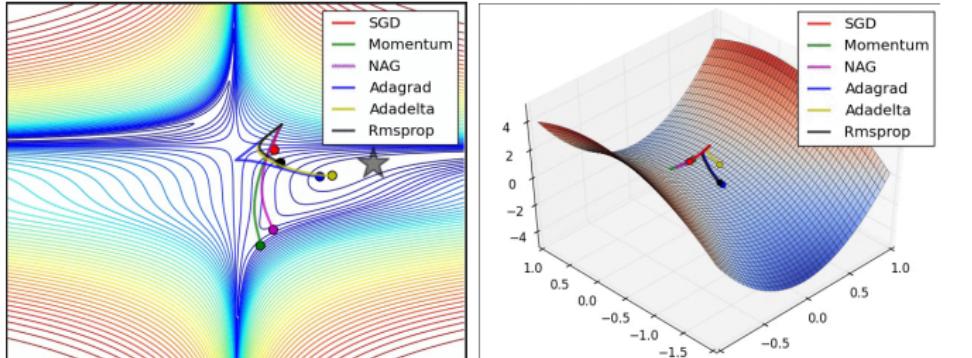
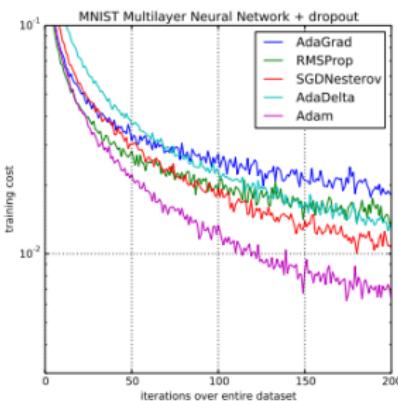
- Dado que m_t y v_t se inicializan con vectores de ceros, los autores observaron que estaban sesgados hacia el cero, sobre todo en los pasos iniciales y especialmente cuando las tasas de decaimiento son pequeñas (o sea, β_1 y β_2 cercanos a 1). Contrarrestaron este efecto calculando estimados de primer y segundo momento con corrección de sesgo:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} ; \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Los usan para actualizar los parámetros igual a Adadelta y RMSprop:
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$
- Valores por defecto $\beta_1 = 0.9$, $\beta_2 = 0.999$ y $\epsilon = 10e - 8$.
- Los autores demostraron empíricamente que Adam funciona bien en la práctica y es competitivo en relación a otros algoritmos.
- Bajo uso de memoria (un poco más que SGD+momentum).

Optimizers - Comparaciones

- Existen otros algoritmos, pero los últimos presentados son aún los más utilizados en el estado del arte.
- Para MNIST ver imagen.
- Ver optimizer.gif y optimizer2.gif para una comparación gráfica de los comportamientos estudiados.



Optimizers - Uso en Keras

- SGD + NAG:

```
import tensorflow as tf
opt = tf.keras.optimizers.SGD(lr=0.01, momentum=0.9,
                               nesterov=True)
```

- Adagrad:

```
opt = tf.keras.optimizers.Adagrad(lr=0.01, epsilon=1e-07)
```

- Adadelta (*rho* equivale a decaimiento γ):

```
opt = tf.keras.optimizers.Adadelta(lr=0.001, rho=0.95,
                                    epsilon=1e-07)
```

- RMSprop (*rho* equivale a decaimiento γ):

```
opt = tf.keras.optimizers.RMSProp(lr=0.001, rho=0.9,
                                   epsilon=1e-07)
```

- Adam:

```
opt = tf.keras.optimizers.Adam(lr=0.001, beta_1=0.9,
                               beta_2=0.999, epsilon=1e-07)
```

Optimizers - Learning Rate Schedule

- Manera definida por el usuario para controlar el *learning rate*. Keras lo soporta vía *callbacks*.
- Se recomienda usar SGD cuando se usa un *callback* de *learning rate schedule*.
- Keras dispone de *ReduceLROnPlateau* para reducir el *learning rate* cuando se detecta un "valle" en el rendimiento del modelo (no hay cambio luego de cierto número de epochs).
- *ReduceLROnPlateau* requiere especificar la métrica a monitorear durante el entrenamiento (*monitor*), el valor multiplicador del *learning rate* (*factor*), y *patience* para especificar número de *epochs* a esperar antes de activar el cambio en el *learning rate*.
- Ejemplo: monitorear el *validation loss* y reducir el *learning rate* por un orden de magnitud si el *validation loss* no mejora en 100 *epochs*:

```
from keras.callbacks import ReduceLROnPlateau
...
rlrop = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
                           patience=100)
model.fit(..., callbacks=[rlrop])
```

Optimizers - Learning Rate Scheduler

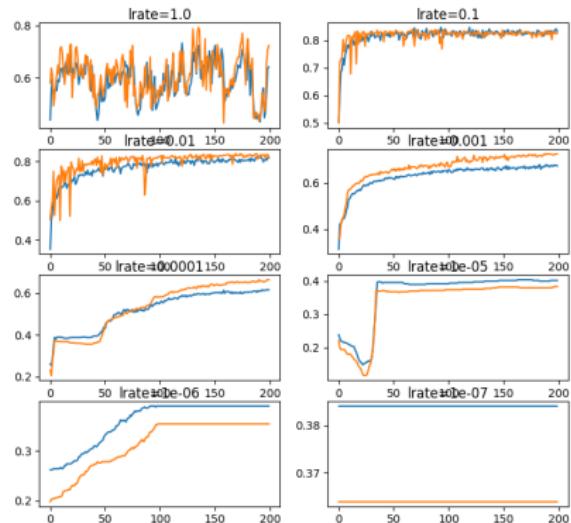
- Keras también permite usar el *callback* **LearningRateScheduler** para especificar una función llamada en cada *epoch* para ajustar el *learning rate*.
- Se puede definir una función de Python que tome dos argumentos (*epoch* y *learning_rate* actual) y retorne el nuevo *learning rate*.

```
from keras.callbacks import LearningRateScheduler
...
def lr_decay(epoch, learning_rate):
    return learning_rate * math.pow(0.6, epoch)
lrs = LearningRateScheduler(lr_decay)
model.fit(..., callbacks=[lrs])
```

- Como previamente visto, existen Keras diversos optimizadores que ya implementan *learning rates* adaptivos: RMSprop, Adadelta, Adagrad, Adam.

Optimizers - Dinámica de los Learning Rate

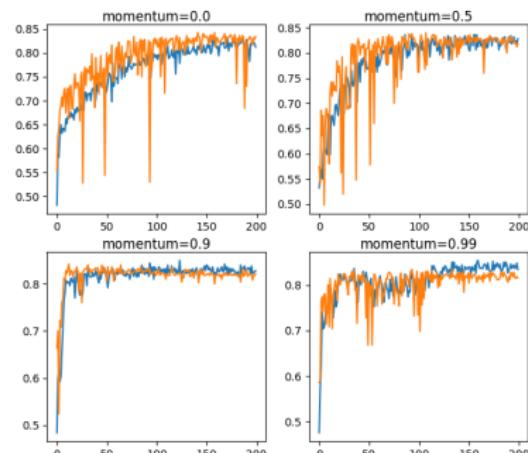
- Ejemplo: *lr1.py*
- Ejemplo considerando múltiples *learning rate*. Los gráficos muestran oscilaciones en el comportamiento para *learning rates* muy grandes (1.0) y la inhabilidad del modelo para aprender con *learning rates* muy pequeños ($1e-6, 1e-7$).



- Aprende bien con *learning rates* de $1e-1, 1e-2$ de $1e-3$, cada vez más lento cuando el *learning rate* se disminuye. Para la configuración del modelo, un *learning rate* moderado de 0.1 resulta en un buen rendimiento del modelo en los conjuntos de entrenamiento y testing.

Optimizers - Dinámica del Momentum

- Ejemplo: *lr2.py*
- El *momentum* puede suavizar la progresión del aprendizaje que, a su vez, puede acelerar el proceso de entrenamiento.
- Ejemplo adaptado para evaluar el efecto del *momentum* con *learning rate* fijo de 0.01.
- La función *fit_model()* se actualiza para tomar *momentum* variable.

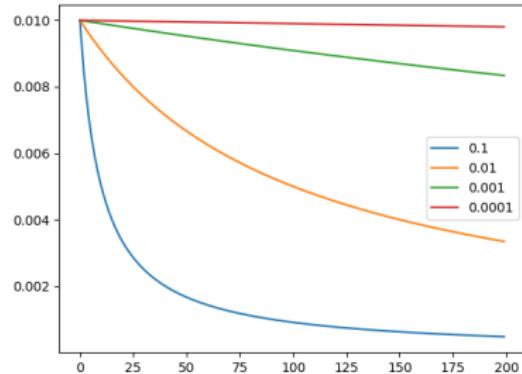


- Los gráficos muestran que agregar *momentum* acelera el entrenamiento. Valores de 0.9 y 0.99 logran *accuracy* razonable en entrenamiento y testing en app. 50 *epochs*, comparado con 200 *epochs* cuando no se usa *momentum*
- En todos los casos, el uso de *momentum* muestra un comportamiento más estable de la *accuracy* del modelo.

Optimizers - Dinámica de Learning Rate Decay

- Ejemplo: *lr3.py*
- SGD permite argumento *decay*, pero no sería claro desde la ecuación o el código su efecto en el *learning rate* luego de cada actualización.
- Primero, se define la función *decay_lrate* que implementa el *learning rate decay* de SGD.

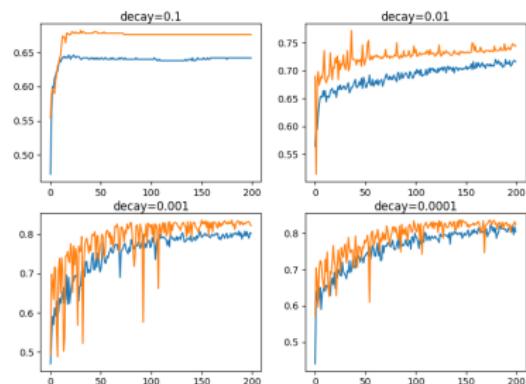
```
def decay_lrate(initial_lrate, decay, iteration):
    return initial_lrate * (1.0 / (1.0 + decay * iteration))
```



- Se usa esta función para calcular el learning rate en múltiples actualizaciones con valores de *decay* = [$1e-1, 1e-2, 1e-3, 1e-4$], con *learning rate* inicial de 0.01 y 200 actualizaciones.
- Decay de $1e-4$ (rojo) casi sin efecto; uno grande de $1e-1$ (azul) con efecto muy fuerte, reduciendo *learning rate* bajo 0.002 en 50 epochs y llega a cerca de 0.0004.
- Cambio no lineal y dependiente de *batch size* (momento en que se hace cada actualización). E.g., en problema anterior, *batch size* de 32 en 500 ejemplos resulta en 16 actualizaciones por epoch y 3200 actualizaciones en 200 epochs.

Optimizers - Dinámica de Learning Rate Decay

- Ejemplo: *lr4.py*
- Del análisis anterior, de fijar un *learning rate* de 0.01, no usando *momentum*, se esperaría un *decay* bajo, ya que uno muy grande resultaría rápidamente en un *learning rate* demasiado pequeño para aprender con efectividad.
- En este ejemplo, se adapta *fit_model()* para tomar los distintos *decay* evaluados en el ejemplo anterior.
 - Se observa que valores grandes de *decay* ($1e-1$ y $1e-2$) hacen decaer muy rápido el *learning rate* para este modelo, resultando en un rendimiento pobre. Los más pequeños resultan en mejor rendimiento, con $1e-4$ siendo similar a no usar *decay*.



Optimizers - Reducción de Learning Rate en Plateau

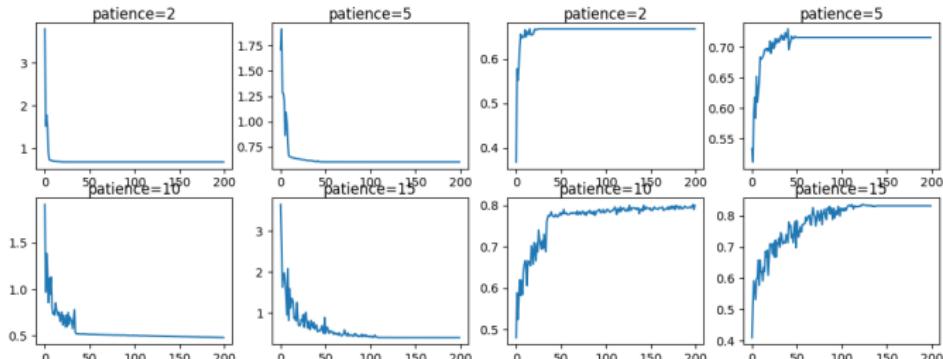
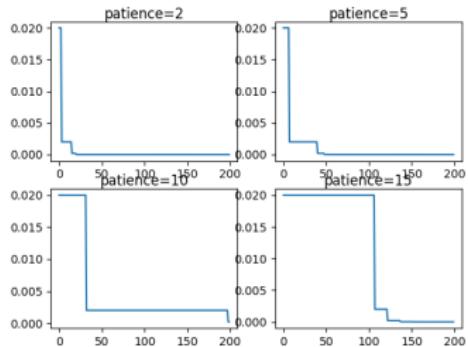
- Ejemplo: *lr5.py*
- El **Callback ReduceLROnPlateau** hará descender el *learning rate* por un factor luego de que métrica monitoreada sin cambios después de número dado de *epochs*. Se analizan distintos valores de *patience*, con *lr=0.02* y *factor=0.1*.
- Se puede monitorear efecto en *learning rate* creando nuevo *callback* que guarde el *learning rate* al final de cada *epoch*: *LearningRateMonitor*, con función *on_train_begin()* que se llama al inicio del entrenamiento; luego, la función *on_epoch_end()* se llama al final de cada *epoch*.

```
class LearningRateMonitor(Callback):  
    def on_train_begin(self, logs={}):  
        self.lrates = list()  
  
    def on_epoch_end(self, epoch, logs={}):  
        optimizer = self.model.optimizer  
        lrate = float(backend.get_value(self.model.optimizer.lr))  
        self.lrates.append(lrate)  
    ....  
    rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.1,  
                             patience=patience, min_delta=1E-7)  
    lrm = LearningRateMonitor()  
    history = model.fit(trainX, trainy, validation_data=(testX, testy),  
                        epochs=200, verbose=0, callbacks=[rlrp, lrm])
```

- Ver clase *Callback* para detalles.

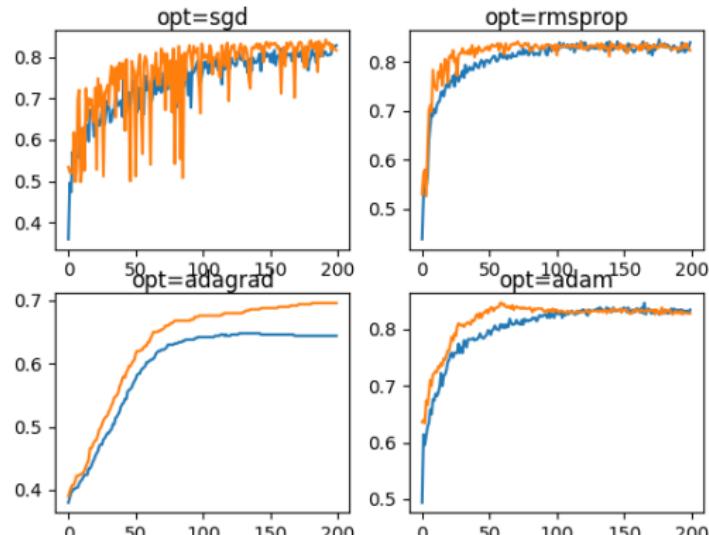
Optimizers - Reducción de Learning Rate en Plateau

- *learning rate*: *patience* más pequeño lo lleva a mínimo en menos de 25 epochs; el *patience* más grande recién desciende sobre 100.
- En general, los que permiten el uso de distintos *lr* por tiempo razonable debiesen obtener mejor rendimiento.
- Los valores de *patience* de 2 y 5 convergen muy rápido a loss sub-óptimo. En los niveles 10 y 15, el loss desciende razonablemente bajo a un nivel con grandes cambios en el loss.
 - Valores pequeños convergen prematuramente a app un 75% de accuracy. Los valores grandes llegan sobre el 80%.



Optimizers - Learning Rates Adaptivos

- Ejemplo: *lr6.py*
- No hay un sólo mejor algoritmo: depende del problema y el modelo.
- Se consideran los algoritmos SGD, RMSprop, Adagrad y Adam, con parámetros por defecto para comparación.
- SGD con *learning rate* por defecto 0.01 y sin momentum logra una buena *accuracy* para el problema, pero en 200 *epochs* y con alta volatilidad en el conjunto de entrenamiento y mucho más en el conjunto de testing.



- Adagrad en varias ejecuciones cayó en un sub-óptimo local y mostró un comportamiento si bien con baja volatilidad, con un rendimiento bajo en comparación.
- RMSProp y Adam mostraron rendimiento similar, siendo capaces de aprender el problema en 50 *epochs*.

Capas en Keras

- Una capa en Keras requiere:
 - *input_shape*: La forma (*shape*) de los datos de entrada
 - *Número de neuronas* en la capa.
 - *Inicializadores* de los pesos y los bias.
 - *Regularizadores* que intentarán optimizar la capa, aplicando penalizaciones de forma dinámica a los pesos durante el proceso de optimización.
 - *Restricciones* (constraints) que especifican el rango en el que se generan los pesos.
 - Las funciones de activación para la salida.

Capas en Keras

- Ejemplo:

```
1 import tensorflow as tf
2 from tensorflow.keras import \
3     layers, initializers, regularizers
4 from tensorflow.keras.models import Sequential
5
6 model = Sequential()
7 model.add(layers.Dense(32, input_shape=(16,),
8                     kernel_initializer = 'he_uniform',
9                     kernel_regularizer = None,
10                    kernel_constraint = 'MaxNorm',
11                    activation = 'relu'))
12 model.add(layers.Dense(16, activation = 'relu'))
13 model.add(layers.Dense(8))
```

- En línea 7 se agrega una capa a modelo secuencial, densa (Dense) con 32 neuronas. Si es la primera capa, se debe definir *input_shape* (en este caso, vector de 16 valores).
- En modelo secuencial se asume para capas siguientes, que salida de capa previa es entrada de la siguiente.
- Todos los demás parámetros son opcionales.

Capas en Keras - Inicializadores

Los inicializadores son diferentes funciones para inicializar los pesos:

- **Zeros:** Inicializa en 0 todos los pesos.

```
my_init = tf.keras.initializers.Zeros()  
model = Sequential()  
model.add(Dense(512, activation = 'relu', input_shape = (784,),  
    kernel_initializer = my_init))
```

- **Ones:** Inicializa en 1 for all input data.

```
my_init = initializers.Ones()
```

- **Constant:** Inicializa con valor constante todos los pesos.

```
my_init = initializers.Constant(value = 5)
```

- **RandomNormal:** Genera valores usando una distribución normal.

```
my_init = initializers.RandomNormal(mean=0.0, stddev = 0.05,  
    seed = None)
```

seed es la semilla aleatoria (al repetir se repite secuencia).

- **RandomUniform:** Genera valores usado distribución uniforme.

```
my_init = initializers.RandomUniform(minval = -0.05,  
    maxval = 0.05, seed = None)
```

seed es la semilla aleatoria (al repetir se repite secuencia).

Capas en Keras - Inicializadores

- *TruncatedNormal*: Genera valores usando una distribución normal truncada. Funciona igual que *RandomNormal*, pero los valores alejados más de dos desviaciones estándar de la media son descartados y regenerados. Es uno de los recomendados para pesos de redes neuronales y filtros.

```
my_init = initializers.TruncatedNormal(mean = 0.0, stddev = 0.05,  
                                         seed = None)
```

- *VarianceScaling*: Genera valores basados en las neuronas de entrada y/o la salida de la capa, basado en la escala especificada.

```
my_init = initializers.VarianceScaling(scale = 1.0,  
                                         mode = 'fan_in', distribution = 'normal', seed = None)
```

Si *distribution*=*"truncated_normal"* o *"normal"*/*"untruncated_normal"*, se generan muestras normales con media cero y $stddev = \sqrt{scale/n}$. Si *distribution*=*"uniform"*, las muestras son uniformes en $[-limit, limit]$, con $limit = \sqrt{3 * scale/n}$.

Valor *n* es número de entradas si *mode* = *"fan_in"*, número de salidas si *mode* = *"fan_out"*, y promedio entre entrada y salida si *mode* = *"fan_avg"*.

Capas en Keras - Inicializadores

- *lecun_normal*: Genera valores usando la distribución normal de [Lecun et al., 1998].

```
my_init = initializers.lecun_normal(seed = None)
```

Usa $stddev = \sqrt{1/fan_in}$, con *fan_in* el número de entradas.

- *lecun_uniform*: Genera valores usando la distribución uniforme de [Lecun et al., 1998].

```
my_init = initializers.lecun_uniform(seed = None)
```

Usa distribución uniforme en $[-limit, limit]$, con $limit = \sqrt{3/fan_in}$, *fan_in* el número de entradas.

- *GlorotNormal*: Genera valores usando la distribución normal de [Glorot and Bengio, 2010].

```
my_init = initializers.GlorotNormal(seed=None)
```

Usa $stddev = \sqrt{2/(fan_in + fan_out)}$, con *fan_in* número de entradas y *fan_out* número de salidas.

- *GlorotUniform*: Genera valores usando la distribución uniforme de [Glorot and Bengio, 2010].

```
my_init = initializers.GlorotUniform(seed=None)
```

Usa distribución uniforme en $[-limit, limit]$, con $limit = \sqrt{6/(fan_in + fan_out)}$, *fan_in* número de entradas y *fan_out* número de salidas.

Capas en Keras - Inicializadores

- *he_normal*: Genera valores usando la distribución normal de [He et al., 2015a].

```
my_init = initializers.he_normal(seed = None)
```

Usa $stddev = \sqrt{2/fan_in}$, con fan_in el número de entradas.

- *he_uniform*: Genera valores usando la distribución uniforme de [He et al., 2015a].

```
my_init = initializers.he_uniform(seed = None)
```

Usa distribución uniforme en $[-limit, limit]$, con $limit = \sqrt{6/fan_in}$, fan_in el número de entradas.

- *Orthogonal*: Genera una matriz ortogonal aleatoria, para tensores de pesos de dos o más dimensiones.

```
my_init = initializers.Orthogonal(gain = 1.0, seed = None)
```

con *gain* factor multiplicativo de la matriz.

- *Identity*: Genera una matriz de identidad. Sólo para matrices 2D.

```
my_init = initializers.Identity(gain = 1.0)
```

Capas en Keras - Restricciones (Constraints)

Una restricción (constraint) se aplicará a un parámetro (peso) durante la fase de optimización.

- *NonNeg*: Restringe a los pesos de ser no negativos.

```
from tensorflow.keras import constraints
my_constraint = constraints.NonNeg()
model = Sequential()
model.add(Dense(512, activation = 'relu', input_shape = (784, ),
    kernel_constraint = my_constraint))
```

- *UnitNorm*: Restringe a que los pesos incidentes en cada neurona escondida tengan norma unitaria.

```
my_constraint = constraints.UnitNorm(axis=0)
```

axis es el eje en torno al cual se calculan las normas de los pesos. Por ejemplo, en una capa Dense la matriz de pesos tiene forma (*dim_entrada*, *dim_salida*), se fija *axis*=0 para restringir aplicación a cada vector de largo (*dim_entrada*,). En capas Conv2D con *data_format*="channels_last", el tensor de pesos tiene *shape*=(*filas*, *columnas*, *dim_entrada*, *dim_salida*), y se fija *axis*=[0, 1, 2] para restringir los pesos de cada filtro tensor de tamaño (*filas*, *columnas*, *dim_entrada*).

Capas en Keras - Restricciones (Constraints)

- *MaxNorm*: Restringe los pesos asociados a cada neurona escondida a tener una norma menor o igual a un valor especificado.

```
my_constraint = constraints.MaxNorm(max_value=2, axis=0)
```

con *max_value* el valor máximo, y *axis* como en *UnitNorm*.

- *MinMaxNorm*: Restringe los pesos asociados a cada neurona escondida a tener una norma que se encuentre entre un mínimo y máximo especificados.

```
my_constraint = constraints.MinMaxNorm(min_value = 0.0,  
                                         max_value = 1.0, rate = 1.0, axis = 0)
```

con *min_value* el mínimo, *max_value* el máximo y *axis* como en *UnitNorm*.

El valor *rate* es para forzar la restricción: los pesos se reescalarán para cumplir $(1 - \text{rate}) * \text{norm} + \text{rate} * \text{norm.clip}(\text{min_value}, \text{max_value})$. Esto significa que *rate=1.0* hará forzar de forma estricta la restricción, mientras *rate < 1.0* significará pesos reescalados en cada paso para moverse hacia un valor en el intervalo.

Capas en Keras - Regularización

- Problema central en deep learning es cómo hacer que el modelo se ajuste no sólo al conjunto de entrenamiento, sino que también al conjunto de testing.
- *Overfitting*: modelo funciona bien para el conjunto de entrenamiento, pero pobremente para el de testing.
- Implica, hacer que el modelo *generalice* bien los ejemplos de entrenamiento → técnicas de regularización.
- *Regularización*: cualquier modificación al proceso de aprendizaje con el fin de reducir el error de generalización, sin aumentar el error de entrenamiento.
- Entonces, los regularizadores se usan en la fase de optimización, aplicando penalizaciones a los parámetros del modelo por cada capa.

Regularización - Penalización a parámetros

- Agregar una norma de penalización de parámetro a la función loss:
 $\hat{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$
con α hiperparámetro para contribución relativa de la norma de penalización Ω .
- En redes neuronales se usa para penalizar los pesos, pero no los *bias*: requieren menos datos para ajustar con precisión que los pesos (los pesos establecen relación entre dos variables y requiere observarlas a ambas en diversas condiciones).
- Se usan las normas $L1$ y $L2$.
- Aunque se considera un efecto marginal, de todas formas se puede aplicar a los *bias*.

Regularización de Parámetros L2

- Conocida como *weight decay*, lleva a los pesos más cerca del origen, con:
$$\Omega(\theta) = \frac{1}{2}||w||_2^2$$
- La función loss entonces queda:
$$\hat{J}(\theta; X, y) = \frac{\alpha}{2} w^T w + J(\theta; X, y)$$
- 1/2 se agrega para que el gradiente dé αw
- La regularización L2 penaliza fuertemente aquellos vectores de pesos con peaks muy pronunciados y tiende a aplanar los vectores de pesos: dadas las interacciones multiplicativas entre pesos y entradas, esto tiene la propiedad de alentar a la red de **utilizar todas sus entradas un poco**, en vez de sólo algunas mucho.
- Significa que ninguna entrada puede tener una influencia muy significativa por sí misma, lo que ayuda a la **generalización de la red**, y por ende, reduce el *overfitting*.

Regularización de Parámetros L1

- Esta regularización tiene la siguiente forma:

$$\Omega(\theta) = ||w||_1 = \sum_i |w_i|$$

- La función loss entonces queda:

$$\hat{J}(\theta; X, y) = \alpha ||w||_1 + J(\theta; X, y)$$

- La regularización L1 tiene la propiedad de generar vectores de pesos sparse (i.e. las neuronas terminan usando sólo un subconjunto sparse de sus más importantes entradas, volviéndose casi invariante a entradas ruidosas).
- En la práctica, si no se está interesado en selección explícita de características, se espera que la regularización L2 tenga mejor rendimiento que la L1.
- Se pueden usar las dos juntas: *elastic net regularization*.

Regularización de Parámetros L1/L2

- Uso en Keras, dentro de una capa:

```
layer = layers.Dense(  
    units=64,  
    kernel_regularizer=regularizers.l1(1e-5),  
    bias_regularizer=regularizers.l2(1e-4),  
)
```

- Permite regularizar pesos (*kernel*) y *bias*.

```
#Regularización L1:  
    kernel_regularizer=regularizers.l1(1e-5)  
#Regularización L2:  
    kernel_regularizer=regularizers.l2(1e-4)  
#Elastic net regularization:  
    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
```

Regularización - Otros mecanismos

- *Dataset Augmentation:* Entrenar con un conjunto de entrenamiento muy grande es la mejor forma de generalizar. Si no se dispone de muchos ejemplos, se pueden aumentar modificando diversos parámetros de los ejemplos originales: cambio de ancho, cambio de altura, flip horizontal/vertical, rotaciones, zoom, y cambio de brillo.

<https://machinelearningmastery.com/>

how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

- *Término anticipado del entrenamiento:* Exceso de tiempo en entrenamiento también incrementa el overfitting. Estrategias: cuando error de validación empieza a aumentar, se congelan los parámetros y se termina el entrenamiento; guardar copia de parámetros cada vez que error de validación mejora y retornar estos parámetros en vez de los últimos, cuando el entrenamiento termine.
- *Bagging:* Reduce error de generalización combinando varios modelos: entrenar varios modelos por separado; luego en inferencia cada modelo vota por la salida (model averaging). Estas técnicas se conocen como métodos *ensemble*, eficientes pues modelos diferentes no cometan los mismos errores.
- *Dropout:* Implementado como capa, se verá a continuación.

Capas en Keras - Capa Densa

- *Dense*: capa completamente conectada de una red neuronal.

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential()
layer_1 = layers.Dense(16, input_shape = (8,))
model.add(layer_1)
print("Shape de la entrada:", layer_1.input_shape)
print("Shape de la salida:", layer_1.output_shape)
print("Units (neuronas):", layer_1.units)
print("Función de activación:", layer_1.activation)
```

Shape de la entrada: (None, 8)

Shape de la salida: (None, 16)

Units (neuronas): 16

Activation: <function linear at 0x7fb3a23b0048>

- Toda capa tendrá *batch size* como la primera dimensión. Por eso *input_shape = (None, 8)* y *output_shape = (None, 16)*. Por el momento, *batch size* es *None* dado que no ha sido fijado (se hace usualmente en fase de entrenamiento).
- En la inicialización, *input_shape* es un argumento especial, que la capa aceptará sólo si es la primera del modelo.

Capas en Keras - Capa Densa

- Otros atributos:

```
print("Use bias:", layer_1.use_bias)
print("Kernel Initializer:", layer_1.kernel_initializer)
print("Bias Initializer:", layer_1.bias_initializer)
print("Kernel Regularizer:", layer_1.kernel_regularizer)
print("Bias Regularizer:", layer_1.bias_regularizer)
print("Kernel Constraint:", layer_1.kernel_constraint)
print("Bias Constraint:", layer_1.bias_constraint)
```

```
Use bias: True
Kernel Initializer: <tensorflow.python.ops.init_ops_v2
                    .GlorotUniform object at 0x7fb418f5e438>
Bias Initializer: <tensorflow.python.ops.init_ops_v2
                    .Zeros object at 0x7fb418f5e4e0>
Kernel Regularizer: None
Bias Regularizer: None
Kernel Constraint: None
Bias Constraint: None
```

- Código en *layers-dense.py*

Capas en Keras - Métodos

- Métodos comunes a todas las capas:

- `get_weights()`: Obtiene todos los parámetros de la capa (pesos y bias).

```
weights = layer_1.get_weights()  
print("Tamaño de resultado:", len(weights))  
print("Shape pesos:", weights[0].shape)  
print("Shape bias:", weights[1].shape)
```

Tamaño de resultado: 2

Shape pesos: (8, 16)

Shape bias: (16,)

- `set_weights(weights)`: Asigna parámetros.

- `get_config()`: Obtiene la configuración completa de la capa como un objeto.

- `from_config(config)`: Carga configuración del objeto a la capa.

```
config = layer_1.get_config()  
model2 = keras.Sequential()  
layer_2 = layers.Dense.from_config(config)  
model2.add(layer_2)
```

Shape de la entrada: (None, 8)

Shape de la salida: (None, 16)

Capas en Keras - Capa Convolucional

Capa Convolucional: Keras contiene varios tipos de capas convolucionales. Las más utilizadas: *Conv1D*, *Conv2D* (imágenes), *Conv3D* (video?).

https://keras.io/api/layers/convolution_layers/

Características particulares:

- *filters*: Número de filtros a ser aplicados.
- *kernel size*: Tamaño de cada filtro.
- *strides*: El paso al que avanzan los filtros.
- *padding*: Agregado a la entrada para ajustar la aplicación de filtros. Valores posibles:
 - *valid*: sin padding.
 - *causal*: convolución causal (se aplica hacia la izquierda en 1D (pasado)).
 - *same*: padding aplicado para que salida quede con iguales dimensiones que la entrada (imagen quede de igual tamaño).
- *dilation_rate*: tasa de separación de los coeficientes de la convolución.
- *data_format*: Valores posibles:
 - *channel_last*: canales en última entrada. Por defecto en Keras.
 - *channel_first*: canales en segunda entrada, luego del *batch_size*.

Ejemplo, entrada de imagen de 28×28 BGR, con *data_format=channel_last* sería $(4, 28, 28, 3)$ para un *batch_size=4*. Si *data_format=channel_first* sería $(4, 3, 28, 28)$.

Capa Convolucional - Conv1D

- *Conv1D*: Se usa en CNN con base temporal.
 $input_shape = (batch_size, timesteps, features)$
 $output_shape = (batch_size, new_steps, filters)$

- Formato:

```
tf.keras.layers.Conv1D(  
    filters,  
    kernel_size,  
    strides=1,  
    padding="valid",  
    data_format="channels_last",  
    dilation_rate=1,  
    groups=1,  
    activation=None,  
    use_bias=True,  
    kernel_initializer="glorot_uniform",  
    bias_initializer="zeros",  
    kernel_regularizer=None,  
    bias_regularizer=None,  
    kernel_constraint=None,  
    bias_constraint=None,  
)
```

- *filters* y *kernel_size* son los únicos obligatorios.
- Actualmente, valor de *dilation_rate != 1* es incompatible con *strides != 1*.
- *groups*: Entero positivo; número de grupos para cortar el eje de canales. Cada grupo se convoluciona separadamente con *filters/groups* filtros. La salida resulta de la concatenación de todos los resultados de los grupos en el eje de canales. Número de canales y filtros deben ser divisibles por *groups*.
- Ver *layers-conv1D.py*

Capa Convolucional - Conv2D

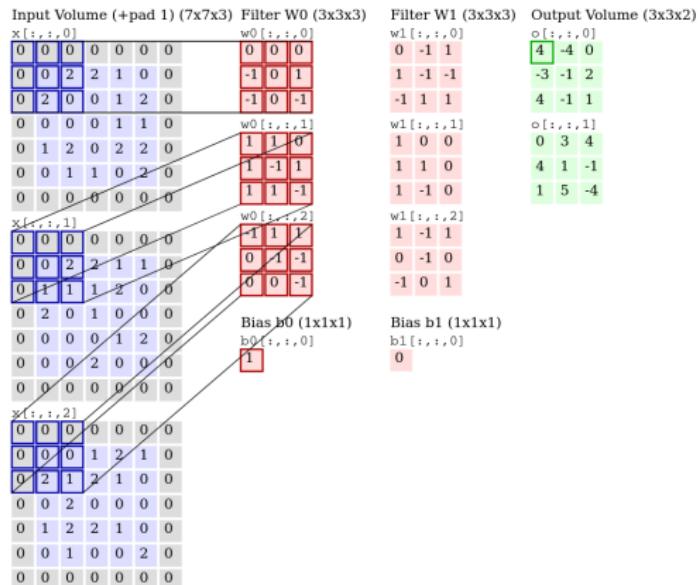
- *Conv2D*: Se usa para imágenes, con: $input_shape = (batch_size, H, W, channels)$ ($data_format=channel_last$) $output_shape = (batch_size, new_dim1, new_dim2, filters)$
- Ejemplo, entrada de imagen de 28×28 BGR, con $data_format=channel_last$ sería $(4, 28, 28, 3)$ para un $batch_size=4$. Si $data_format=channel_first$ sería $(4, 3, 28, 28)$.
- Formato:

```
tf.keras.layers.Conv2D(  
    filters,  
    kernel_size,  
    strides=(1, 1),  
    padding="valid",  
    data_format=None,  
    dilation_rate=(1, 1),  
    #...Igual que Conv1D ...  
)
```

- *filters* y *kernel_size* son los únicos obligatorios.
- Actualmente, valor de *dilation_rate* $\neq 1$ es incompatible con *strides* $\neq 1$. Puede definirse en general o por dimensiones, al igual que *stride*.
- No acepta *padding causal* ni *groups*.
- Ver *layers-conv2D.py*

Capa Convolucional - Conv2D

- Recordar esta imagen con $stride=(2, 2)$ y $padding='same'$.



[CS231n Stanford]

- $input_shape = (None, 5, 5, 3)$
- $output_shape = (None, 3, 3, 2)$
- $Shape\ de\ pesos = (3, 3, 3, 2)$

Capa Convolucional - Flatten

- Implementado como capa, en realidad lo que hace es dejar toda la entrada en una sola dimensión.
- Ideal para ser usado entre una capa *Conv* y una *Dense*.
- Tiene solo argumento *data_format*, que funciona igual que en capas de convolución.
- Ejemplo:

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
                       border_mode='same',
                       input_shape=(3, 32, 32)))
# aqui: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# ahora: model.output_shape == (None, 65536)
# 64x32x32 = 65536
```

Ejemplo 1 - MNIST

- Ejemplo en *mnist-conv2D.py* y *mnist-conv2D-client.py*
- El primero realiza entrenamiento y guarda el modelo:
Conv2D, *Flatten*, *Dense*.
- El segundo, hace inferencia a una imagen aleatoria del test set.

Capa Convolucional - Conv3D

- *Conv3D*: Se usa para secuencias de imágenes (frames 3D), con: *input_shape* = (*batch_size*, *S*, *H*, *W*, *channels*) (*data_format=channel_last*, con *S* largo de la secuencia) *output_shape* = (*batch_size*, *S*, *new_dim1*, *new_dim2*, *filters*)
- Ejemplo, entrada de secuencia de 10 imágenes de 640×480 BGR, con *data_format=channel_last* sería (4, 10, 480, 640, 3) para un *batch_size=4*. Si *data_format=channel_first* sería (4, 3, 10, 480, 640).
- Formato:

```
tf.keras.layers.Conv3D(  
    filters,  
    kernel_size,  
    strides=(1, 1, 1),  
    padding="valid",  
    data_format=None,  
    dilation_rate=(1, 1, 1),  
    #...Igual que Conv2D ...  
)
```

- *filters* y *kernel_size* son los únicos obligatorios.
- Actualmente, valor de *dilation_rate != 1* es incompatible con *strides != 1*.
- No acepta *padding causal* ni *groups*.
- Ver *layers-conv3D.py*

Capas en Keras - Capa de Pooling

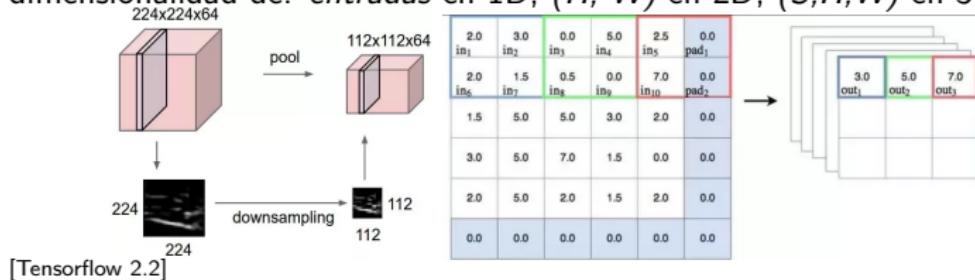
- La capa de *pooling* es preferentemente utilizada inmediatamente después de una capa convolucional.
- Se usa para reducir el tamaño (sólo ancho y alto, no profundidad), overfitting y la computación.
- El más común es *Max pooling* que aplica a una ventana la operación máximo. El parámetro *stride* se aplica igual que en convolucional.
- En Keras hay tres versiones, para distintas dimensiones de entrada (para *data_format='channels_last'*):
 - *MaxPooling1D*: Entradas en formato (*batch_size, entradas, canales*).
 - *MaxPooling2D*: Entradas en formato (*batch_size, H, W, canales*)
 - *MaxPooling3D*: Entradas en formato (*batch_size, S, H, W, canales*)
- Para *data_format='channels_first'*, la dimensión canales queda a la derecha de *batch_size*

Capas en Keras - Capa de Pooling - Max Pooling

- Construcción en Keras.

```
tf.keras.layers.MaxPooling1D(pool_size=2, strides=None,  
                             padding="valid", data_format=None)  
tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None,  
                             padding="valid", data_format=None)  
tf.keras.layers.MaxPooling3D(pool_size=(2, 2, 2), strides=None,  
                             padding="valid", data_format=None)
```

- Tanto en *pool_size*, como en *strides*, puede ser simplemente un número para más de una dimensión, donde el número se aplica a cada dimensión.
- *Padding*: opción "valid" no aplica; opción "same" aplica al final de la dimensión, si necesario.
- Por cada ejemplo del batch se aplica *MaxPooling*, operando independiente por cada canal; de acuerdo a *pool_size* y *strides*, sólo reduce dimensionalidad de: *entradas* en 1D; (*H*, *W*) en 2D; (*S,H,W*) en 3D.



Capas en Keras - Capa de Pooling - Max Pooling Global

- Capa que obtiene máximo global de la entrada:

```
tf.keras.layers.GlobalMaxPooling1D(data_format=None)  
tf.keras.layers.GlobalMaxPooling2D(data_format=None)  
tf.keras.layers.GlobalMaxPooling3D(data_format=None)
```

- Acepta mismas entradas que *MaxPooling*.
- Obtiene el máximo para cada entrada del batch, con canales por separado.
- Por cada ejemplo del batch se aplica *GlobalMaxPooling*, obteniendo el valor máximo independientemente por cada canal, para: *entradas* en 1D; (H, W) en 2D; (S, H, W) en 3D.

Capas en Keras - Capa de Pooling - Max Pooling

- Ejemplo:

```
x = tf.constant([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16])
x = tf.reshape(x, [1, 2, 4, 2])
Batch_size=1, H=2, W=4, canales=2.
Canal 1           Canal 2
1    3    5    7      2    4    6    8
9   11   13   15     10   12   14   16

max_pool_2d = tf.keras.layers.MaxPooling2D(pool_size=(2, 2),
    strides=(1,1), padding='same')
y = max_pool_2d(z2)
Hace padding para mantener shape=(1, 2, 4, 2)
Canal 1           Canal 2
11   13   15   15     12   14   16   16
11   13   15   15     12   14   16   16

max_pool_2d = tf.keras.layers.MaxPooling2D(pool_size=(2, 2),
    strides=(1,1), padding='valid')
y = max_pool_2d(z2)
No aplica padding, shape=(1, 1, 3, 2)
Canal 1           Canal 2
11   13   15       12   14   16

gmax_pool_2d = tf.keras.layers.GlobalMaxPooling2D()
y = gmax_pool_2d(z2)
Suprime dimensiones (H,W): shape=(1, 2)
Canal 1           Canal 2
15                  16
```

Capa de Pooling - Average Pooling

- Funciona igual que *MaxPooling*, pero con el promedio.
- Local:

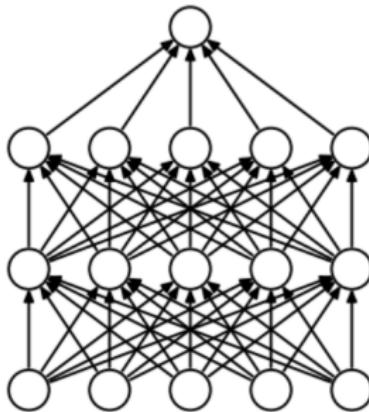
```
tf.keras.layers.AveragePooling1D(pool_size=2,  
                                 strides=None, padding="valid",  
                                 data_format=None)  
tf.keras.layers.AveragePooling2D(pool_size=(2, 2),  
                                 strides=None, padding="valid",  
                                 data_format=None)  
tf.keras.layers.AveragePooling3D(pool_size=(2, 2, 2),  
                                 strides=None, padding="valid",  
                                 data_format=None)
```

- Global:

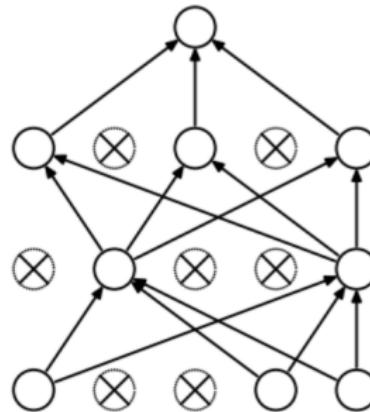
```
tf.keras.layers.GlobalAveragePooling1D(data_format=None)  
tf.keras.layers.GlobalAveragePooling2D(data_format=None)  
tf.keras.layers.GlobalAveragePooling3D(data_format=None)
```

Capas en Keras - Dropout

- Mecanismo de control de *overfitting*.
- En simple, *Dropout* se refiere a ignorar neuronas durante la fase de entrenamiento escogidas aleatoriamente: neuronas no consideradas en algún paso de optimización en particular.
- En cada iteración de entrenamiento, neuronas no consideradas con probabilidad $1-p$ (participan en actualización de pesos, con tasa p).



Red normal



Después de capa Dropout

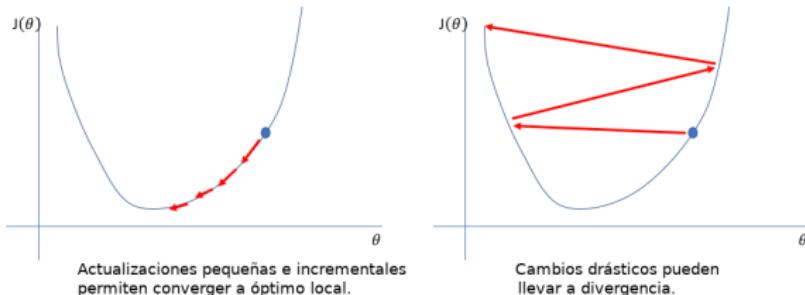
- En Keras (con $rate=p$):
`tf.keras.layers.Dropout(rate)`

Capas en Keras - Dropout

- Dropout fuerza a la red a aprender características más robustas. Este mecanismo resolvió parte del *overfitting* para redes grandes, permitiendo arquitecturas más grandes y precisas; complementario con otras técnicas de regularización.
- Resolvió problema de redes grandes: *co-adaptación*
 - Si se entranan todos los pesos juntos, es común que algunas conexiones tengan mayor poder predictivo.
 - Como se entrena iterativamente, las conexiones fuertes se refuerzan más y las débiles se ignoran.
 - Entonces, expandir la red ya no ayuda y, por ende, el tamaño y *accuracy* de las redes estaba limitado.
- Valores recomendados: $p=0.5$ para capa escondida; $p=0.8$ para capa de entrada.
- Dropout aproximadamente dobla el número de iteraciones requeridas para converger, pero el tiempo de entrenamiento por *epoch* es menor.

Normalización de la Entrada

- Todos los métodos de optimización de *loss* escalan la magnitud de la actualización por el *learning rate*: controla no hacer cambios muy drásticos en los pesos entre iteraciones, que eviten converger al óptimo.



- Importante considerar que la superficie de la función *loss* está caracterizada por los valores de los pesos en la red.
- Ejemplo de red con dos entradas: x_1 varía en $[0; 1]$ y x_2 en $[0; 0.01]$; como la red debe aprender a combinar las entradas con una serie de combinaciones y activaciones no lineales, los pesos asociados a cada entrada también existirán en diferentes escalas (puede llevar a una topología de la función *loss* que ponga más énfasis en gradientes sólo de ciertos pesos).

Normalización de la Entrada

- Normalizar todas las entradas a una escala estándar permite que la red aprenda más rápidamente los parámetros óptimos para cada nodo de entrada, pues permite que también los pesos tengan un comportamiento más controlado en el proceso de aprendizaje, lo que también permite usar *learning rates* más grandes.
- También es útil asegurar que las entradas se encuentren aproximadamente en torno al rango $[-1; 1]$ para evitar artifacts asociados a precisión de punto flotante.
- *Standard Scaler*: Escalar las entradas para tener media cero (restar media) y varianza unitaria (dividir por varianza); se ajusta al entrenamiento (media y varianza) y luego los parámetros del *scaler* deben ser usados en la fase de inferencia por toda nueva entrada.
- En imágenes es común simplemente escalar por $1/255$ para que las intensidades de cada canal se encuentren en $[0; 1]$ (y por simpleza).
- También, se considera útil descorrelacionar las entradas con PCA (*whitening*), pero es un proceso costoso y por eso no es tan frecuente.

Capas en Keras - Normalización de Batch

- *Batch normalization* reduce la cantidad en la que varían los valores de una capa escondida (*internal covariance shift*); sigue la lógica de la normalización de entradas, pero aplicado como capa de normalización en el modelo.
- *Internal covariate shift*: ocurre cuando hay un cambio en la distribución de la entrada de la red. Al cambiar, las capas escondidas tratan de aprender para adaptarse a la nueva distribución (lo que ralentiza el proceso → más tiempo para converger a mínimo). Ocurre cuando la distribución estadística de la entrada es drásticamente diferente de la entrada que se ha presentado hasta ese momento.
- Ejemplo de *covariance shift*: red para detección de gatos; primero se entrena con imágenes de gatos negros (aún no funciona bien para gatos de colores). El conjunto de entrenamiento y el de testing son ambos imágenes de gatos, pero difieren significativamente en algunas características.
- En resumen, si un modelo aprendió a mapear de X a Y, y la distribución de X cambia, entonces se necesita reentrenar la red tratando de realinear el mapeo de X a Y.

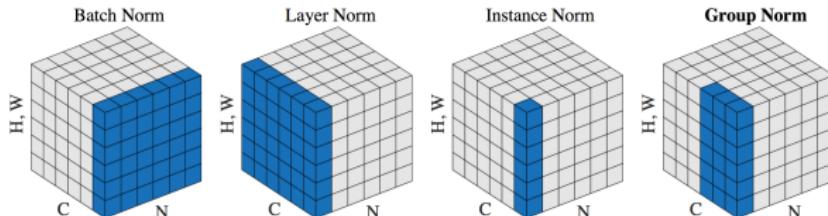
Capas en Keras - Normalización de Batch

- El transformar linealmente para media cero, varianza unitaria y descorrelacionar a la entrada para cada capa, permite tener distribuciones internas que eliminan los efectos del *internal covariate shift*.
- Capa *BatchNormalization* normaliza activaciones de capa anterior, para cada batch: aplica transformación que mantiene media cercana a 0 y s.d. cerca de 1.
- Se aplica antes de la activación (transformación lineal) y del Dropout.
- Ejemplo en Keras (*normalizacion1.py*):

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from matplotlib import pyplot
model = keras.models.Sequential()
#Capas por defecto con activación "linear" (sin activación):
model.add(layers.Dense(64, input_dim=16))
#Se aplica normalización a la salida lineal:
model.add(layers.BatchNormalization())
#Capa de activación (aplica la función):
model.add(layers.Activation('tanh'))
#... y finalmente el Dropout:
model.add(layers.Dropout(0.5))
model.add(layers.Dense(64))
model.add(layers.BatchNormalization())
model.add(layers.Activation('tanh'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(2))
model.add(layers.BatchNormalization())
model.add(layers.Activation('softmax'))
model.compile(loss='binary_crossentropy', optimizer="sgd")
model.summary()
```

Capas en Keras - Normalización de capas

- También es posible aplicar normalización a una capa de forma independiente por cada ejemplo de entrenamiento, en vez del batch, como en *BatchNormalization*.
- Aplica transformación que mantiene la activación media de cada ejemplo cerca de 0 y con s. d. cerca de 1.
- Construcción en Keras:
`tf.keras.layers.LayerNormalization()`
- Se usa insertando posterior a otra capa con activación *linear* y antes de aplicar otra función de activación o *Dropout*, igual que *BatchNormalization*.
- Existen incluso otros tipos de normalización en Tensorflow (en azul los que se normalizan por misma media y varianza) - C: canales, N: tamaño de batch:



Keras - API Funcional

- Otra forma de construir modelos, considerando cada capa como una función, con parámetro de entrada y retorno de salida.

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Model

#Crear nodo de entrada:
inputs = layers.Input(shape=(784,))

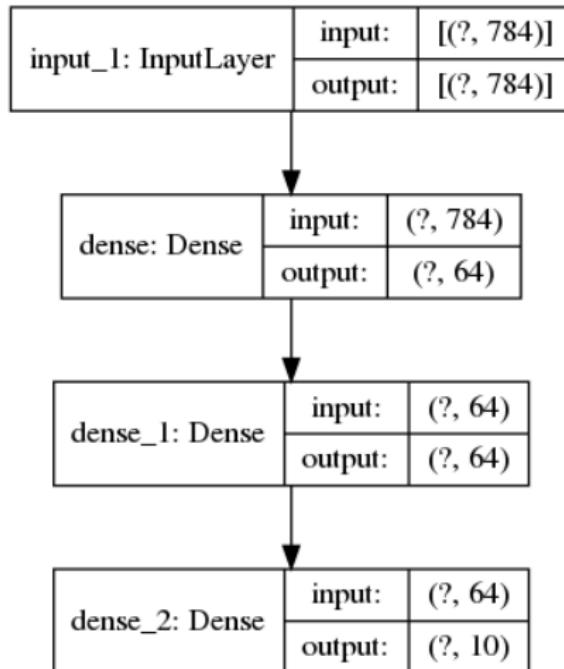
#Crear segunda capa:
dense = layers.Dense(64, activation='relu')
#Conectar tratando capa como función, con "inputs" como parámetro, y
# capturando salida para siguiente capa y así sucesivamente:
x = dense(inputs)

#Lo mismo para un par de capas más:
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10)(x)

#Construir modelo conectando primera capa con última:
model = Model(inputs=inputs, outputs=outputs, name='un_modelo')

#Visualizar:
model.summary()
tf.keras.utils.plot_model(model, 'grafico.png', show_shapes=True)
```

Capas en Keras - Normalización de capas



Keras - API Funcional

- No restringe a un modelo *Sequential* (agregar capas en orden), pues se basa en conectar de acuerdo a las entradas y salidas definidas para cada capa.
- Ahora podemos construir modelos no secuenciales:

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Model

#Crear nodo de entrada:
inputs = layers.Input(shape=(784,))

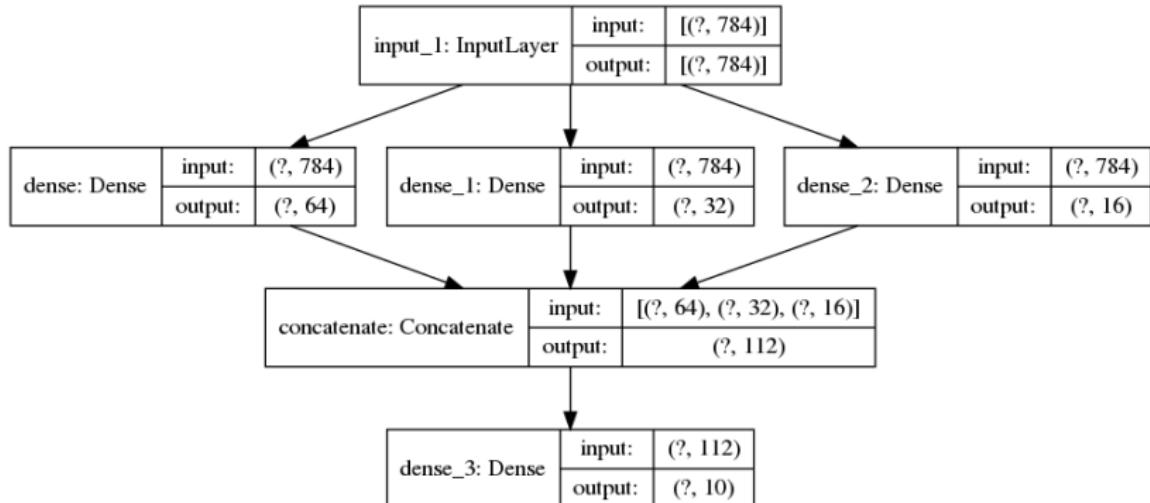
#Create segundas capas:
dense2 = layers.Dense(64, activation='relu')
dense3 = layers.Dense(32, activation='relu')
dense4 = layers.Dense(16, activation='relu')

#Conectar pasando a las capas, la misma entrada "inputs"
# como parámetro, y capturando salida para siguiente capa:
x = dense2(inputs)
y = dense3(inputs)
z = dense4(inputs)

#Concatenar salida, en torno al segundo eje (más adelante):
concat = layers.concatenate([x, y, z])
outputs = layers.Dense(10)(concat)

#Construir modelo conectando primera capa con la última:
model = Model(inputs=inputs, outputs=outputs, name='modelo_no_secuencial')
```

Capas en Keras - Normalización de capas



Capas en Keras - Fusión de capas (merge)

- Luego que se puede separar las capas como un grafo, es necesario en muchos casos juntar el resultado de varias capas.
- Hay varias formas. Una de ellas es concatenar. Ejemplo:

```
from keras.models import Model
from keras.layers import Dense, Merge, concatenate, Input
inp1 = Input(shape=(10,20))
inp2 = Input(shape=(10,32))
ccl = Concatenate(axis=2)([inp1, inp2]) # Une canales, mismos datos
output = Dense(30, activation='relu')(ccl)
model = Model(inputs=[inp1, inp2], outputs=output)
model.summary()
Layer (type)          Output Shape         Param #     Connected to
=====
input_40 (InputLayer) (None, 10, 20)      0
input_41 (InputLayer) (None, 10, 32)      0
concatenate_21 (Concatenate) (None, 10, 52) 0           input_40[0][0]
                                                input_41[0][0]
dense_12 (Dense)        (None, 10, 30)    1590       concatenate_21[0][0]
```

- Junta los datos de acuerdo al *axis* definido (dimensión de la *shape* de las entradas).
- El valor de las otras dimensiones (diferentes de *axis*) deben ser iguales entre todas las entradas.

Fusión de capas - concatenate

```
inp1 = Input(shape=(20,10))
inp2 = Input(shape=(32,10))
ccl = Concatenate(axis=1)([inp1, inp2]) # Une datos, mismos canales
output = Dense(30, activation='relu')(ccl)
model = Model(inputs=[inp1, inp2], outputs=output)
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
input_42 (InputLayer)	(None, 20, 10)	0	
input_43 (InputLayer)	(None, 32, 10)	0	
concatenate_22 (Concatenate)	(None, 52, 10)	0	input_42[0][0] input_43[0][0]
dense_13 (Dense)	(None, 52, 30)	1590	concatenate_22[0][0]

Fusión de capas - concatenate

```
inp1 = Input(shape=(10,10))
inp2 = Input(shape=(10,10))
ccl = Concatenate(axis=0)([inp1, inp2]) # Une batches, mismos datos y canales
output = Dense(30, activation='relu')(ccl)
model = Model(inputs=[inp1, inp2], outputs=output)
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
input_44 (InputLayer)	(None, 10, 10)	0	
input_45 (InputLayer)	(None, 10, 10)	0	
concatenate_23 (Concatenate)	(None, 10, 10)	0	input_44[0][0] input_45[0][0]
dense_14 (Dense)	(None, 10, 30)	1590	concatenate_23[0][0]

Fusión de capas - Operaciones

- Existen operaciones elemento a elemento que se pueden realizar para unir distintos elementos: *Average*, *Maximum*, *Minimum*, *Add*, *Subtract*, *Multiply*.
- Todos funcionan de misma forma: reciben

```
input1 = tf.keras.layers.Input(shape=(16, ))
x1 = tf.keras.layers.Dense(8, activation='relu')(input1)
input2 = tf.keras.layers.Input(shape=(32, ))
x2 = tf.keras.layers.Dense(8, activation='relu')(input2)
added = tf.keras.layers.Add()([x1, x2])
out = tf.keras.layers.Dense(4)(added)
model = tf.keras.models.Model(inputs=[input1, input2],
                               outputs=out)
```

- También existe la función producto punto *Dot*:

```
#Requiere definir axes: primer número es eje de x,
# segundo es eje de y:
tf.keras.layers.Dot(axes=(1, 2))([x, y])
#Para tensores de entrada shape=(batches, n),
# retorna shape=(batches, 1)
```

Keras - Transfer Learning

- *Transfer Learning* consiste en tomar características aprendidas para un problema y usarlas para un problema nuevo similar.
- Normalmente se usa para tareas donde el dataset tiene muy pocos datos para entrenar un modelo completo desde el inicio.
- Proceso normal de *transfer learning*:
 - ① Usar las capas de modelo previamente entrenado.
 - ② Congelar las capas, es decir, no actualizar los pesos para estas capas.
 - ③ Agregar nuevas capas entrenables, después de las capas congeladas.
 - ④ Entrenar las nuevas capas con el dataset.
- Último paso opcional es *fine-tuning*: descongelar el modelo previo (o parte de él) y reentrenarlo con nuevos datos con un *learning rate* muy bajo, adaptando incrementalmente las características pre-entrenadas a los nuevos datos.

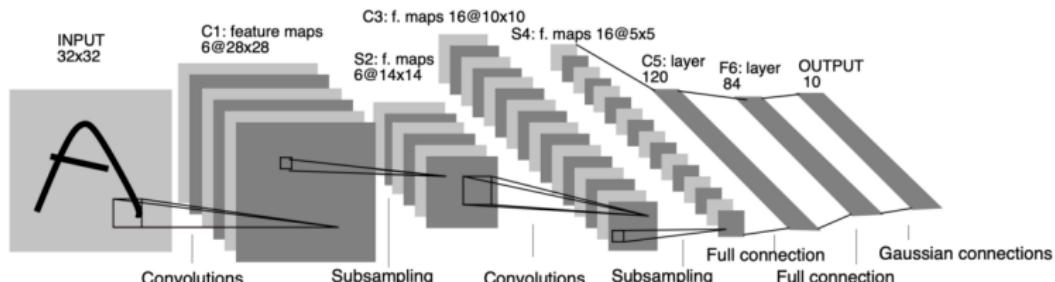
Keras - Arquitecturas existentes

- Modelos disponibles (resultado en clasificación de ImageNet):

Modelo	Tamaño	Top-1	Top-5	Parámetros
Xception	88 MB	0.790	0.945	22,910,480
VGG16	528 MB	0.713	0.901	138,357,544
VGG19	549 MB	0.713	0.900	143,667,240
ResNet50	98 MB	0.749	0.921	25,636,712
ResNet101	171 MB	0.764	0.928	44,707,176
ResNet152	232 MB	0.766	0.931	60,419,944
ResNet50V2	98 MB	0.760	0.930	25,613,800
ResNet101V2	171 MB	0.772	0.938	44,675,560
ResNet152V2	232 MB	0.780	0.942	60,380,648
InceptionV3	92 MB	0.779	0.937	23,851,784
InceptionResNetV2	215 MB	0.803	0.953	55,873,736
MobileNet	16 MB	0.704	0.895	4,253,864
MobileNetV2	14 MB	0.713	0.901	3,538,984
DenseNet121	33 MB	0.750	0.923	8,062,504
DenseNet169	57 MB	0.762	0.932	14,307,880
DenseNet201	80 MB	0.773	0.936	20,242,984
NASNetMobile	23 MB	0.744	0.919	5,326,716
NASNetLarge	343 MB	0.825	0.960	88,949,818
EfficientNetB0	29 MB	-	-	5,330,571
EfficientNetB1	31 MB	-	-	7,856,239
...				

Deep Learning - Arquitecturas

- *LeNet5 (1998)*: primera CNN, por Yann LeCun (Bell Labs) [LeCun et al., 1998]. Incluye capas convolucionales (dos de 5×5 , stride 1), de pooling (dos de 2×2 , stride 2) y densas (tres), con algoritmo backprop para entrenamiento, con activaciones *tanh* y loss *MSE*. Los autores lo usaron para detección de firmas manuscritas en cheques (uso comercial).

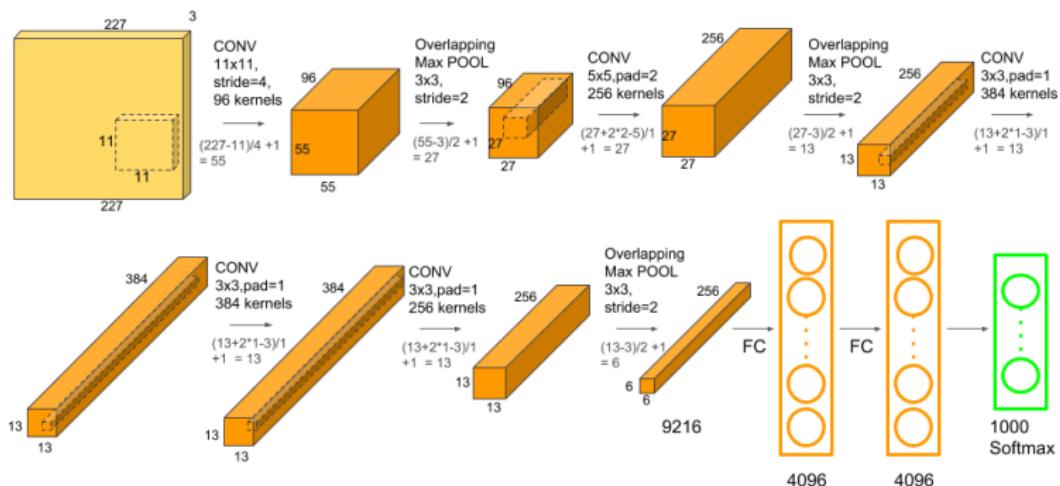


Deep Learning - Arquitecturas

- *Alexnet (2012)*: El desarrollo se frenó hasta el 2012 pues se requería entrenar millones de parámetros y se necesitaban datasets grandes (millones de imágenes). Datasets de imágenes de este tamaño recién aparecieron en 2010 (e.g. ImageNet). AlexNet mejoró significativamente los resultados de su antecesor [Krizhevsky et al., 2012] con arquitectura muy similar a LeNet5, pero con las siguientes mejoras:
 - Activación *ReLU* y *Cross Entropy loss*.
 - Conjunto de entrenamiento mucho más grande: LeNet5 entrenó con dataset MNIST (50,000 imágenes y 10 clases) y AlexNet usó subconjunto de ImageNet (1+ millón de imágenes y 1000 clases).
 - AlexNet usa *Dropout* (0.5) sólo en las capas densas. Usa también normalización por capas (en desuso porque no mejora significativamente el rendimiento de las CNN).
 - Entrenó usando GPUs: dos GTX 580 por 5-6 días.
 - Usa 5 capas convolucionales, combinadas con capas de pooling, con tres densas al final. La entrada al sistema eran imágenes de $227 \times 227 \times 3$. Tiene 60 millones de parámetros
 - Para prevenir *overfitting* también se usó Data Augmentation (15 millones de imágenes en total).

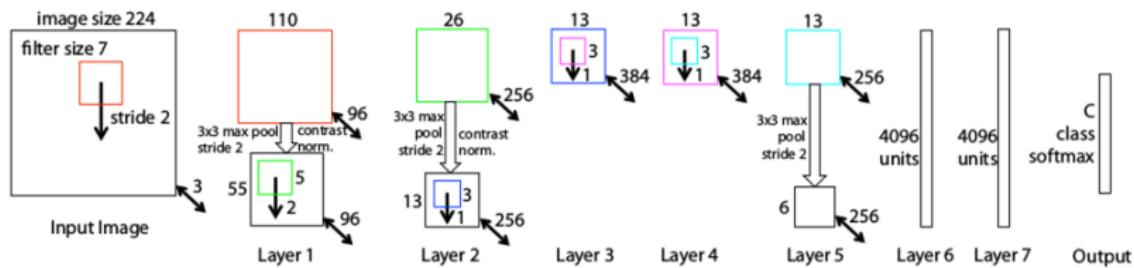
Deep Learning - Arquitecturas

- AlexNet (2013):



Deep Learning - Arquitecturas

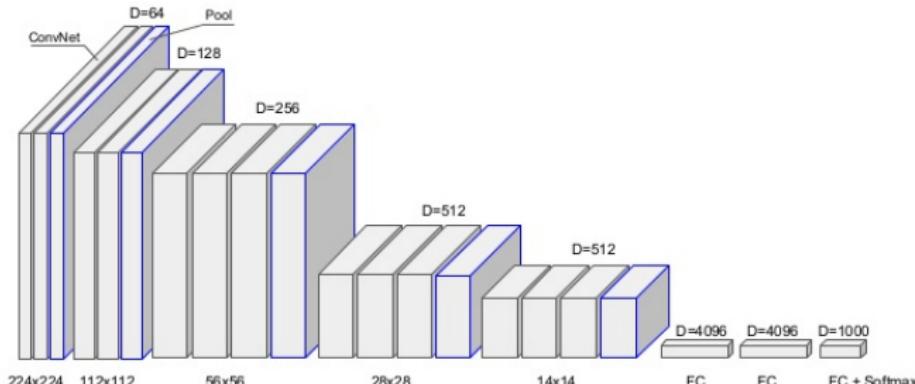
- *ZFNet (2013)*: Diseñada por Zeiler y Fergus [Zeiler and Fergus, 2013], la arquitectura tenía pequeñas modificaciones con respecto a AlexNet:
 - En primera capa convolucional usó filtro de 7×7 con stride 2 (características más finas), en vez de 11×11 de AlexNet.
 - Más mapas de activación en 3a, 4a y 5a capas convolucionales (más parámetros).



ZF Net Architecture

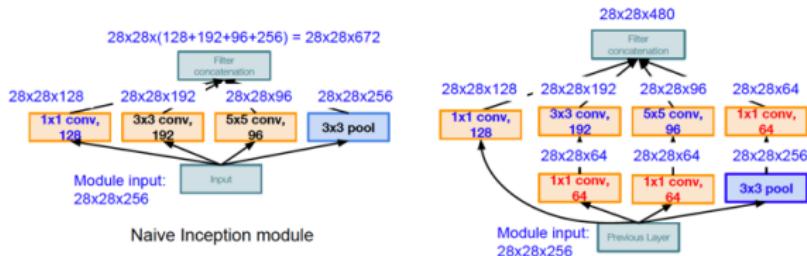
Deep Learning - Arquitecturas

- **VGGNet (2014)**: Simonyan y Zisserman [Simonyan and Zisserman, 2014] crearon la red VGGNet, cuyas mayores innovaciones fueron:
 - Número de capas: 19 (16 convolucionales + 3 densas).
 - Arquitectura simplificada, repitiendo misma aplicación de filtros de 3×3 , con stride y padding de 1, con MaxPooling de 2×2 , con stride 2 (mismo efecto de filtros más grandes, pero con menos parámetros y mayor no-linealidad del modelo).
 - 144 millones de parámetros (124 millones de ellos en las capas densas).



Deep Learning - Arquitecturas

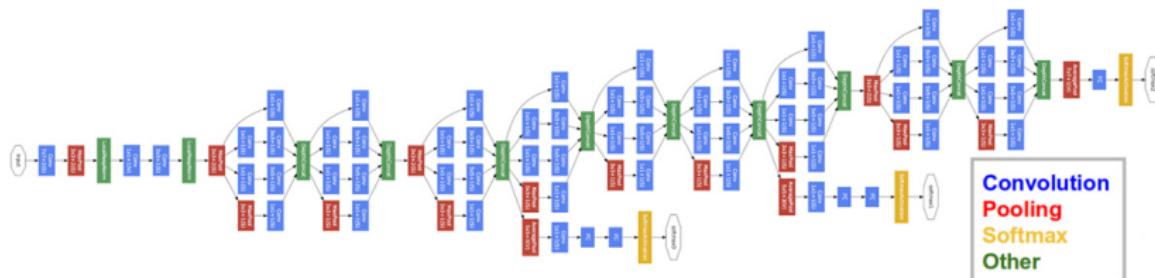
- *Inception Network (2014)*: Creada por Google [Szegedy et al., 2016] propuso una arquitectura de alta complejidad.
- Primer diseño que se aleja de un modelo *secuencial*. Reduce significativamente los parámetros usando las siguientes innovaciones:
 - Eliminación de todas las capas densas y uso de *AveragePooling* (sólo 5 millones de parámetros).
 - *Módulos Inception*: Estos módulos se repiten en la red Inception múltiples veces. El módulo procesa la entrada usando múltiples filtros en paralelo, antes de concatenar sus mapas de activación, formando la entrada de la siguiente etapa. Usa capas convolucionales de: 1×1 , 3×3 , 5×5 , y 3×3 con MaxPooling (extracción a distintos niveles de detalle).



Deep Learning - Arquitecturas

- *Inception Network (2014)*:

- Para evitar la explosión de activaciones generadas, se agregaron tres mapas de activación al diseño con filtros de 1×1 en serie con capas pooling, reduciendo las computaciones numéricas de 854 a 358 millones.
- Se incluyeron un par de capas de salida extra en el medio de la red, para amplificar la señal de error propagándose en la porción inicial de la red, pues preocupaba a los autores que el gran número de capas hiciera que la señal inicial se desvaneciera.

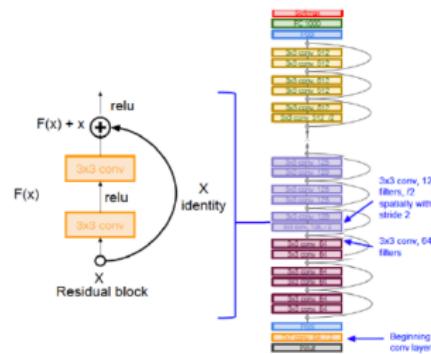


Deep Learning - Arquitecturas

- *ResNets (2015)*: Las Residual Neural Networks (ResNet), introducidas por He et al. [He et al., 2015b], comenzaron con la era de las *ultra-deep CNN* (152 capas). Innovaciones parten de la siguiente observación: número de capas limitado pues la optimización se vuelve más y más difusa, dadas las multiplicaciones con los pesos y las activaciones en el gran número de capas; resulta en que los pesos en las capas iniciales no se modifican en forma óptima durante back-propagation (comprobado empíricamente).

Solución a problema de capas:

- Autores introdujeron el *bypass link*.
- En paso forward de Backprop, el *bypass link* permite que la señal de entrada se salte algunas capas convolucionales y luego se suma a la salida de esas capas: El bypass permite que la activación desde la primera capa Conv se propague al frente de la red, mientras se modifica por algún *delta*(o *residuo*) que proviene de cada capa saltada.

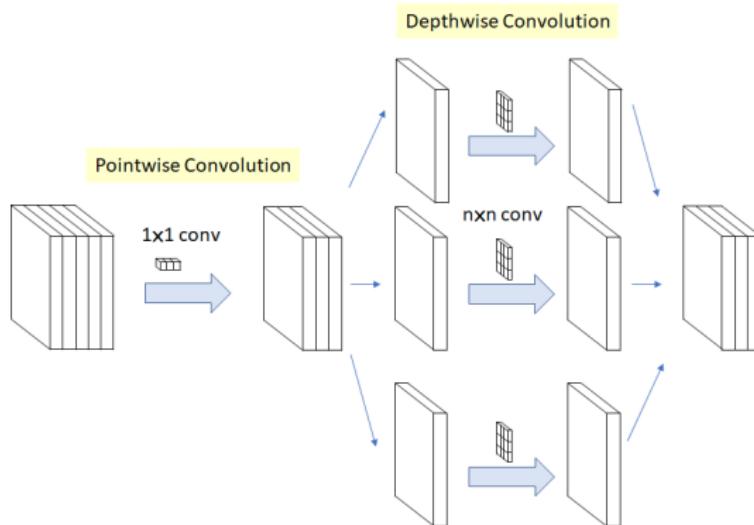


Deep Learning - Arquitecturas

- *ResNets (2015)*:
- En paso backward de Backprop, el bypass permite que la información de error de gradientede la capa de salida, se propague sin modificaciones por las capas intermedias, hasta la primera capa convolucional. Esto permite a las ResNets ser ultra-Deep sin sacrificar rendimiento.
- Otros aspectos:
 - Uso extensivo de *Batch Normalization*, implementaddo al final de cada capa.
 - *Learning rate* relativamente grande de 0.1, divido por 10 cuando el error de validación se estanca durante el entrenamiento. El *learning rate* grande es posible gracias a *Batch Normalization*.
 - No usa Dropout. En teoría Dropout no sería necesario, gracias a *Batch Normalization*.
 - No usa capas densas al final de la red.

Deep Learning - Arquitecturas

- *Xception* (2017): Desarrollado por Google [Chollet, 2016], significa Extreme version of Inception.
- Usa una convolución depthwise separable modificada: usa convolución *pointwise* (1×1) seguida por otras *depthwise*, motivada por el módulo *Inception* de *Inception-v3*.
- En la convolución depthwise separable modificada no hay capas ReLU intermedias.



Deep Learning - Convoluciones Separables

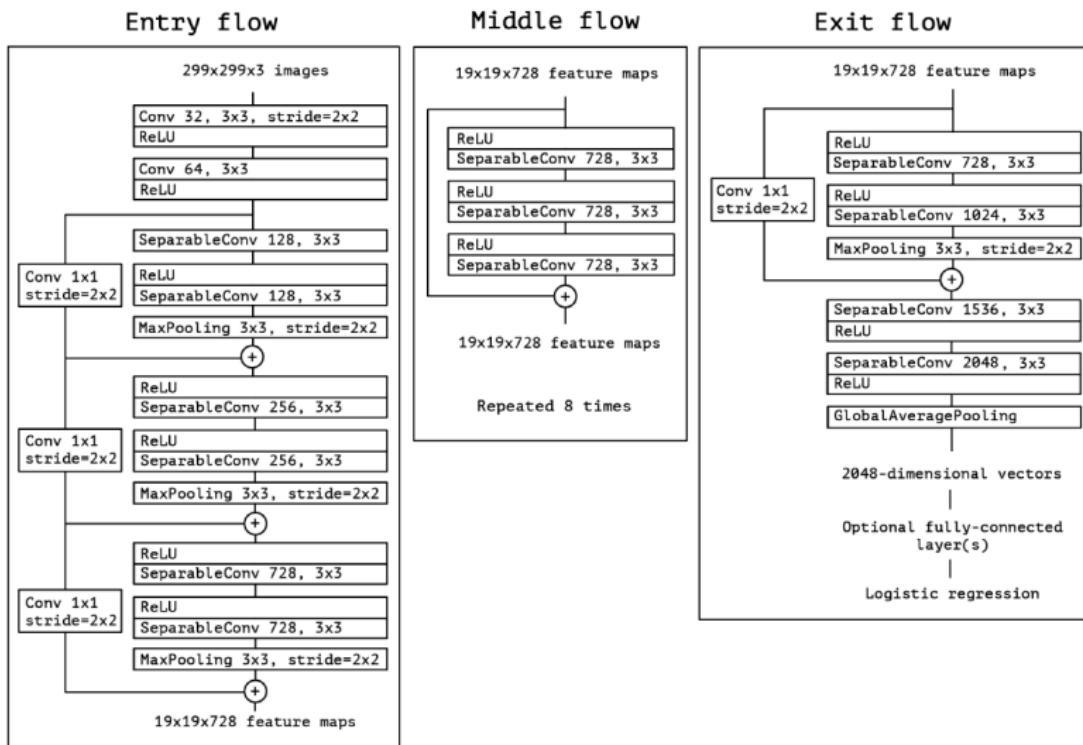
- Una convolución separable separa un kernel en dos. Ejemplo con sobel:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times [-1 \quad 0 \quad 1]$$

- Ejemplo: Imagen de entrada de $12 \times 12 \times 3$. En convolución original número de multiplicaciones para 256 kernels de $5 \times 5 \times 3$ que se mueven 8×8 veces ($256 \times 3 \times 5 \times 5 \times 8 \times 8 = 1,228,800$ multiplicaciones). En convolución depthwise, 3 kernels $5 \times 5 \times 1$ $5 \times 5 \times 1$ que se mueven 8×8 veces ($3 \times 5 \times 5 \times 8 \times 8 = 4,800$ multiplicaciones) y luego 256 kernels de $1 \times 1 \times 3$ que se mueven 8×8 veces ($256 \times 1 \times 1 \times 3 \times 8 \times 8 = 49,152$ multiplicaciones); sumando ambos son 53,952 multiplicaciones.
- Con menos cómputos la red es capaz de procesar más en un tiempo más corto.
- La desventaja es que reduce el número de parámetros en la convolución, lo que afecta a redes pequeñas, pues el número bajo de parámetros limitaría las capacidades del entrenamiento.

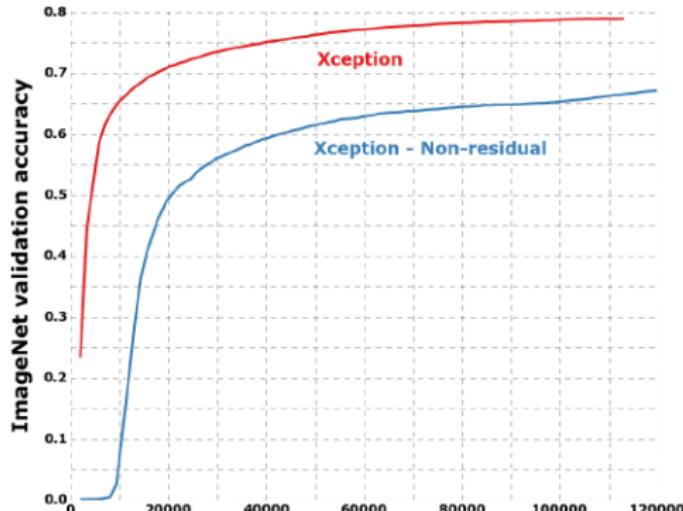
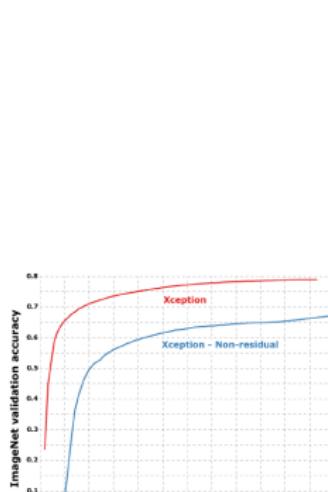
Deep Learning - Arquitecturas

- Xception (2017): Arquitectura:



Deep Learning - Arquitecturas

- *Xception* (2017): Arquitectura:
 - *SeparableConv* es la convolución depthwise separable modificada. Se tratan como módulos *Inception* repetidos en la estructura.
 - Hay también conexiones residuales, como las propuestas en ResNet.
- Rendimiento:



Keras - Implementando Transfer Learning

- Usando un modelo existente: ejemplo con *Xception* (*transfer-learning3.py*)

```
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import load_img
import matplotlib as m
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.xception import preprocess_input
from tensorflow.keras.applications.xception import decode_predictions
#Cargar modelo:
model = Xception()
#Cargar imagen desde archivo:
image = load_img('dog.jpg', target_size=(224, 224))
#Convertir imagen a numpy array
image = img_to_array(image)
images = np.zeros((1, 224, 224, 3))
images[0] = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
#Preparar imagen para modelo Xception:
images = preprocess_input(images)

#Predecir la probabilidad para las clases de salida:
yhat = model.predict(images)

#Convertir las probabilidades a etiquetas de clase:
labels = decode_predictions(yhat)

#Obtener el resultado más probable:
label1 = labels[0][0]

# Imprimir clasificación:
print('%s (%.2f%%)' % (label1[1], label1[2]*100))
```

Keras - Implementando Transfer Learning

- Reutilizar un modelo existente: ejemplo con *InceptionV3* (*transfer-learning5.py*)

```
from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.applications.inception_v3 import preprocess_input
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Lambda, Input

#1. Preparar las imágenes para el modelo Inception (MNIST)
images = preprocess_input(train_images)
#Cargar el modelo de base (sin capas densas al final):
base_model = InceptionV3(include_top=False)

#2. Construir modelo de pre-procesamiento
#2.1 Capa de entrada:
in_layer = Input(shape=(28, 28, 3))
#2.1 Capa Lambda aplica transformación a mínimo tamaño aceptado:
l_layer = Lambda(lambda image: tf.image.resize(image, (75, 75)))(in_layer)
#2.2 Construir pre-procesamiento:
in_model = tensorflow.keras.models.Model(inputs=in_layer, outputs=l_layer)

#3. Agregar capas a modelo final:
#3.1 Partir de salida de modelo de pre-procesamiento:
x = in_model.output
#3.2 Se conecta modelo Inception:
x = base_model(x)
#3.3 Agregar capa de pooling global o flatten:
x = GlobalAveragePooling2D()(x)
#3.4 Agregar capa densa:
x = Dense(1024, activation='relu')(x)
#3.5 Agregar capa de salida con 10 clases
predictions = Dense(10, activation='softmax')(x)
#3.6 Construir el modelo a entrenar
model = tensorflow.keras.models.Model(inputs=in_layer, outputs=predictions)
```

Keras - Implementando Transfer Learning

- Reutilizar un modelo existente: ejemplo con *InceptionV3* (*transfer-learning5.py*)

```
#4.1 Congelar las capas del modelo base (congela pesos):
base_model.trainable = False
#4.1 Congelar capas individualmente:
for layer in base_model.layers:
    layer.trainable = False
#4.2 Alternativo: Activar capa:
base_model.get_layer('block5_conv3').trainable = True

#5 Pasos finales:
#5.1 Compilar el modelo:
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
#5.2 Entrenar:
model.fit(train_images, train_labels, batch_size=32, epochs=10)
#5.3 Evaluar:
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('\nTest accuracy: '.format(test_acc))
#5.4 Guardar el modelo:
model.save('./transfer_model')
```

Segmentación con Deep Learning

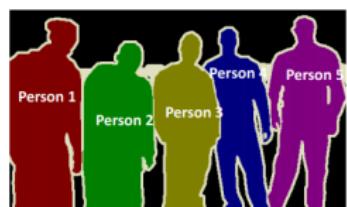
- En el contexto de imágenes, también es posible resolver problemas de localización de los objetos, o incluso obtener máscaras de segmentación y otras representaciones (e.g. modelos articulados).
- Existen modelos focalizados en obtener la segmentación semántica de un set de imágenes, que consiste en obtener directamente una imagen de segmentación desde una imagen de entrada. El modelo más conocido es *U-Net*.



Object Detection



Semantic Segmentation



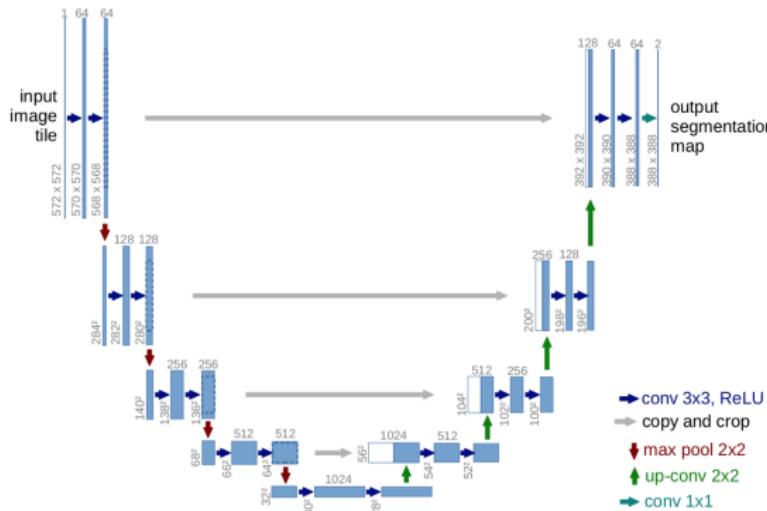
Instance Segmentation

Segmentación con Deep Learning

- Aquellos modelos que trabajan en clasificar y a su vez localizar los objetos, ya sea mediante cajas englobantes (bounding boxes) o segmentación, son los conocidos como la familia de *Region-based CNN* o *R-CNN*.
- El modelo precursor es R-CNN [Girshick et al., 2013], luego Fast-RCNN [Girshick, 2015], luego YOLO [Redmon et al., 2015], y finalmente Faster-RCNN [Ren et al., 2015], todos capaces de clasificar y localizar mediante bounding boxes en una imagen a los objetos presentes en la escena, para los cuales fueron entrenados.
- Mask-RCNN [He et al., 2017] además agrega la posibilidad de generar una máscara de segmentación.

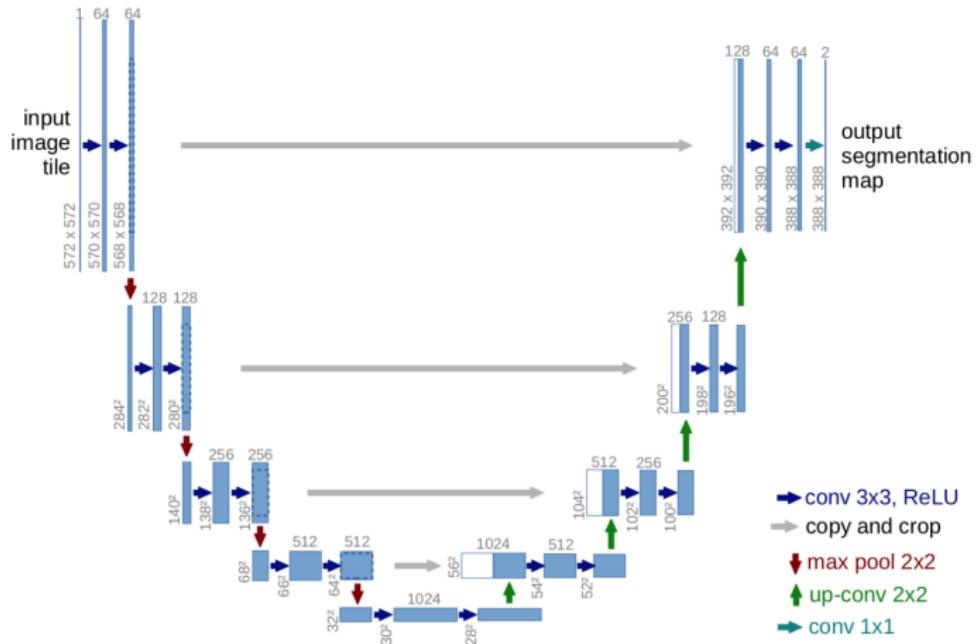
U-Net

- U-Net [Ronneberger et al., 2015] evolucionó de las CNN tradicionales, siendo inicialmente diseñado y aplicado en 2015 para procesar imágenes biomédicas.
- Consiste en una imagen de entrada procesada por la red para obtener una máscara de segmentación de igual tamaño que la imagen de entrada, que permite reconocer áreas de interés (anormalidad en el caso médico): clasificación por cada pixel.



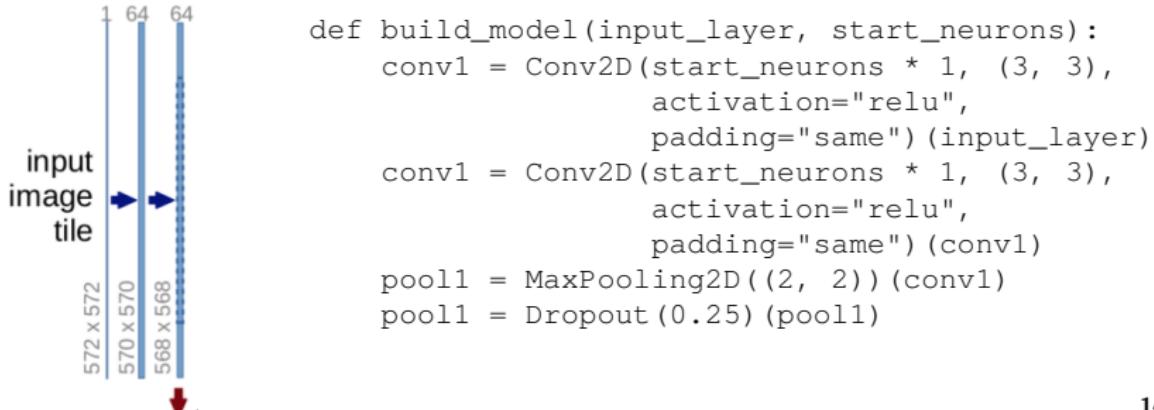
U-Net

- U-Net se estructura simétrica en dos fases (forma de U):
 - Fase de Contracción: fase convolucional.
 - Fase Expansiva: similar a un upsampling.



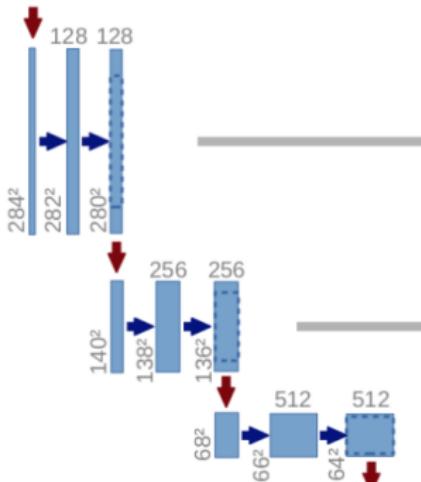
U-Net - Fase de Contracción

- En la fase de Contracción, cada subfase consiste en:
 $\text{convolucion1} \rightarrow \text{convolucion2} \rightarrow \text{max_pooling} \rightarrow \text{dropout}(\text{opcional})$
- En imagen se muestra parte del modelo original. Cada proceso consiste en 2 capas convolucionales (canales de 1 a 64).
- La flecha roja es el proceso de max pooling que reduce a la mitad el tamaño de la imagen.
- En modelo original antes reduce de 572×572 a 568×568 (por padding), pero la implementación usa `padding="same"`):
<https://www.kaggle.com/phoenigs/u-net-dropout-augmentation-stratification>



U-Net - Fase de Contracción

- El proceso se repite 3 veces más:



```
conv2 = Conv2D(start_neurons * 2, (3, 3),  
               activation="relu",  
               padding="same") (pool1)  
conv2 = Conv2D(start_neurons * 2, (3, 3),  
               activation="relu",  
               padding="same") (conv2)  
pool2 = MaxPooling2D((2, 2))(conv2)  
pool2 = Dropout(0.5)(pool2)  
  
conv3 = Conv2D(start_neurons * 4, (3, 3),  
               activation="relu",  
               padding="same") (pool2)  
conv3 = Conv2D(start_neurons * 4, (3, 3),  
               activation="relu",  
               padding="same") (conv3)  
pool3 = MaxPooling2D((2, 2))(conv3)  
pool3 = Dropout(0.5)(pool3)  
  
conv4 = Conv2D(start_neurons * 8, (3, 3),  
               activation="relu",  
               padding="same") (pool3)  
conv4 = Conv2D(start_neurons * 8, (3, 3),  
               activation="relu",  
               padding="same") (conv4)  
pool4 = MaxPooling2D((2, 2))(conv4)  
pool4 = Dropout(0.5)(pool4)
```

U-Net - Fase de Contracción

- Cuando alcanza el fondo (parte inferior de la U) se realizan dos convoluciones más.
- La imagen se ha convertido a dimensiones 28x28x1024.



```
convm = Conv2D(start_neurons * 16, (3, 3), activation="relu",
                padding="same") (pool4)
convm = Conv2D(start_neurons * 16, (3, 3), activation="relu",
                padding="same") (convm)
```

U-Net - Fase Expansiva

- En la fase Expansiva se va recuperando el tamaño de la imagen original, para obtener una imagen de segmentación. Sub-fases siguen estructura: conv2D_traspuesta → concatenar → convolucion1 → convolucion2
- La convolución transpuesta es una técnica de upsampling. Es app. una capa que combina capas *UpSampling2D* y *Conv2D*.

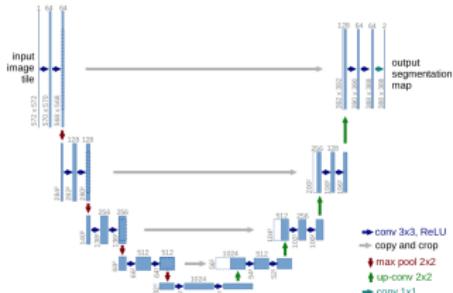


- Luego de convolución traspuesta imagen crece de 28x28x1024 a 56x56x512, y se concatena con la imagen correspondiente de la fase de contracción (56x56x1024): combina información de capas previas para obtener predicción más precisa. Luego, se agregan 2 convoluciones.

```
deconv4 = Conv2DTranspose(start_neurons * 8, (3, 3),
                         strides=(2, 2), padding="same") (conv4)
uconv4 = concatenate([deconv4, conv4])
uconv4 = Dropout(0.5) (uconv4)
uconv4 = Conv2D(start_neurons * 8, (3, 3),
               activation="relu", padding="same") (uconv4)
uconv4 = Conv2D(start_neurons * 8, (3, 3),
               activation="relu", padding="same") (uconv4)
```

U-Net - Fase Expansiva

- El proceso de expansión se repite 3 veces más:



```

deconv3 = Conv2DTranspose(start_neurons * 4, (3, 3),
                         strides=(2, 2), padding="same") (uconv4)
uconv3 = concatenate([deconv3, conv3])
uconv3 = Dropout(0.5) (uconv3)
uconv3 = Conv2D(start_neurons * 4, (3, 3),
               activation="relu", padding="same") (uconv3)
uconv3 = Conv2D(start_neurons * 4, (3, 3),
               activation="relu", padding="same") (uconv3)

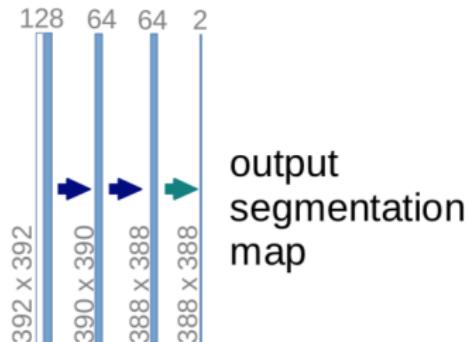
deconv2 = Conv2DTranspose(start_neurons * 2, (3, 3),
                         strides=(2, 2), padding="same") (uconv3)
uconv2 = concatenate([deconv2, conv2])
uconv2 = Dropout(0.5) (uconv2)
uconv2 = Conv2D(start_neurons * 2, (3, 3),
               activation="relu", padding="same") (uconv2)
uconv2 = Conv2D(start_neurons * 2, (3, 3),
               activation="relu", padding="same") (uconv2)

deconv1 = Conv2DTranspose(start_neurons * 1, (3, 3),
                         strides=(2, 2), padding="same") (uconv2)
uconv1 = concatenate([deconv1, conv1])
uconv1 = Dropout(0.5) (uconv1)
uconv1 = Conv2D(start_neurons * 1, (3, 3),
               activation="relu", padding="same") (uconv1)
uconv1 = Conv2D(start_neurons * 1, (3, 3),
               activation="relu", padding="same") (uconv1)

```

U-Net - Fase Expansiva

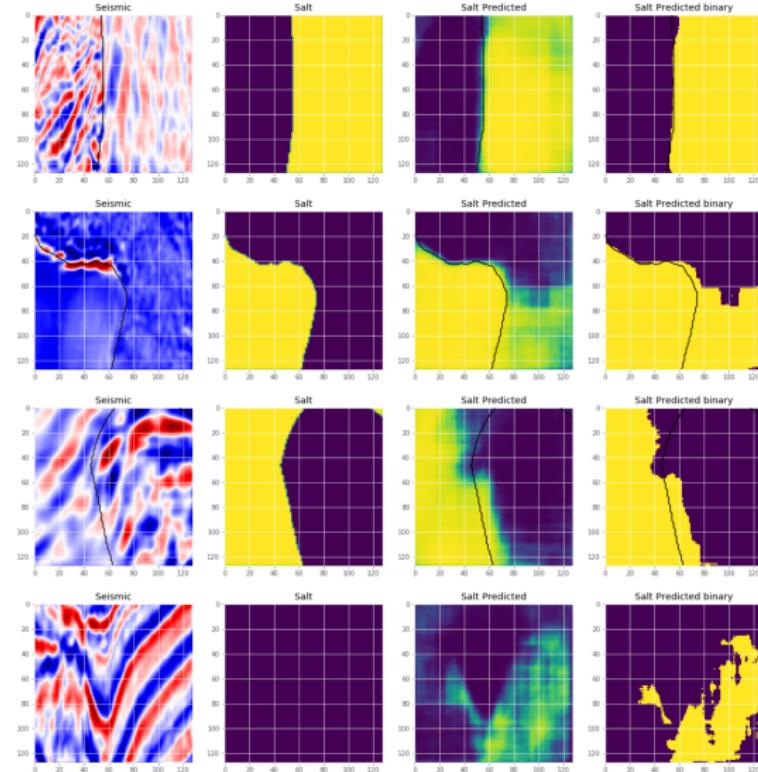
- Última paso es reestructurar la imagen para satisfacer los requerimientos de predicción.
- La última capa es una convolución con un filtro de 1x1 y activación sigmoidal.
- Nótese que no se usan capas densas en toda la red.



```
output_layer = Conv2D(1, (1,1), padding="same",
                      activation="sigmoid") (uconv1)
```

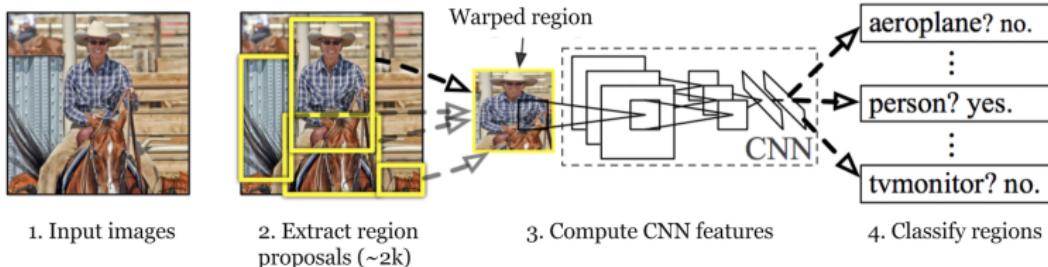
U-Net - Ejemplo de aplicación

- Imágenes sísmicas, para predecir si existen depósitos de sal en una región o no.



R-CNN

- La idea principal de *Region-based Convolutional Neural Networks* (R-CNN) [Girshick et al., 2013] se compone de tres pasos:
 - 1 Usar *selective search* para identificar un número manejoable de candidatos a región en la forma de *bounding boxes* (*region of interest* o *RoI*).
 - 2 Extraer características mediante CNN desde cada región independientemente para clasificación.
 - 3 Clasificar con SVM binarios, en base a características desde CNN.



1. Input images

2. Extract region proposals (~2k)

3. Compute CNN features

4. Classify regions

R-CNN - Selective Search

- *Selective search* algoritmo para regiones candidatas que potencialmente contienen objetos.
- Funcionamiento:
 - ① Inicialización de regiones R con segmentación de Felzenszwalb.
 - ② Iniciar conjunto de similaridades S entre todos los pares de regiones.
 - ③ Usar algoritmo *greedy* para agrupar las regiones iterativamente:
 - ① Agrupar las dos regiones más similares.
 - ② Eliminar de S todas las similares asociadas a las dos regiones previas.
 - ③ Ingresar a S nuevas *similaridades* entre región resultante y vecinos.
 - ④ Agregar región resultante en R .
 - ④ Paso 2 se repite hasta que toda imagen se vuelve una sola.
 - ⑤ Finalmente, generar bounding boxes desde R .
- Proponen medida integrada de similaridad: color, textura (SIFT), tamaño(pequeñas se unen antes), forma(región llena agujero de otra).
- Se pueden generar múltiples conjuntos de regiones candidatas: ajustando k de segmentación, cambiando espacio de color, tomando distintas combinaciones de las medidas de similaridad.

R-CNN

- Funcionamiento de R-CNN:

- Pre-entrenar la CNN en clasificación (e.g VGG, ResNet o Xception con ImageNet) para N clases.
- Proponer regiones independientes de clase con *selective search* (ap. 2k candidatos por imagen).
- Regiones candidatas se ajustan a tamaño fijo requerido por CNN (warping).
- Continuar ajustando la CNN con cada región candidata para $N + 1$ clases (suma clase *background*); usar *learning rate* pequeño y repetir casos positivos en el batch, dado que la mayoría de regiones candidatas será *background*.
- Para cada región candidata, se infiere con CNN un vector de características, que se usa para entrenar un SVM binario para cada clase.
- Las muestras positivas son regiones candidatas con métrica IoU (intersección sobre unión) ≥ 0.3 , y las demás son negativas.
- Para reducir error de localización, se entrena un modelo de regresión (*Bounding Box Regression*).

R-CNN - Bounding Box Regression

- Dadas coordenadas de una *bounding box* predicha $p = (p_x, p_y, p_w, p_h)$ (coordenada central, ancho, alto) y sus correspondientes coordenadas de *bounding box* de *ground-truth* $g = (g_x, g_y, g_w, g_h)$.
- Modelo de regresión:

$$\hat{g}_x = p_w d_x(p) + p_x$$

$$\hat{g}_y = p_h d_y(p) + p_y$$

$$\hat{g}_w = p_w e^{d_w(p)}$$

$$\hat{g}_h = p_h e^{d_h(p)}$$

d corresponden a las transformaciones a aprender.

- Salidas conocidas:

$$t_x = (g_x - p_x)/p_w$$

$$t_y = (g_y - p_y)/p_h$$

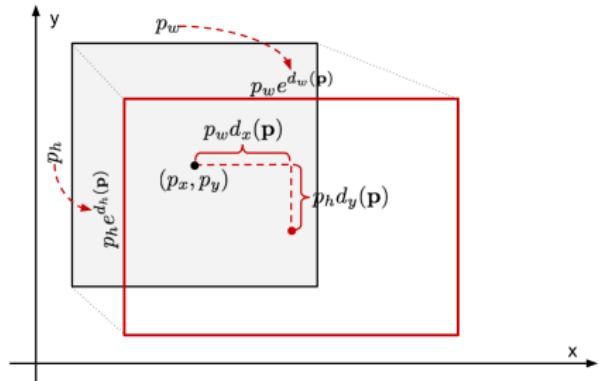
$$t_w = \log(g_w/p_w)$$

$$t_h = \log(g_h/p_h)$$

- Se entrena modelo de regresión, con *loss*:

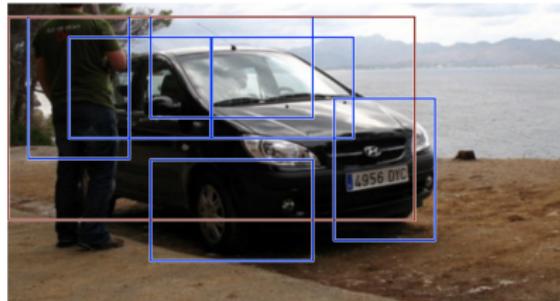
$$L = \sum_{i \in \{x, y, w, h\}} (t_i - d_i(p))^2 + \lambda ||w||^2$$

- Sólo para $IoU \geq 0.6$.



R-CNN - Non-Maximum Suppression

- Es probable que el modelo encuentre múltiples *bounding boxes* para el mismo objeto.
- *Non-maximum suppression:*
 - ① Ordenar *bounding boxes* por *score* de clasificación.
 - ② Descartar aquellas con baja probabilidad.
 - ③ Iterativamente seleccionar la de *score* más alto y eliminar las que tengan un IoU mayor a un umbral.



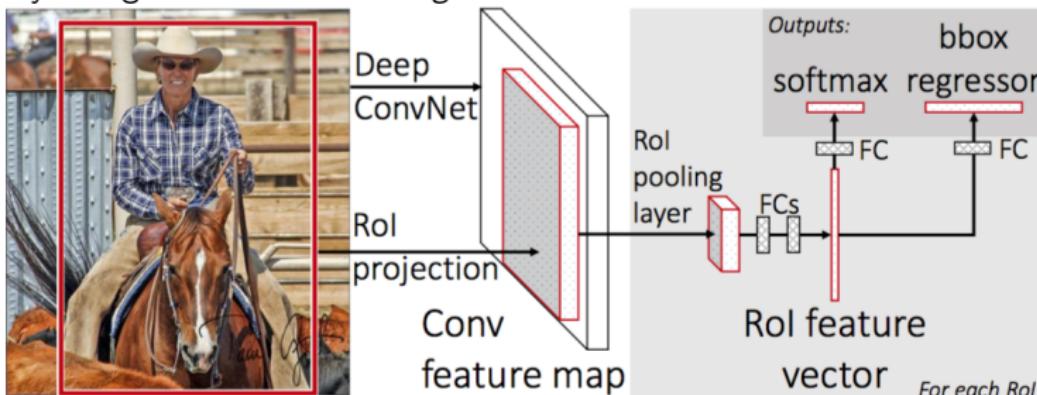
Before non-max suppression



After non-max suppression

Fast R-CNN

- R-CNN es lento y costoso: 2000 regiones candidatas desde *selective search* por N imágenes; generar vector candidato desde CNN por cada una; tres modelos independientes (CNN de características, SVM para clasificación y modelo de regresión para ajustar *bounding boxes*).
- Para acelerar R-CNN, se propone Fast-RCNN[Girshick, 2015] que mejora el entrenamiento unificando los modelos: se agregan una capa (RoI Pooling) que conecta la salida de las características extraídas por CNN desde las regiones candidatas, con las capas encargadas de la clasificación y la regresión de la *bounding box*.



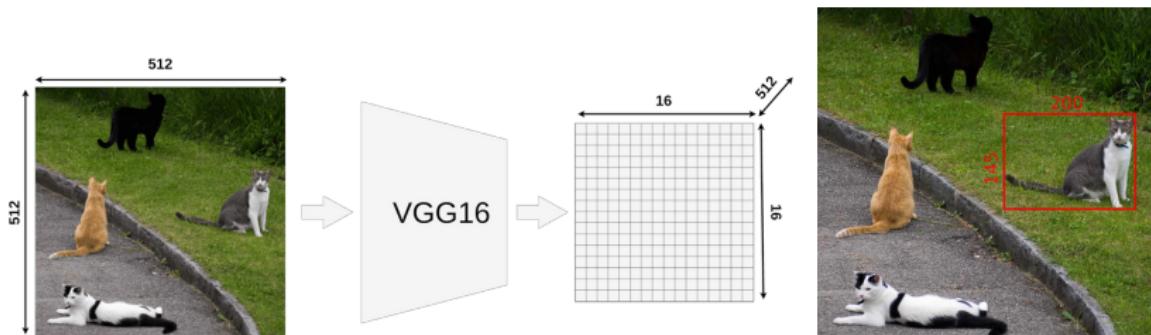
Fast R-CNN

Funcionamiento de Fast R-CNN:

- Pre-entrenar la CNN en clasificación.
- Proponer regiones con *selective search* (aprox. 2k candidatos por imagen).
- Alterar la CNN pre-entrenada:
 - Reemplazar la última capa max pooling de modelo pre-entrenado con *Roi pooling layer*: obtiene características de tamaño fijo para regiones candidatas de distinto tamaño.
 - Reemplazar la última capa densa y la capa softmax (N clases) con otra capa densa y capa softmax para $N + 1$ clases.
 - También, agregar para regresión de la *bounding box*, capa densa y capa de salida para regresión.
- Las muestras positivas son regiones candidatas con métrica IoU (intersección sobre unión) ≥ 0.3 , y las demás son negativas.

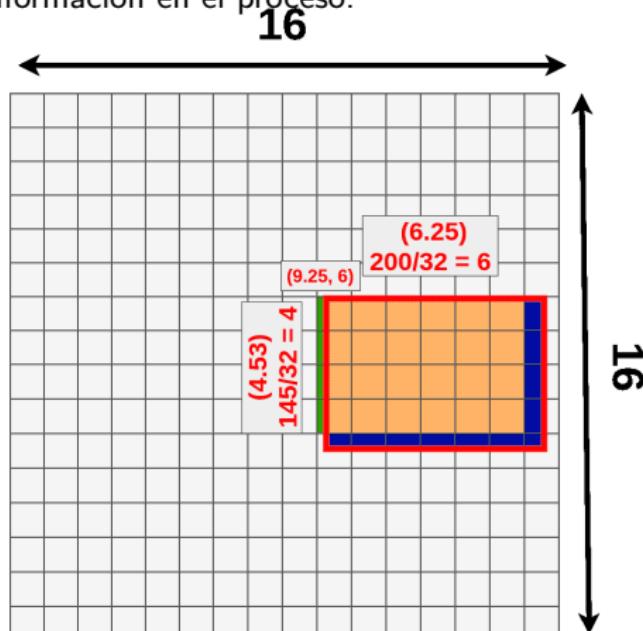
Fast R-CNN - Conexión a capa RoI Pooling

- Cada RoI tiene sus propias coordenadas y tamaño.
- Es necesario mapearlas a la salida del modelo pre-entrenado (En ejemplo, $16 \times 16 \times 512$; $512/16=32$ veces más pequeña).
- Como ejemplo se toma una RoI con tamaño 145×200 y esquina superior izquierda (192, 296). No divide exactamente por 32: $w = 200/32 = 6.25$; $h = 145/32 = 4.53$; $x = 296/32 = 9.25$; $y = 192/32 = 6$.



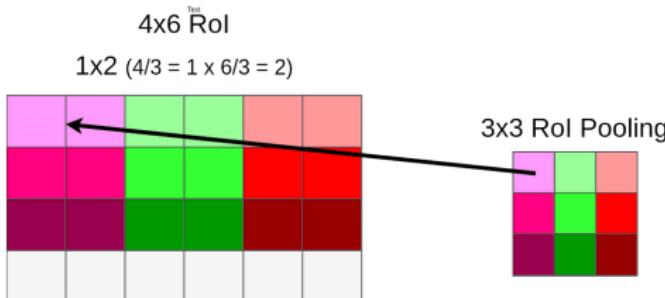
Fast R-CNN - Conexión a capa Rol Pooling

- Es necesario entonces *cuantizar*: llevar valores a enteros.
- La cuantización redondea hacia abajo cada dimensión: $9.25 \rightarrow 9$, $4.53 \rightarrow 4$, ...
- Se pierde información en el proceso.



Fast R-CNN - Capa Rol Pooling

- Con el Rol mapeado al mapa de características, se aplica pooling: es necesario, porque después viene una capa densa de tamaño fijo.
- Como los Rols tiene diferentes tamaños se debe aplicar pooling para llevarlos a un mismo tamaño ($3 \times 3 \times 512$ en el ejemplo).
- El Rol mapeado es de $4 \times 6 \times 512$ y 4 no es divisible por 3.
- Se aplica nuevamente cuantización: Para altura 4 se aplicará pooling de tamaño $4/3 = 1.33 \approx 1$ (pérdida de información) y para ancho 6 se aplicará pooling de tamaño $6/3 = 2$.



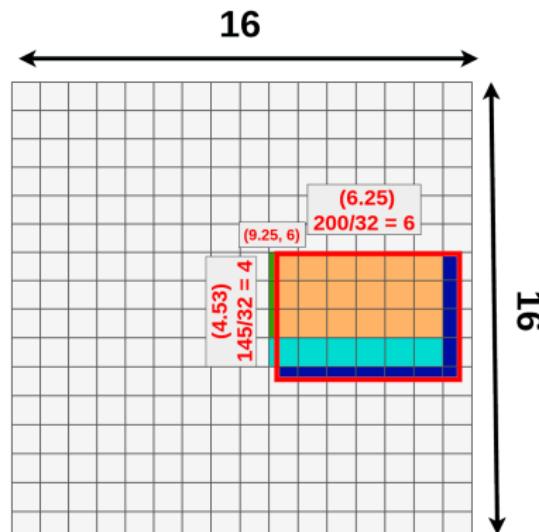
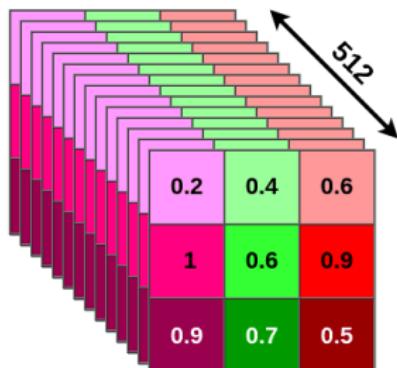
4x6 Rol (Lost Data)

0.1	0.2	0.3	0.4	0.5	0.6
1	0.7	0.2	0.6	0.1	0.9
0.9	0.8	0.7	0.3	0.5	0.2
0.2	0.5	1	0.7	0.1	0.1

Fast R-CNN - Capa Rol Pooling

- Se aplica mismo proceso a cada capa de activación.
- Esto se realiza para cada región candidata.
- Mucha información perdida en el proceso.

3x3 RoI Pooling (full size)



Fast R-CNN - Función Loss

- El modelo se optimiza para *loss* que combina dos tareas (clasificación + localización). Se define:
 - u : etiqueta de clase de entrenamiento, con $u \in 0, 1, \dots, K$ (convención: clase 0 para *background*).
 - p : Probabilidad (por Rol) para $K + 1$ clases, con $p = (p_0, \dots, p_K)$, obtenida con un softmax sobre $K + 1$ salidas de una capa densa.
 - v : *bounding box* del ejemplo, con $v = (v_x, v_y, v - w, v_h)$.
 - t^u : Corrección de la *bounding box* predicha, con $t^u = (t^u x, t^u y, t^u w, t^u h)$.

- Función *loss* suma costo de clasificación y de predicción de la *bounding box*: $L = L_{cls} + L_{box}$. Se define función indicadora:

$$1[u \geq 1] = \begin{cases} 1 & \text{si } u \geq 1 \\ 0 & \text{sino} \end{cases}$$

- Función *loss* completa:

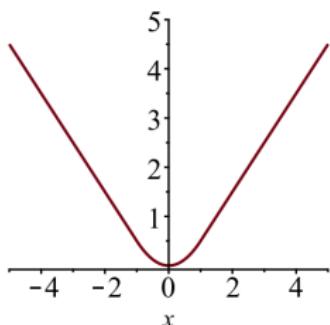
$$L(p, u, t^u, v) = L_{cls}(p, u) + 1[u \geq 1]L_{box}(t^u, v)$$

$$L_{cls}(p, u) = -\log(p_u)$$

$$L_{box}(t^u, v) = \sum_{i \in \{x, y, w, h\}} L_1^{smooth}(t_i^u - v_i), \text{ con}$$

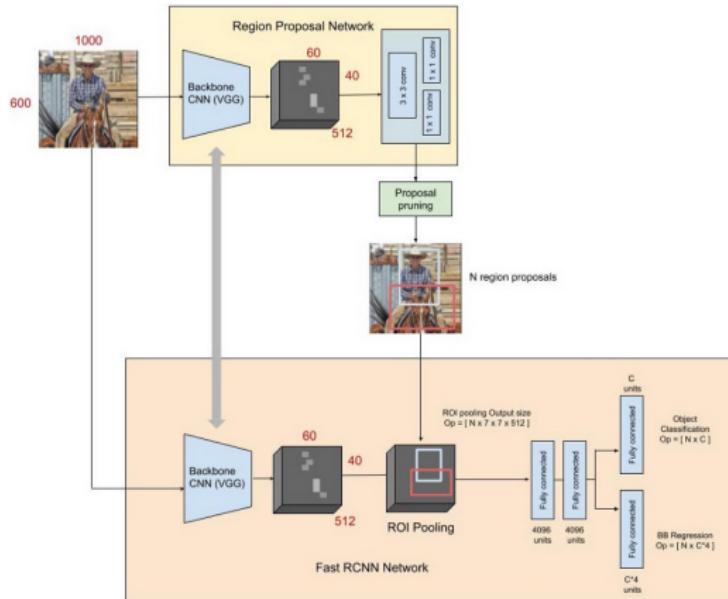
$$L_1^{smooth}(x) = \begin{cases} 0.5x^2 & \text{si } |x| \leq 1 \\ |x| - 0.5 & \text{sino} \end{cases}$$

- Función L_1^{smooth} menos sensible a outliers.



Faster R-CNN

- Fast R-CNN mucho más rápido en entrenamiento y testing, pero mejora limitada porque regiones candidatas se generan por algoritmo costoso.
- Faster R-CNN [Ren et al., 2015] integra la generación de regiones candidatas con la CNN: un solo modelo unificado, con una RPN (region proposal network) y Fast R-CNN con capas convolucionales compartidas.



Faster R-CNN

Funcionamiento de Faster R-CNN:

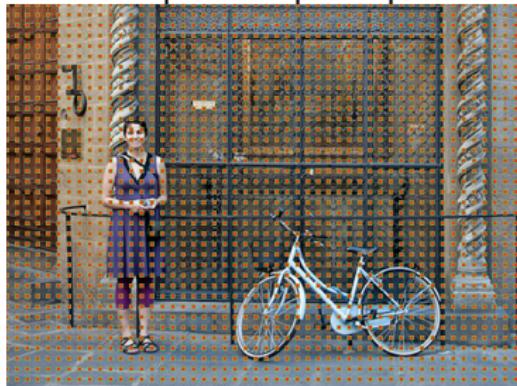
- Pre-entrenar la CNN en clasificación.
- Entrenamiento alternado de 4 pasos (compartir pesos):
 - ① Ajustar la RPN para obtener regiones candidatas, inicializadas por el clasificador pre-entrenado. Los ejemplos positivos con $IoU > 0.7$ y negativos con $IoU < 0.3$:
 - Deslizar pequeña ventana espacial sobre el mapa de características de convolución en la imagen entera.
 - Al centro de cada ventana deslizante, predecir múltiples regiones en varias escalas y *ratios* simultáneamente. *anchor*: combinación (centro de ventana deslizante, escala, ratio).
 - ② Detector Fast R-CNN entrenado independiente también. CNN base es entonces ajustado para clasificación. Los pesos de RPN se congelan y las regiones candidatas de RPN se usan para entrenar Faster R-CNN completo.
 - ③ RPN inicializado con pesos obtenidos del paso anterior (parte CNN) y ajustado para tarea de regiones candidatas; se congelan pesos en capas CNN; sólo capas propias de RPN son ajustadas: RPN final.
 - ④ Con el nuevo RPN, se ajusta el detector Fast R-CNN. Se congelan los pesos de CNN y sólo ajusta capas finales.

Faster R-CNN - Anchors

- El problema del tamaño variable de los ejemplos es resuelto por la RPN usando *anchors*: *bounding boxes* de referencia de tamaño fijo que se ubican de forma uniforme sobre la imagen original. El problema entonces se separa en dos partes: primero, verificar si el *anchor* tiene un objeto relevante; segundo, ajustar el *anchor* para calzar mejor con el objeto.
- Después de tener una lista de los posibles objetos relevantes y sus ubicaciones en la imagen original, se usan las características extraídas por la CNN y las *bounding boxes* con objetos relevantes y se aplica *RoI Pooling* para conectar con Fast-RCNN.

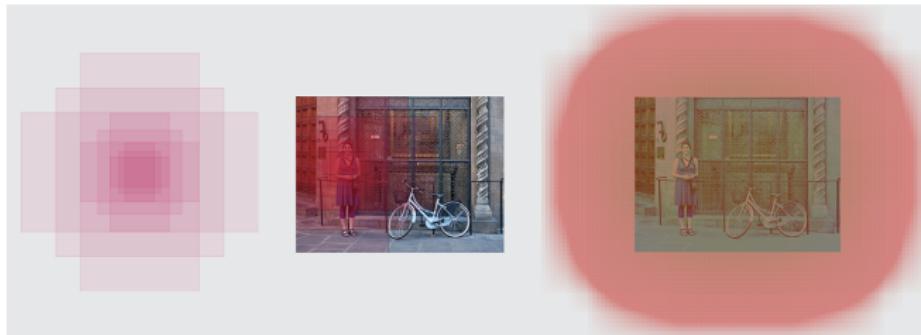
Faster R-CNN - Anchors

- Dado que se está trabajando con el mapa de características convolucional de tamaño $conv_H \times conv_W \times conv_C$, se crea un conjunto de *anchors* para cada punto en $conv_H \times conv_W$ (que implícitamente hacen referencia a la imagen original, de forma proporcional).
- Si la imagen es de $h \times w$, el mapa de características será de tamaño $h/r \times w/r$, con r llamado *subsampling ratio*. Como se define una posición de *anchor* por cada posición en este mapa, la imagen final terminará con anchors separadas por r píxeles.



Faster R-CNN - Anchors

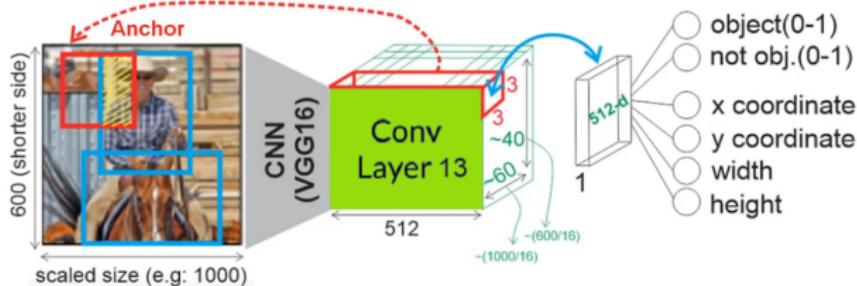
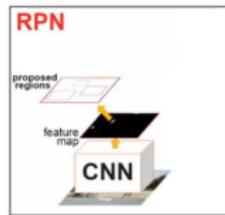
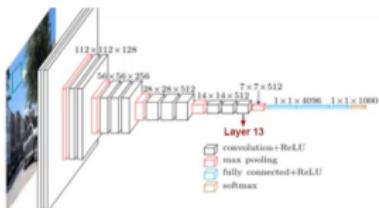
- Para escoger el conjunto de *anchors* usualmente se define un conjunto de tamaños (e.g. 64px, 128px, 256px) y otro de *ratios* entre ancho y alto de las cajas.
- Luego, se usan todas las combinaciones posibles entre tamaños y *ratios* para generarlas.



Faster R-CNN - Region Proposal Network

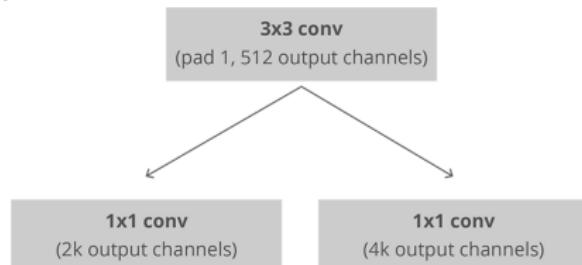
- Red independiente para proponer regiones para la red Faster-RCNN.
- La RPN toma todos los *anchors* y genera un conjunto de propuestas para objetos.

$$\text{IoU} = \frac{A \cap \text{Gt}}{A \cup \text{Gt}} \begin{cases} > 0.7 = \text{object} \\ < 0.3 = \text{not object} \end{cases}$$



Faster R-CNN - Region Proposal Network

- Lo hace optimizando dos salidas:
 - La probabilidad que el *anchor* sea un objeto (*objectness score*). A la RPN no le interesa de qué clase es el objeto (sólo si es objeto o *background*). El *score* entonces se usa para filtrar malas predicciones para la segunda fase.
 - La regresión de la *bounding box* para ajustar los *anchors* para calzar mejor con el objeto que se está prediciendo.
- La RPN usa el mapa de características convolucional como entrada. Aplica primero un kernel de 3×3 con 512 canales y luego dos capas convolucionales paralelas de kernel 1×1 , con número de canales dependiente de número de *anchors* por punto (valor k). Una con salida $2k$ (es objeto o no) y otra con salida $4k$ (regresor para corrección de *bounding box*).



Faster R-CNN - Función Loss

- Tanto Fast-RCNN como la RPN usan *loss* similar a Fast-RCNN. Se define:
 - p_i : probabilidad que *anchor* i sea un objeto.
 - p_i^* : etiqueta (binaria) de ground-truth, de si el *anchor* i es un objeto.
 - t_i : Las cuatro coordenadas de *anchor* parametrizadas.
 - t_i^* : Coordenadas en la ground-truth.
 - N_{cls} : Término de normalización, que corresponde a tamaño de mini-batch (~ 256 en el paper).
 - N_{box} : Término de normalización, que corresponde a número de ubicaciones del anchor (~ 2400 en el paper).
 - λ : Parámetro de balanceo, con valor ~ 10 en el paper (app. igual peso para L_{cls} y L_{box}).
- Función *loss* combina costo de clasificación y de predicción de la *bounding box*: $L = L_{cls} + L_{box}$:

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \frac{\lambda}{N_{box}} \sum_i L_{box} L_1^{smooth}(t_i, t_i^*)$$

$$L_{cls}(p, u) = -p^* \log(p) - (1 - p^*) \log(1 - p)$$

- RPN optimiza para dos clases (objeto + *background*), y la Faster-RCNN para $N + 1$ clases (N objetos + *background*).

Faster R-CNN - Entrenamiento en RPN

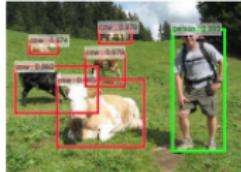
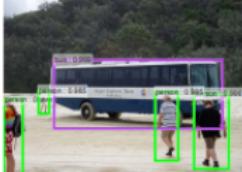
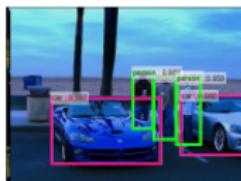
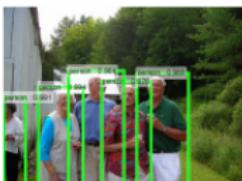
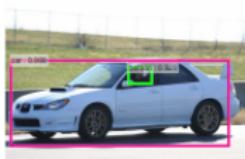
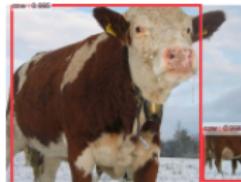
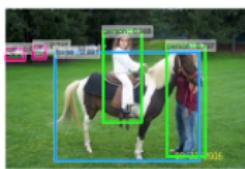
- Con mapa de características de 40×60 ubicaciones, se tienen $40 \times 60 \times 9 \sim 20000$ *anchors* en total.
- Durante entrenamiento, *anchors* que cruzan límites de imagen se descartan (no contribuyen a *loss*); deja 6000 *anchors* por imagen.
- *anchor* positiva cumpliendo ya sea: a) IoU más alto contra algún ejemplo de *ground-truth*; b) $IoU > 0.7$ contra algún ejemplo.
- *anchor* negativa si $IoU < 0.3$ para todos los ejemplos de la imagen; las demás se descartan del entrenamiento.
- Considera mini-batch proveniente de una misma imagen: 128 positivas y 128 negativas seleccionadas para formar el *batch*, agregando más negativas, si no hay suficientes positivas.
- Para los $k (= 9)$ *anchors* se tienen regresores independientes (no comparten pesos).

Faster R-CNN - Testing en RPN y Fast-RCNN

- En fase de testing, las 20000 *anchors* desde cada imagen se post-procesan:
 - Los coeficientes de regresión se usan para corregir localización de los *anchors*: *bounding boxes* precisas.
 - Se ordenan las *bounding boxes* por probabilidad de ser objeto.
 - Se aplica non-maximum suppression (NMS) con umbral 0.7: considerando orden, se deja la más probable de grupo de cajas con traslape (resulta en ~ 2000 propuestas).
 - Las *bounding boxes* que pasan límite de imagen, se recortan.
- Estas ~ 2000 son las regiones usadas para entrenar Faster R-CNN completo. Luego, en fase de testing, solo se utilizan las top N candidatas.

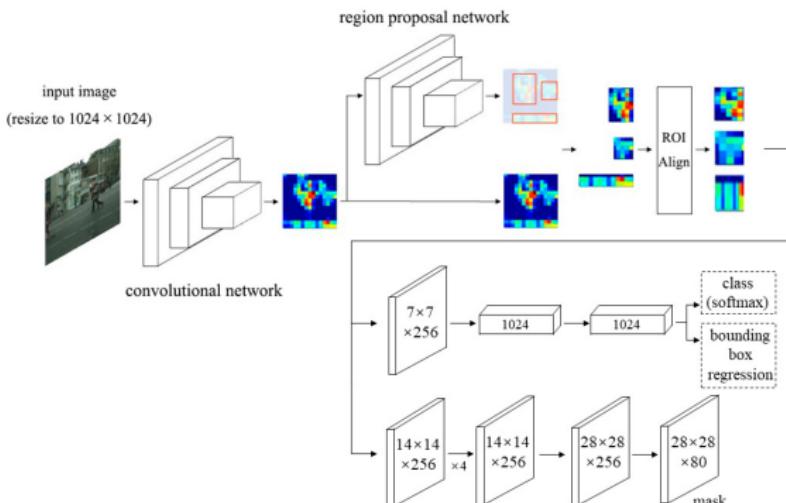
Faster R-CNN - Resultados

- En el paper se ve una ligera mejora en inferencia, en relación a predecesor.
- Lo más significativo es en tiempo de ejecución: Detección con VGG + RPN toma 198[ms] contra 1.8[s] de Selective Search.



Mask R-CNN

- Mask R-CNN [He et al., 2017] extiende Faster R-CNN a segmentación de imágenes a nivel de pixels.
- Agrega tercera red paralela de salida a clasificación y localización, para predicción de la máscara a nivel de píxeles.
- Nueva subred es una pequeña red densa aplicada a cada RoI, para predicción de la máscara de segmentación.



Mask R-CNN - RoI Align

- Dado que segmentación a nivel de pixel requiere un alineamiento mucho más fino de las *bounding boxes*, mask R-CNN mejora la capa *RoIPooling* para que la RoI se pueda mapear mejor y más precisamente a las regiones de la imagen original: nueva capa *RoIAvgPool*, que arregla el desalineamiento por la cuantización en RoI pooling.
- Usa las posiciones exactas en el mapa de características, para que las *bounding boxes* queden perfectamente alineadas.
- Usa interpolación bilineal para computar la ubicación.



Mask R-CNN - Función Loss

- Función *loss* combina clasificación, localización y máscara de segmentación: $L = L_{cls} + L_{box} + L_{mask}$
- L_{cls} y L_{box} mismos que en Faster-RCNN.
- La rama de la máscara genera una máscara de $m \times m$ píxeles para cada Rol y cada clase con K clases en total: salida total de tamaño $K \cdot m^2$.
- L_{mask} es la función *loss cross-entropy* binaria promedio, sólo incluyendo la k -ésima máscara si la región está asociada con la clase *ground-truth* k :

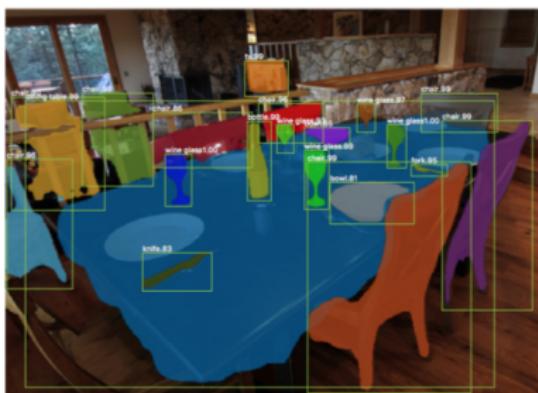
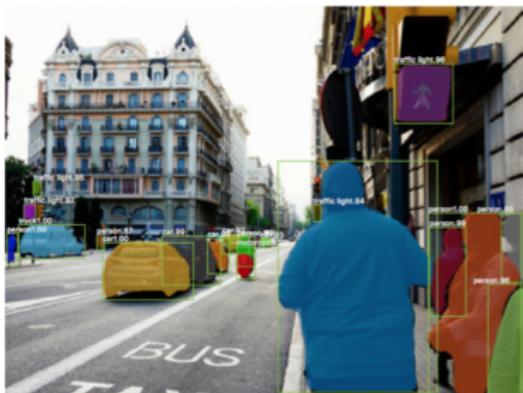
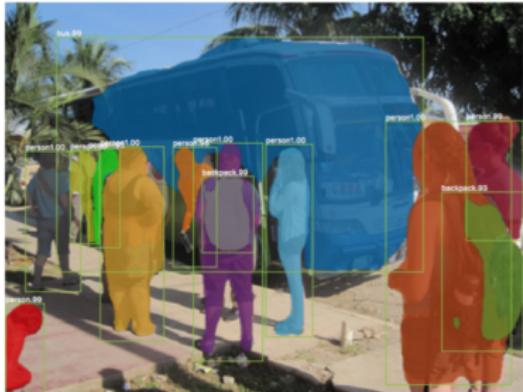
$$L_{mask} = -\frac{1}{m^2} \sum_{1 \leq i, j \leq m} [y_{ij} \log \hat{y}_{ij}^k + (1 - y_{ij}) \log(1 - \hat{y}_{ij}^k)],$$

con y_{ij} etiqueta en posición (i, j) de la máscara de ground-truth para la región de tamaño $m \times m$; \hat{y}_{ij}^k es la predicción para la misma celda en la máscara aprendida para la clase k .

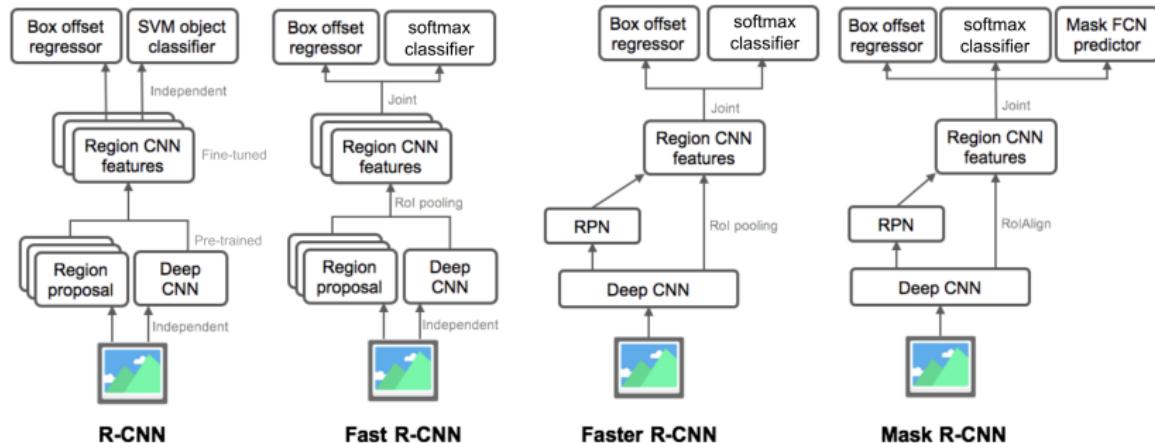
Mask R-CNN - Entrenamiento y Testing

- En **entrenamiento**, Rol positiva si $IoU \geq 0.5$ y negativo sino.
- L_{mask} sólo actualiza para casos positivos.
- Imágenes escaladas para que dimensión más corta sea de 800 píxeles.
Cada mini-batch tiene 2 imágenes por GPU y cada imagen tiene N Rols muestreados, con tasa 1 : 3 de positivos VS negativos $N = 64$ para CNN, y 512 para clasificación.
- *anchors* de RPN para 5 escalas y 3 *aspect ratios*.
- En fase de **testing**, se consideran 1000 Rols para clasificación.
- Se usa non-maximum suppression posterior a la rama de predicción de *bounding box*. Luego, se aplica la rama de la máscara, para las 100 Rols de mayor probabilidad. Esto acelera la inferencia y mejora la precisión (menos Rols, más precisas). Se usa sólo la máscara de la clase predicha por la rama de clasificación.
- Finalmente, la máscara de $m \times m$ se redimensiona al tamaño del Rol y se binariza con umbral 0.5.
- Como se computan sólo las top 100 Rols, Mask R-CNN sólo agrega un *overhead* de app. 20%, comparado con Faster R-CNN.

Mask R-CNN

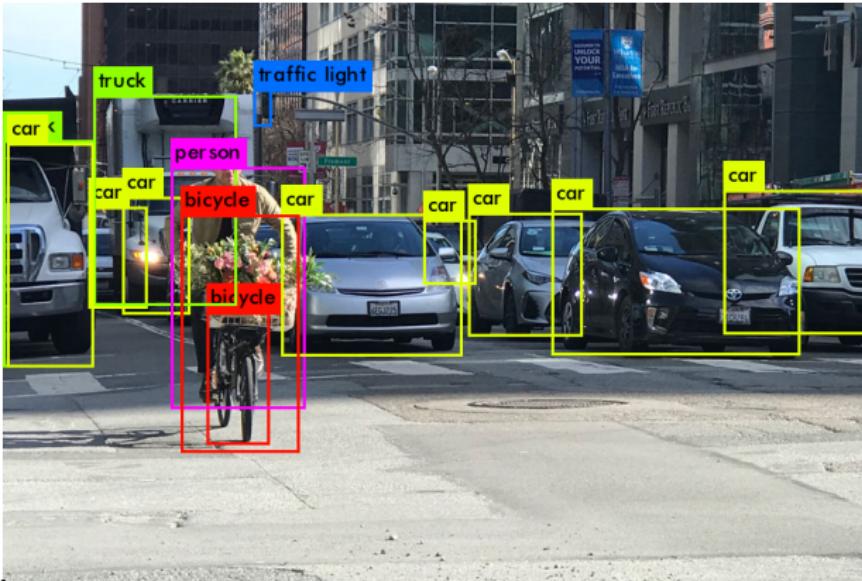


R-CNN - Resumen



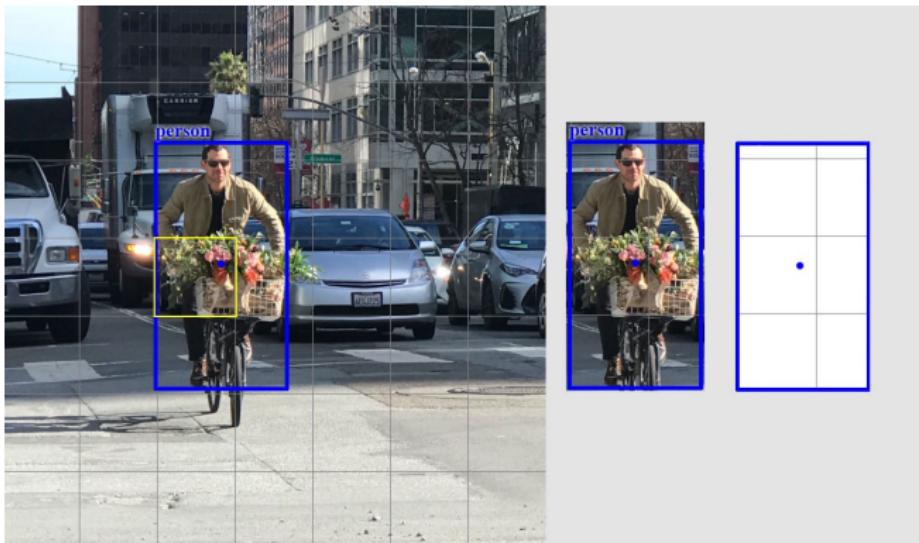
YOLO

- You only look once (YOLO) [Redmon et al., 2015] es un modelo de detección de objetos orientado a tiempo real.
- La filosofía es bastante diferente de los R-CNN, pues no considera regiones candidatas.



YOLO

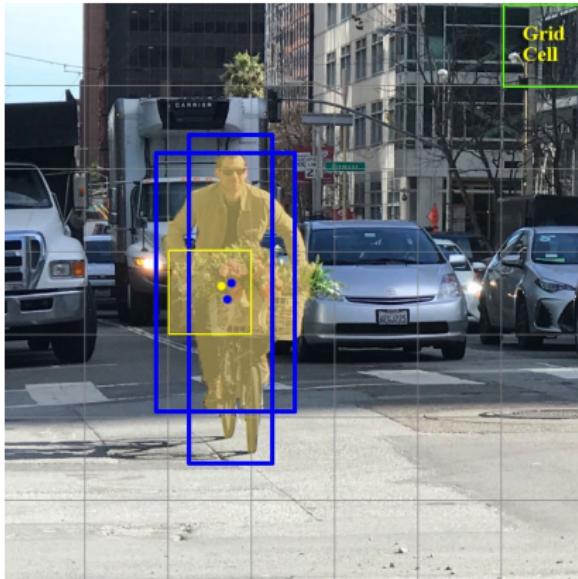
- YOLO divide la imagen de entrada en una grilla de $S \times S$: cada celda predice sólo un objeto.
- Por ejemplo, la celda amarilla trata de predecir el objeto *persona*, cuyo centro cae dentro de la celda (punto azul).



[Jonathan Hui]

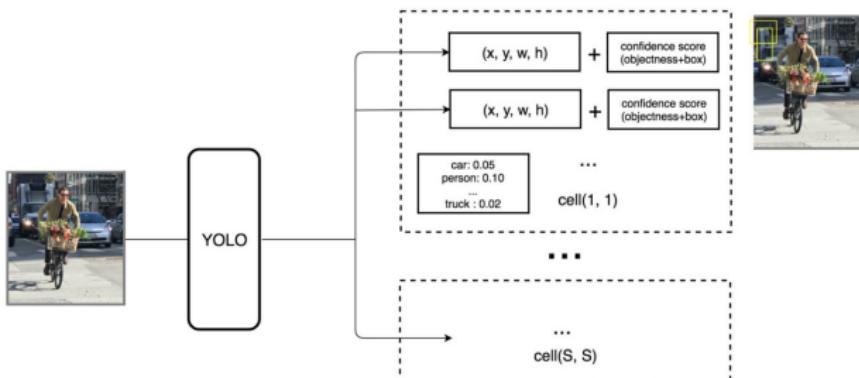
YOLO

- Cada celda predice además un número fijo de *bounding boxes*, con un puntaje de *confianza*, cada una.
- En ejemplo, la celda amarilla hace dos predicciones (cajas azules) para ubicar dónde se encuentra la persona.
- Predice también las probabilidades para C clases.



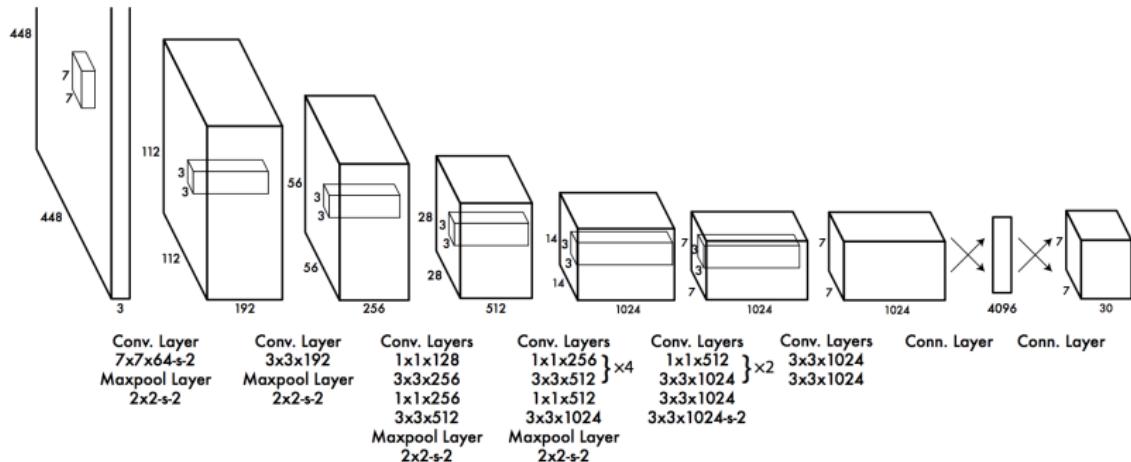
YOLO - Modelo

- La salida del modelo es entonces por celda.
- Cada *bounding box* contiene: (x, y, w, h) y el puntaje de *confianza*.
- La *confianza* refleja cuán probable es que la caja contenga un objeto (*objectness*) y cuán precisa es la *bounding box*.
- El ancho y alto de la *bounding box* se normalizan con el ancho y alto de la imagen: x e y son *offsets* a la celda correspondiente; $\{x, y, w, h\} \in [0; 1]$.
- Cada celda tiene asociadas 20 probabilidades de clase.
- Shape de la predicción de YOLO:
 $(S, S, B \times 5 + C) = (7, 7, 2 \times 5 + 20) = (7, 7, 30)$; B es número de bounding boxes y se muestran valores típicos.



YOLO - Modelo

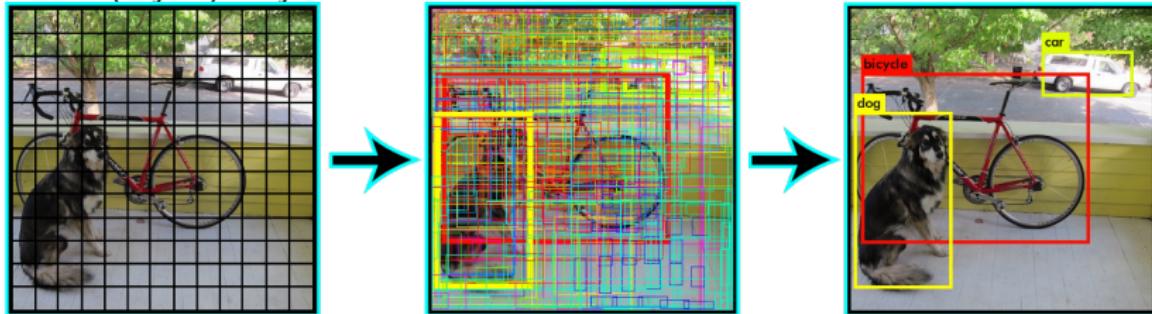
- Entonces, YOLO es una CNN multi-capa para predecir un tensor de $(7, 7, 30)$ por grilla.
- 24 capas convolucionales, seguidas de 2 densas.
- Reduce a 7×7 con 1024 canales de salida en cada ubicación.
- Luego hace una regresión para obtener las *bounding boxes*.



YOLO - Predicción

- Obtiene $7 \times 7 \times 2$ predicciones de *bounding box* (imagen central) por celda.
- Para la predicción final, se queda con las cajas con mayor puntaje de confianza, siempre que sean mayores que un umbral (0.25); (imagen derecha).
- El *class confidence score* se calcula para cada caja predicha; mide la confianza en la clasificación y localización, simultáneamente:
$$\text{class confidence score} = \text{box confidence score} \times P(\text{clase}_i | \text{objeto})$$
, con
$$\text{box confidence score} = P(\text{objeto}) \cdot \text{IoU}$$

 $P(\text{clase}_i | \text{objeto})$: probabilidad que objeto pertenezca a clase_i , dada la presencia del objeto.
 $P(\text{objeto})$: objectness.



YOLO - Función Loss

$$\begin{aligned} L = & \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] + \\ & \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 + \sum_{i=0}^{S^2} \mathbb{1}_{ij}^{obj} \sum_{c \in \text{clases}} (p_i(c) - \hat{p}_i(c))^2 \end{aligned}$$

- λ regulan importancia de distintos criterios.
- $\mathbb{1}_{ij}^{obj}$: 1 si celda contiene un objeto.
- C_i : *box confidence score (objectness)*.
- p_i : probabilidad de clase c en i .
- Primer término evalúa ajuste de coordenadas.
- Segundo y tercero evalúan *objectness* en presencia y ausencia de objetos, respectivamente.
- Cuarto término evalúa predicción de la clase.

YOLO - Rendimiento

- Un objeto por celda limita de acuerdo a la cercanía de los objetos: si los objetos están muy cerca en relación al tamaño de la grilla, pierde detecciones. En ejemplo, hay 9 Santas, pero YOLO puede detectar sólo 5.
- YOLO es significativamente más rápido que Faster R-CNN, pero no maneja tan bien la generalización de las formas.
- Para esto YOLO ha continuado evolucionando (YOLOv2, YOLOv3, YOLO9000). Más detalles en:

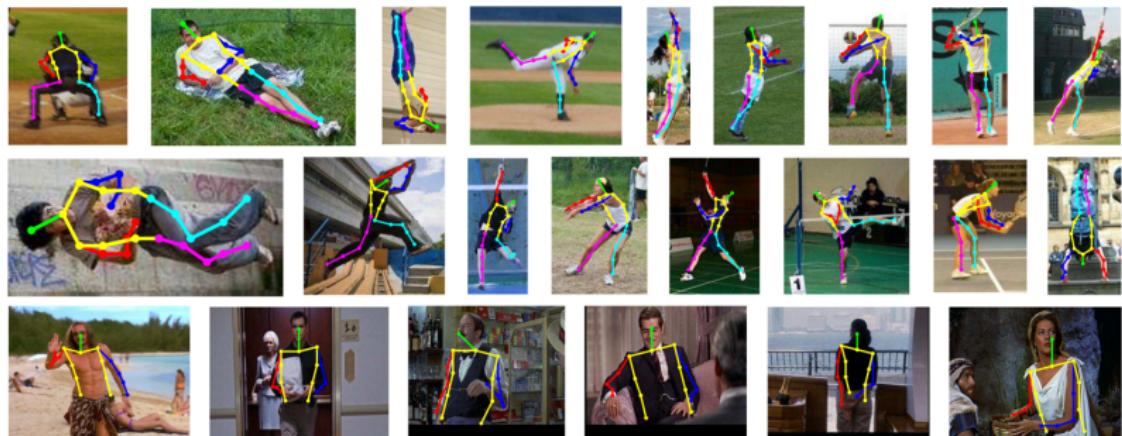
https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088



[Jonathan Hui]

Estimación de Pose (Humana)

- La estimación de pose (humana) se define como el problema de localizar las articulaciones (en un humano) en imágenes o videos; articulaciones llamadas *keypoints*.
- También se define como la búsqueda de una pose específica en el espacio de todas las poses.
 - *Estimación de pose 2D*: Pose en coordenadas 2D de la imagen.
 - *Estimación de pose 3D*: Pose en coordenadas 3D de la escena.
- Problema complejo: poses inusuales, articulaciones pequeñas y escasamente visibles, ocultaciones, ropa.



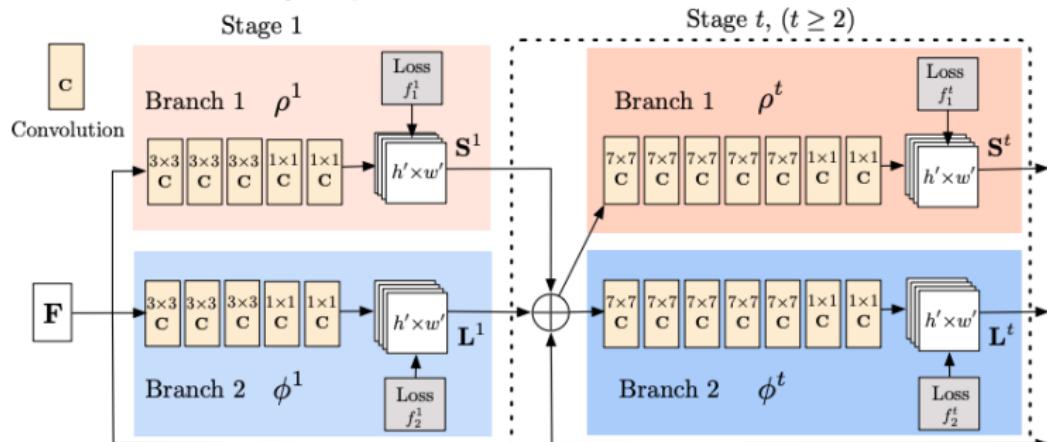
Estimación de Pose (Humana) - Mask R-CNN

- Mask R-CNN se puede extender con facilidad a Estimación de Pose Humana: se modela la ubicación de los *keypoints* con una máscara one-hot y se usa Mask R-CNN para predecir K máscaras, una por cada tipo de *keypoint* (e.g., hombro izquierdo, codo derecho, ...).
- Para cada uno de los K *keypoints* de una instancia, el resultado de entrenamiento es una máscara binaria one-hot de $m \times m$ con un sólo pixel etiquetado como *foreground*.
- Durante entrenamiento, por cada *keypoint* visible de *ground-truth*, se minimiza la *cross-entropy loss* sobre un softmax en m^2 (para motivar detección única).
- Más detalles en [He et al., 2017].



Estimación de Pose (Humana) - OpenPose

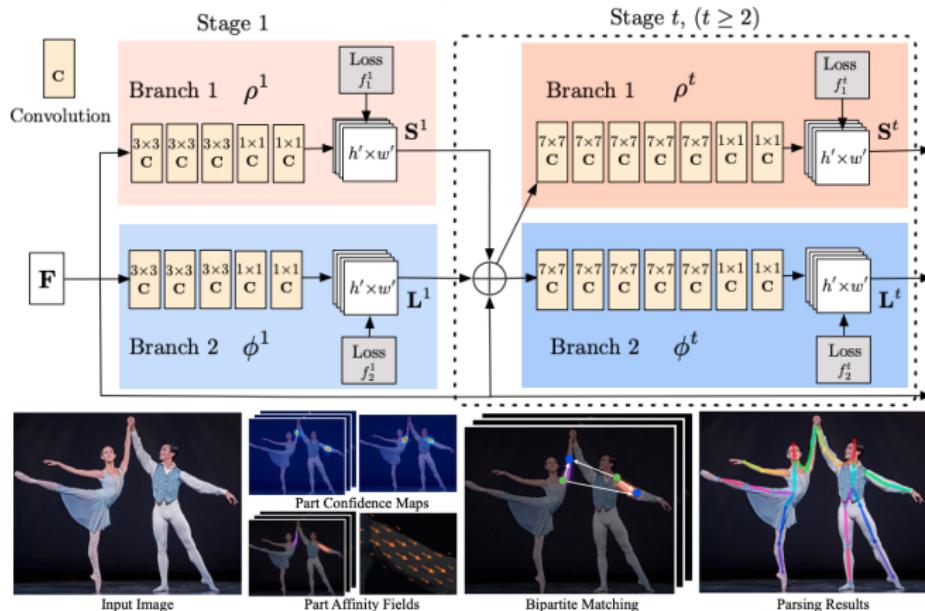
- Modelo para estimación de pose en tiempo real [Cao et al., 2018].
- Arquitectura:
 - Fase 1: entran mapas de características de imagen original RGB (procesados con red pre-entrenada, e.g. VGG) a un CNN con dos ramas (produce dos salidas diferentes). La rama superior predice los mapas de confianza de la ubicación de diferentes partes del cuerpo, como ojo derecho, codo derecho, etc. La rama inferior predice los campos de afinidad (*affinity fields*), que representan el grado de asociación entre diferentes partes del cuerpo. Entrega conjunto inicial de mapas de confianza S y mapas de afinidad L .



Estimación de Pose (Humana) - OpenPose

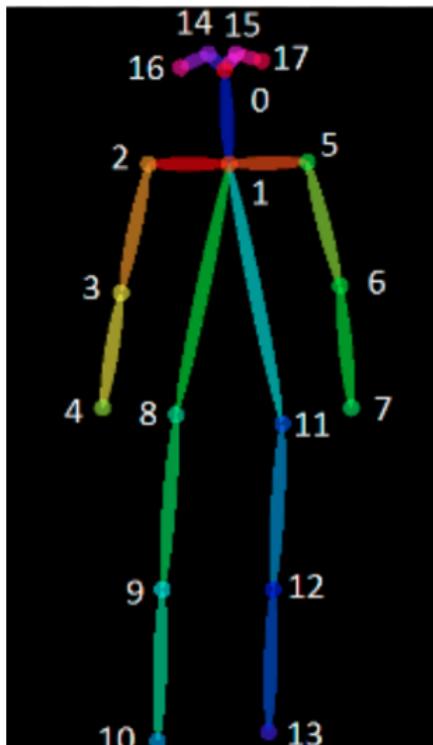
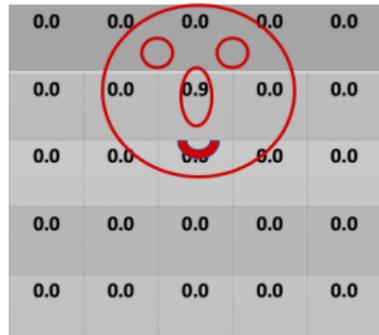
- Arquitectura:

- Fases t : entra concatenación de mapas de características de imagen original RGB, con L y S . Se usa para producir predicciones más refinadas. En implementación de *OpenPose*, fase final con $t = 6$.
- Entonces, L y S finales son procesados para obtener los *keypoints* 2D para todas las personas en la imagen.



OpenPose - Mapas de Confianza

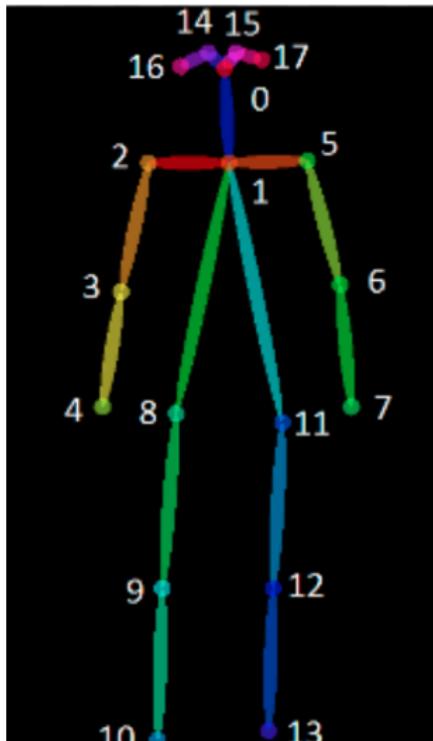
- Produce conjunto de mapas de confianza S :
 $S = (S_1, S_2, S_3, \dots, S_j)$, $S_j \in \mathbb{R}^{w \times h}$, con
 $j \in \{1 \dots J\}$, donde J es número total de partes del cuerpo.
- J depende del conjunto con el que se entrene OpenPose.
Para dataset COCO, $J = 19$ (*18 keypoints + background*).
- *Representación de S* : por ejemplo, entrenado con COCO,
se tendrá $S = (S_1, S_2, \dots, S_{19})$. Suponga que el elemento
 S_1 corresponde a mapa de confianza para el *keypoint* con
id=0 en la imagen a la derecha (nariz). Entonces, el
mapa de confianza debiese verse como la imagen inferior.



OpenPose - Mapas PAF

- Mapas *Part Affinity Fields* (PAF) corresponden a la rama inferior del modelo. Se definen como L :
 $L = (L_1, L_2, L_3, \dots, L_c)$, $L_c \in \mathbb{R}^{w \times h \times 2}$, con $c \in \{1 \dots C\}$, donde C es número total de extremidades (o partes del cuerpo).
- C depende del conjunto con el que se entrene OpenPose. Para dataset COCO, $C = 19$.
- Representación de L : cada elemento en L es un mapa de tamaño $w \times h$, donde cada celda contiene un vector 2D que representa la dirección de pares de elementos. Por ejemplo, en el diagrama a la derecha, el par (1, 2) corresponde a la dirección del cuello al hombro izquierdo.

```
CocoPairs = [
    (1, 2), (1, 5), (2, 3), (3, 4), (5, 6), (6, 7), (1, 8), (8, 9), (9, 10), (1, 11),
    (11, 12), (12, 13), (1, 0), (0, 14), (14, 16), (0, 15), (15, 17), (2, 16), (5, 17)
] # = 19
```



OpenPose - Función Loss

- Se usan dos *loss* al final de cada fase, una por rama. Usan una *loss* estándar L2 para la diferencia entre predicciones y *ground-truth*, en los mapas de confianza S y en los PAF L .
- Además, se agregaron pesos a cada *loss* para cubrir el problema práctico que algunos datasets no etiquetan completamente a todas las personas.
- Para cada fase t se tiene:

$$f_S^t = \sum_{j=1}^J \sum_p W(p) \|S_j^t(p) - S_j^*(p)\|_2^2$$

$$f_L^t = \sum_{c=1}^C \sum_p W(p) \|L_c^t(p) - L_c^*(p)\|_2^2$$

donde p representa una ubicación de pixel en la imagen. La notación $*$ en S y L identifica la *ground-truth*. Salida $S(p)$ es vector 1D para el *score* de confianza para esa parte del cuerpo j en particular en ubicación p . Salida $L(p)$ es vector 2D como vector direccional para la extremidad particular c en p . En paper, $J = 19$ y $C = 19$. $W(p)$ es la función de pesos ($W(p) = 0$ cuando no hay anotación en p).

- Función *loss* global:

$$f = \sum_{t=1}^T (f_S^t + f_L^t)$$

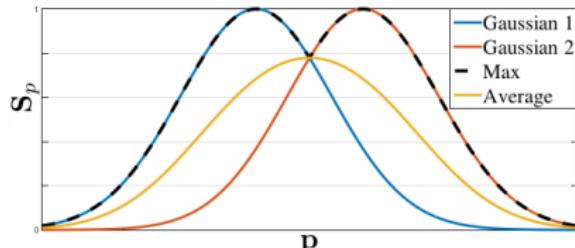
OpenPose - Entrenamiento - Mapas de confianza

- Para evaluar f_S durante entrenamiento, se generan los mapas de confianza de la *ground-truth* S^* , desde los *keypoints 2D anotados*.
- Cada mapa de confianza es una representación 2D sobre la certeza de que una parte del cuerpo determinada se ubique en cada pixel (un *peak* por persona presente).
- Primero se generan mapas de confianza $S_{j,k}^*$ individuales para cada persona k . Sea $x_{j,k} \in \mathbb{R}^2$ la posición de *ground-truth* de la parte j de la persona k en la imagen. El valor del pixel en la posición $p \in \mathbb{R}^2$ en $S_{j,k}^*$ es:

$$S_{j,k}^* = \exp\left(-\frac{\|p-x_{j,k}\|_2^2}{\sigma^2}\right),$$

donde σ controla dispersión del *peak*.

- Un mapa de confianza de la *ground-truth* S_j^* para una parte j está dado por:
$$S_j^*(p) = \max_k S_{j,k}^*(p).$$
- Se usa max para que la precisión de *peaks* cercanos se preserve.



OpenPose - Entrenamiento - PAF

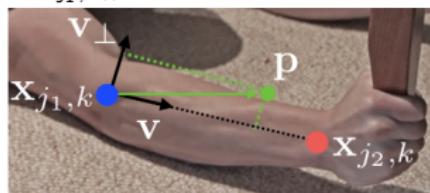
- Cada tipo de extremidad tiene su propio PAF: sean $x_{j_1,k}$ y $x_{j_2,k}$ dos posiciones en la *ground-truth* de partes del cuerpo j_1 y j_2 para la extremidad c de persona k en la imagen. Si un punto p se encuentra en c , el valor en $L_{c,k}^*(p)$ es un vector unitario que apunta desde j_1 a j_2 ; para todos los demás, se asigna vector cero.
- Para evaluar f_L durante el entrenamiento, se define la PAF de *ground-truth* $L_{c,k}^*$ considerando para cada pixel en la posición p :

$$L_{c,k}^*(p) = \begin{cases} v & \text{si } p \text{ en extremidad } c, k \\ 0 & \text{sino} \end{cases},$$

con $v = (x_{j_2,k} - x_{j_1,k}) / \|x_{j_2,k} - x_{j_1,k}\|_2$ vector unitario en dirección de la extremidad. El conjunto de puntos de la extremidad se define como aquellos dentro de una distancia umbral del segmento de línea:

$$0 \leq v \cdot (p - x_{j_1,k}) \leq l_{c,k} \wedge |v_\perp \cdot (p - x_{j_1,k})| \leq \sigma_l,$$

con ancho de la extremidad σ_l distancia máxima en píxeles a v , y largo de la extremidad $l_{c,k} = \|x_{j_2,k} - x_{j_1,k}\|_2$, y v_\perp vector perpendicular a v .



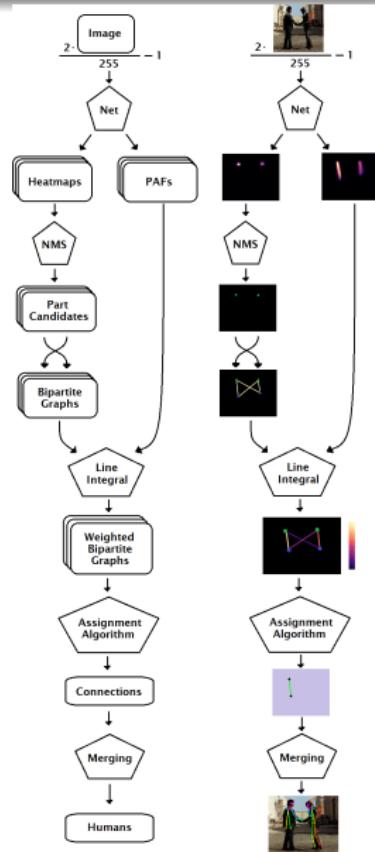
- El PAF de *ground-truth* promedia los PAF de todas las personas en la imagen:

$$L_c^* = \frac{1}{n_c(p)} \sum_k L_{c,k}^*(p), \quad n_c(p) : \text{núm. vectores } \neq 0 \text{ en } p \text{ entre las } k \text{ personas.}$$

OpenPose - Inferencia

Las salidas de la red en OpenPose no están listas para entregar directamente las poses humanas. El proceso de inferencia completo (pipeline) sigue diversos pasos para el resultado final.

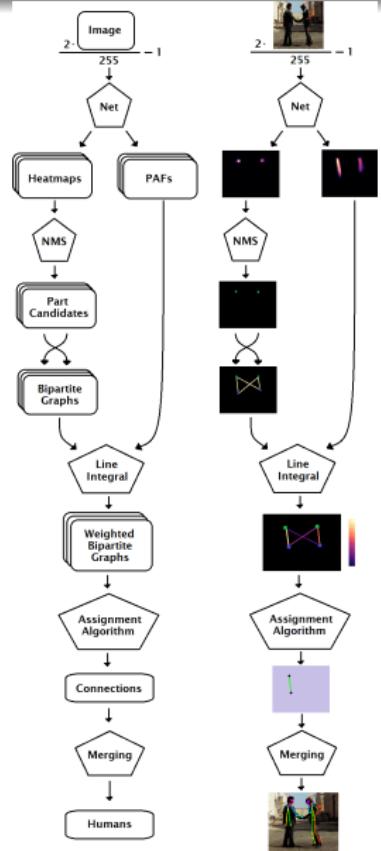
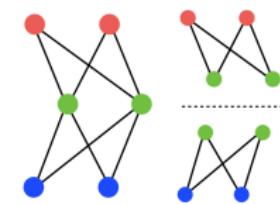
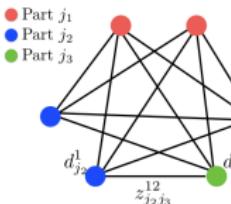
- ① *Pre-procesamiento:* Para la entrada a la red se convierte la imagen desde $[0, 255]$ a $[-1, 1]$.
- ② *CNN:* Se aplica la red descrita previamente, obteniendo los mapas de confianza como mapas de calor (heatmaps) y los PAFs.
- ③ *NMS:* A los mapas de calor se les aplica *non-maximum suppression* para obtener un conjunto discreto de ubicaciones de partes candidatas. Para cada parte, pueden haber varios candidatos (múltiples personas o falsos positivos).



OpenPose - Inferencia

4 Grafos bipartitos:

- Para simplificar el problema se generan grafos bipartitos entre los distintos pares de partes conectadas (pares de mapas).
- Para obtener poses completas de múltiples personas, se tiene problema *NP-Duro*.
- Los autores consideran dos relajaciones de restricción: primero, escoger número mínimo de aristas para obtener *spanning tree* del esqueleto de pose humana (árbol que conecta todas las partes de una persona), en vez de usar grafo completo; segundo, se descompone el problema de *matching* en un conjunto de subproblemas de *matching* bipartito, para determinar el *matching* en nodos adyacentes de forma independiente.



OpenPose - Inferencia

5 Integral de línea:

- Para ubicaciones de cada par de partes d_{j_1} y d_{j_2} , se muestrea el PAF predicho L_c a lo largo del segmento de línea para medir la confianza de su asociación:

$$E = \int_{u=0}^{u=1} L_c(p(u)) \cdot \frac{d_{j_2} - d_{j_1}}{\|d_{j_2} - d_{j_1}\|_2} du ,$$

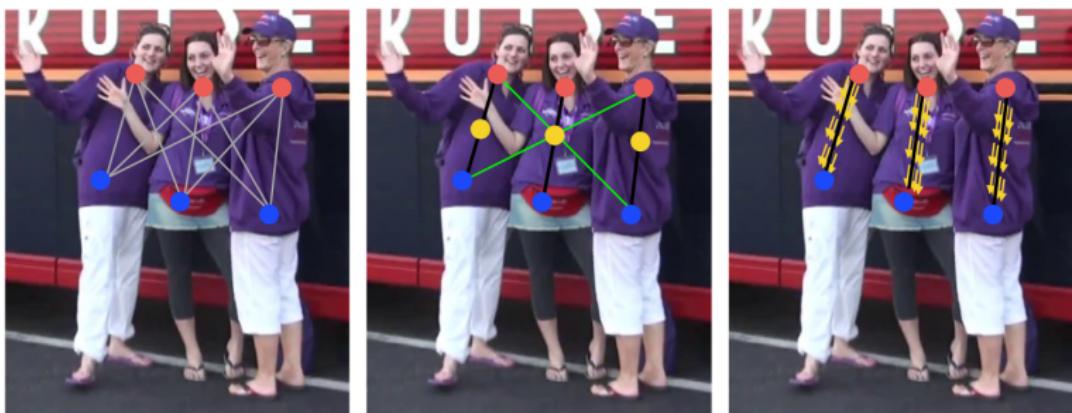
donde $p(u)$ interpola posición de las dos partes: $p(u) = (1-u)d_{j_1} + ud_{j_2}$.

- En la práctica, se aproxima integral muestreando y sumando a valores de u espaciados uniformemente.
- Para un conjunto de candidatas a detección de parte del cuerpo para múltiples personas $D_J = \{d_j^m : \text{para } j \in \{1 \dots J\}, m \in \{1 \dots N_j\}\}$ con N_j el número de partes candidatas j y $d_j^m \in R^2$ la ubicación del m -ésimo candidato de detección para la parte del cuerpo j .

OpenPose - Inferencia

5 Integral de línea:

- Considerando un sólo par de partes ($j_1; j_2$) para la c -ésima extremidad, encontrar la asociación óptima se reduce a un problema de *matching* de grafo bipartito de peso máximo.



- En este problema de *matching*, los nodos del grafo son los candidatos a detección de partes del cuerpo j_1 y j_2 y las aristas son todas las conexiones posibles entre pares de candidatos de detección. Además, cada arista está ponderada por E (integral de línea desde los PAF).

OpenPose - Inferencia

6 Algoritmo de asignación:

- Ahora, es necesario encontrar las conexiones que maximicen el puntaje total: resolver problema de asignación.
- Hay muchas formas, pero los autores por eficiencia utilizan una estrategia *greedy*:
 - Ordenar cada posible conexión según E de mayor a menor.
 - La conexión de mayor E se asigna de inmediato como conexión definitiva.
 - Moverse a siguiente conexión: Si ninguna de las partes de la conexión han sido ya asignadas a otra conexión, pasa a ser definitiva.
 - Repetir paso 3 hasta completar la lista.

	n1	n2	n3	n4
m1	.1	2	2.5	.3
m2	.2	1	4	.3
m3	3	.8	.2	.1



Emn	m	n	
4	m2	n3	✓
3	m3	n1	✓
2.5	m1	n3	✗
2	m1	n2	✓
1	m2	n2	✗
.8	m3	n2	✗

OpenPose - Inferencia

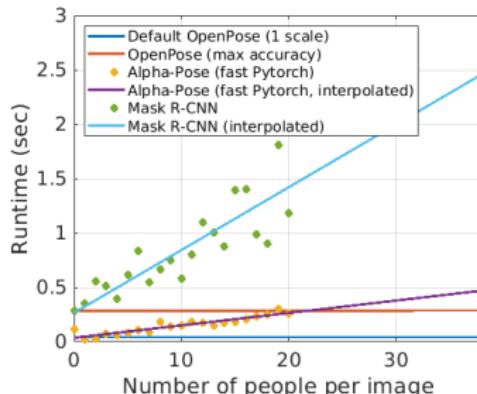
7 Fusión de extremidades (Merging):

- Finalmente, las conexiones definitivas que comparten las mismas partes candidatas se ensamblan, para formar poses de cuerpos completos para múltiples personas.
- El algoritmo parte con supuesto simple: cada conexión pertenece a un humano diferente (humanos = conexiones).
- Sean los humanos H una colección de conjuntos $H = \{H_1, H_2, \dots, H_k\}$ (k : número de humanos). Cada conjunto (cada humano) contiene, al principio, dos partes (un par).
- Si los humanos H_m y H_n comparten una parte j con las mismas coordenadas, entonces H_m pasa a ser $H_m \cup H_n$ y se elimina H_n .
- Se repite este proceso para cada par de humanos hasta que ninguna pareja comparta una parte.
- El resultado es un conjunto de poses de humanos, con las posiciones de cada parte del cuerpo.

OpenPose - Resultados

- OpenPose toma ventaja en cuanto a tiempo de ejecución.
- Existen mejores métodos en cuanto a AP (average precision).
- Reportes de algunos autores indican que falla bastante con personas parcialmente visibles (ocultación).
- Es uno de los pocos con código totalmente disponible para usar con facilidad (open source - con licencia para uso comercial, con restricción de uso en deportes).
- Existe herramienta comercial competitiva y mucho más rápida (wrnchAI - pagada, sin restricciones de uso):

<https://www.learnopencv.com/pose-detection-comparison-wrnchai-vs-openpose/>



Method	Hea	Sho	Elb	Wri	Hip	Kne	Ank	Subset of 288 images as in [1]		mAP	s/image
								Full testing set			
Deepcut [1]	73.4	71.8	57.9	39.9	56.7	44.0	32.0	54.1	57995		
Iqbal et al. [41]	70.0	65.2	56.4	46.1	52.7	47.9	44.5	54.7	10		
DeeperCut [2]	87.9	84.0	71.9	63.9	68.8	63.8	58.1	71.2	230		
Newell et al. [48]	91.5	87.2	75.9	65.4	72.2	67.0	62.1	74.5	-		
ArtTrack [47]	92.2	91.3	80.8	71.4	79.1	72.6	67.8	79.3	0.005		
Fang et al. [6]	89.3	88.1	80.7	75.5	73.7	76.7	70.0	79.1	-		
Ours	92.9	91.3	82.3	72.6	76.0	70.9	66.8	79.0	0.005		
DeeperCut [2]	78.4	72.5	60.2	51.0	57.2	52.0	45.4	59.5	485		
Iqbal et al. [41]	58.4	53.9	44.5	35.0	42.2	36.7	31.1	43.1	10		
Levinko et al. [71]	89.8	85.2	71.8	59.6	71.1	63.0	53.5	70.6	-		
ArtTrack [47]	88.8	87.0	75.9	64.9	74.2	68.8	60.5	74.3	0.005		
Fang et al. [6]	88.4	86.5	78.6	70.4	74.4	73.0	65.8	76.7	-		
Newell et al. [48]	92.1	89.3	78.9	69.8	76.2	71.6	64.7	77.5	-		
Fieraru et al. [72]	91.8	89.5	80.4	69.6	77.3	71.7	65.5	78.0	-		
Ours (one scale)	89.0	84.9	74.9	64.2	71.0	65.6	58.1	72.5	0.005		
Ours	91.2	87.6	77.7	66.8	75.4	68.9	61.7	75.6	0.005		

Keras - TensorFlow Serving

- TensorFlow Serving es un servicio de Tensorflow flexible y de alto rendimiento, para disponibilizar modelos de deep learning para inferencia.
- Se puede disponibilizar en el servidor distintos modelos que quedan activos para recibir consultas de inferencia.
- Luego, se pueden hacer consultas al servidor usando distintas API estándar: ejemplo con API Rest.
- Disponibilizado para el curso script para levantar servicio (*to_serve.py*), y clientes para inferencia en Python y C++.

Otros temas interesantes

- *Redes GAN*: Generative adversarial networks (GANs) son arquitecturas que usan dos redes adversarias. Una es el *generador*, para generar nuevas instancias de datos, y la otra, el *discriminador*, evalúa su autenticidad (decide si la instancia generada pertenece al conjunto de entrenamiento o no). Se generan nuevas instancias sintéticas. Usadas ampliamente en generación de imagen, voz y video.
- *RNN*: Una Recurrent Neural Network (RNN) es un tipo de red neuronal que contiene ciclos, que permiten que información se almacene dentro de la red: usa su razonamiento de experiencias previas para inferir eventos futuros. Se usan para secuencias de información (e.g. NLP (chatbots)), y en imágenes en particular para clasificación de acciones y generación de imágenes (entre otras).
- *CNN 3D*: Redes para secuencias de imágenes, usadas para reconocimiento de actividades.

Referencias I

- F. Caba H., V. Escorcia, B. Ghanem, and J. C. Niebles. Activitynet: A large-scale video benchmark for human activity understanding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 961–970, 2015.
- Z. Cao, G. Hidalgo, T. Simon, S. Wei, and Y. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *CoRR*, abs/1812.08008, 2018. URL <http://arxiv.org/abs/1812.08008>.
- J. Carreira, E. Noland, C. Hillier, and A. Zisserman. A short note on the kinetics-700 human action dataset. *CoRR*, abs/1907.06987, 2019. URL <http://arxiv.org/abs/1907.06987>.
- F. Chollet. Xception: Deep learning with depthwise separable convolutions, 2016. URL <http://arxiv.org/abs/1610.02357>.
- S. Das, R. Dai, M. Koperski, L. Minciullo, L. Garattoni, F. Bremond, and G. Francesca. Toyota smarthome: Real-world activities of daily living. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.
- J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011. URL <http://jmlr.org/papers/v12/duchilla.html>.

Referencias II

- M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results.
<http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html>, 2012.
- R. Girshick. Fast r-cnn. *CoRR*, abs/1504.08083, 2015. URL
http://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Girshick_Fast_R-CNN_ICCV_2015_paper.pdf.
- R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL <http://arxiv.org/abs/1311.2524>.
- X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.
- K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015a.
- K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015b. URL <http://arxiv.org/abs/1512.03385>.

Referencias III

- K. He, G. Gkioxari, P. Dollar, and R. Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017.
- D. P. Kingma and J. L. Ba. Adam: a method for stochastic optimization. In *International Conference on Learning Representations*, pages 1–13, 2015.
- I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, S. Kamali, M. Malloci, J. Pont-Tuset, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from <https://storage.googleapis.com/openimages/web/index.html>*, 2017.
- A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.
- A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.
- Y. Lecun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop, 1998.

Referencias IV

- Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Doklady ANSSSR*, 269:543–547, 1983.
- N. Qian. On the momentum term in gradient descent learning algorithms. *The Official Journal of the International Neural Network Society*, 12(1):145–151, 1999.
URL [http://doi.org/10.1016/S0893-6080\(98\)00116-6](http://doi.org/10.1016/S0893-6080(98)00116-6).
- J. Redmon, S. Kumar Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.
- S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015. URL <http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposals.pdf>.
- O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.
- O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, April 2015. doi: <https://doi.org/10.1007/s11263-015-0816-y>.

Referencias V

- A. Shahroudy, J. Liu, T.-T. Ng, and G. Wang. NTU RGB+D: A large scale dataset for 3d human activity analysis. *CoRR*, abs/1604.02808, 2016. URL <http://arxiv.org/abs/1604.02808>.
- K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- C. Szegedy, S. Ioffe, and V. Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. URL <http://arxiv.org/abs/1602.07261>.
- L. Tsung-Yi, M. Michael, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and C. L. Zitnick. Microsoft coco: Common objects in context. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Proceedings of the European Conference on Computer Vision - ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.
- J. Wang, Z. Liu, J. Chorowski, Z. Chen, and Y. Wu. Robust 3d action recognition with random occupancy patterns. In *Proceedings of the European Conference on Computer Vision - ECCV 2014*, 2012.
- H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL <http://arxiv.org/abs/1708.07747>.

Referencias VI

M. D. Zeiler. Adadelta: An adaptive learning rate method.
<http://arxiv.org/abs/1212.5701>, 2012.

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks.
CoRR, abs/1311.2901, 2013. URL <http://arxiv.org/abs/1311.2901>.