Facultad de Ingeniería y Ciencias Agrarias

Ingeniería Informática

Trabajo Final

Reconocimiento facial: FaceNet y AWS Rekognition

Marzo 2020

Alumno: Juan Francisco Diaz y Diaz

Tutor: Ricardo Hector Di Pasquale

Índice

Resumen

La evolución de tecnología y las comunicaciones nos ha permitido representar nuestro entorno en formato digital tanto en texto como en imágenes y video. La digitalización del medio que nos rodea dió la posibilidad de capturar grandes conjuntos de datos dando lugar a conceptos como "Big Data". Muchos de estos conjuntos pueden ser descargados gratuitamente de Internet e incluso hay competencias como ILSVRC (ImageNet Large Scale Visual Recognition Challenge) [1] y COCO (Common Objects in Context)[2], entre otros, donde se ponen a prueba algoritmos de clasificación de imágenes.

El desafío que propone Big Data no radica únicamente en el manejo de grandes conjuntos de datos sino también en el valor que podamos obtener de estos. Como consecuencia surge la necesidad de poder extraer patrones a partir de esta información de manera simple e involucrando lo menos posible al ser humano.

La solución a este desafío que ha ganado auge durante la última década es "Deep Learning" y esto se debe no solo a la habilidad de acceder fácilmente a dichos conjuntos de datos sino también al avance en la capacidad de cómputo tanto de CPU como GPU y la disponibilidad de herramientas que permiten enfocarse en el desarrollo de la idea y no en la complejidad del problema. Podemos destacar herramientas como Scikit-learn[3], Tensorflow[4], Keras[5] y servicios en cloud que permiten implementar Machine Learning y Deep Learning de manera accesible, sencilla y en poco tiempo.

En este trabajo se implementará un algoritmo de reconocimiento facial y se lo comparará con un servicio similar en cloud. Para poder dar contexto a la solución, se repasará brevemente la historia de Machine Learning desde sus comienzos en 1950 con la definición de la primer neurona Perceptron a su rápida evolución en la última década. También será necesario desarrollar conceptos teóricos de Machine Learning, haciendo énfasis en la clasificación de imágenes utilizando Neural Networks y Deep Learning.

La implementación estará dividida en dos partes, la primera basada en el paper "*FaceNet: A Unified Embedding for Face Recognition and Clustering*"[6] donde se utilizará el concepto de

[1] http://www.image-net.org/challenges/LSVRC
[2] http://cocodataset.org
[3] https://scikit-learn.org
[4] https://www.tensorflow.org
[5] https://keras.io
[6] https://arxiv.org/pdf/1503.03832.pdf

transfer learning para extender y aplicar un modelo preexistente sobre un conjunto de datos. La segunda implementación estará basada en el servicio cloud "*AWS Rekognition*"[7].

Estas dos implementaciones permitirán no solo utilizar conceptos de Deep Learning sino también realizar comparativas de:

- Costos
- Complejidad
- Performance
- Nivel de confianza en las predicciones
- Escalabilidad de la solución

[7] https://aws.amazon.com/rekognition

Introducción

En un principio la inteligencia artificial (IA) se comenzó a utilizar para solucionar problemas intelectualmente complejos para el ser humano pero relativamente simples para una computadora, ya que estos podían ser descriptos a través de reglas matemáticas (Kasparov vs IBM's Deep Blue, 1997). El verdadero desafío de la inteligencia artificial radica en poder resolver aquellas tareas que nos resultan simples de ejecutar pero sumamente complejas de formalizar, entre ellas, el reconocimiento del habla o el reconocimiento de rostros en imágenes. Estas acciones son intuitivas para nosotros pero no para un máquina.

Pero, qué es la Inteligencia Artificial? Formalmente podemos definirla como la subdisciplina del campo de la Informática, que busca la creación de máquinas o algoritmos que puedan imitar comportamientos inteligentes.

Actualmente resolver un problema de ajedrez no plantea un desafío para un máquina. Esto es así gracias al descubrimiento de algoritmos y nuevas técnicas durante la última década. Esta transformación promete ser disruptiva en campos como la medicina, la robótica, el transporte y las comunicaciones para dar algunos ejemplos.

Andrew Ng[8] profesor de la Universidad de Stanford y uno de los precursores en publicar abiertamente cursos de Machine Learning menciona, "AI Is the New Electricity". En muy pocas palabras resume el poder que conlleva la inteligencia artificial. En el desarrollo del trabajo veremos que una pieza fundamental en el poder de Machine Learning radica en el conjunto de datos que tengamos para analizar.

Pero, qué es Machine Learning? Formalmente podemos definirla como la rama del campo de la Inteligencia Artificial, que busca como dotar a las máquinas de capacidad de aprendizaje.

El desarrollo de un modelo de Machine Learning, sugiere que los sistemas de IA deben contar con la habilidad de obtener su conocimiento extrayendo patrones de los conjuntos de datos. Una de las soluciones a este desafío es hacer que las computadoras aprendan de la experiencia y entiendan nuestro entorno a través de una jerarquía de conceptos. Si dibujáramos un diagrama mostrando estos conceptos veríamos que se construyen uno por encima del otro, en muchas capas. A este comportamiento de la IA se lo denomina "Deep Learning" (DL). La esencia de Deep Learning radica no solo en apilar capas una por encima de la otra sino también en poder reutilizar componentes para crear nuevos modelos y arquitecturas. Durante este trabajo profundizaremos en conceptos de Deep Learning que nos permitirán entender cómo los modelos pueden comprender y procesar imágenes.

Deep Learning es un tipo de Machine Learning (ML) y para poder comprenderlo es necesario tener un entendimiento sólido de los principios de Machine Learning. Comenzaremos el

[8] https://www.linkedin.com/in/andrewyng/

desarrollo del trabajo recorriendo la historia de Machine Learning orientada en los hitos más importantes de Computer Vision. Seguiremos explicando conceptos de Deep Learning, enfocándonos en la arquitectura Convolucional, utilizada principalmente para el procesamiento de imágenes. Daremos un cierre a todos estos conceptos con la implementación de un modelo de Machine Learning para reconocer rostros y lo compararemos con el servicio de AWS Rekognition.

Motivación

Durante la última década el campo de la Inteligencia Artificial impulsó cambios disruptivos. Su rápida evolución se debe principalmente a la capacidad que tenemos de generar conjuntos de datos con millones de ejemplos gracias a las redes sociales, los sensores y los dispositivos móviles. Vale destacar también la apertura a compartir el conocimiento, tanto de empresas como universidades e investigadores. Compartir el conocimiento permitió que la evolución sea exponencial, ya que muchas de las herramientas que usamos son de código abierto, como Tensorflow, Keras, Scikit-learn. Contamos con cursos abiertos de universidades como "MIT 6.S094: Deep Learning for Self-Driving Cars[9]", "CS229 - Machine Learning[10]" y "CS231n: Convolutional Neural Networks for Visual Recognition[11]".

La posibilidad de acceso al conocimiento es muy grande lo que por momentos suele ser abrumador. Por este motivo, es importante definir un objetivo claro en el que podamos enfocarnos y nos ayude a fijar el conocimiento. Entender como una computadora "puede ver" resulta muy interesante y a la vez nos dará la capacidad de modelar e implementar soluciones como las planteadas en DeepLens[12] o DeepRacer[13]. Si bien hoy podemos encontrar soluciones que resuelven problemas como la identificación de objetos en tiempo real en imágenes y videos como YOLO[14], no sucede lo mismo para la identificación de rostros. Facebook comenzó a desarrollar esta tecnología hace años con el soporte de millones de usuarios que etiquetaban de manera gratuita los rostros de las personas que aparecían en imágenes construyendo conjuntos de datos extremadamente complejos. A partir de esto surgió el interés de poder implementar una solución similar que nos permita entender la teoría y la complejidad que hay detrás. Si bien la teoría exige contar con conceptos de Matemática y Estadística las herramientas y las capas de abstracción con las que contamos hacen que podamos prescindir, en alguna medida, de estos.

Una vez entendidos los conceptos teóricos y la implementación de la solución de reconocimiento, veremos que las posibilidades de añadir nuevas funcionalidades extendiendo la solución es enorme.

[9] https://selfdrivingcars.mit.edu/
[10] https://see.stanford.edu/Course/CS229/
[11] http://cs231n.stanford.edu/
[12] https://aws.amazon.com/es/deeplens/
[13] https://aws.amazon.com/es/deepracer/
[14] https://pjreddie.com/darknet/yolo/

Historia de Machine Learning

Los conceptos de IA no son nuevos y es necesario repasar su historia para poder comprender a qué se debe su auge durante la última década.

Se describirán algunos de los eventos históricos de mayor relevancia en el campo de Deep Learning mostrando principal atención en "Computer Vision".

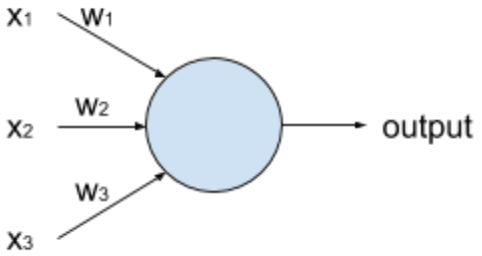
1943 Neural networks, Warren McCulloch y Walter Pitts[15]

Warren McCulloch (Neurofisiólogo) y Walter Pitts (Matemático) definieron el comportamiento de una neurona artificial. A partir de esta definición modelaron una red de neuronas simples utilizando circuitos electrónicos. Su trabajo se basó en estudios anteriores realizados sobre las neuronas del cerebro humano.

1957 Perceptron, Frank Rosenblatt[16]

Es un tipo de neurona artificial desarrollada por el científico Frank Rosenblatt, inspirado en trabajos anteriores realizados por Warren McCulloch y Walter Pitts. Estas neuronas reciben un conjunto de entradas binarias (x_i) y devuelven un único resultado binario (output). Para poder determinar la salida de estas neuronas, Rosenblatt introdujo el concepto de pesos (weights) y umbrales (threshold). Por lo tanto la salida de estas neuronas es computada a través de la sumatoria del producto del peso (w_i) y su entrada (x_i). Matemáticamente podemos representar a la Neurona Perceptron en términos de,

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$



Esta unidad lógica es la precursora de las redes neuronales actuales. En los primeros años de vida se creyó que este sería el algoritmo que resolvería una gran variedad de problemas aunque pronto se hizo evidente una de sus limitaciones más importantes. El algoritmo de aprendizaje automático que se utilizaba para entrenar al Perceptron no era extensible a otros tipos de redes más complejas y por lo tanto no se sabía cómo entrenarlas. Por este motivo

[15] https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf
[16] http://neuralnetworksanddeeplearning.com/chap1.html

hubo un corte repentino en la financiación de proyectos de inteligencia artificial. Este período de unos 15 años es conocido como el invierno de la IA.

1974 Backpropagation, Paul Werbos

Backpropagation (backward propagation of errors) es un método utilizado para el cálculo del gradiente.

A este concepto no se le dio suficiente importancia sino hasta 1986 con el paper "Learning representations by back-propagating errors" por David Rumelhart et al. Hoy este método es el "caballo de batalla" utilizado por los algoritmos de Deep Learning.

Este tema será explicado con mayor detenimiento a lo largo de este trabajo.

1986 Learning representations by backpropagating errors, Rumelhart et al.[17]

Esta publicación volvió a dar popularidad al concepto planteado por Perceptron. El trabajo muestra experimentalmente como utilizando un algoritmo de aprendizaje automático (backpropagation) una red neuronal ajusta automáticamente sus parámetros para aprender la representación interna de la información que está procesando. Para esto se introduce el concepto de Recurrent Neural Networks (RNN) como un nuevo proceso de aprendizaje utilizando backpropagation. Estas neuronas cuentan con un estado interno o memoria para procesar las secuencias de entrada y son comúnmente utilizadas en Natural Language Processing (NLP) para poder estimar el siguiente caracter en una secuencia de letras.

1989 a 1998: LeNet-5, Convolutional Neural Networks (CNNs)[18]

Las CNN son un tipo especial de multi-layer Neural Networks. Estas redes, al igual que la mayoría de las redes, son entrenadas utilizando el algoritmo de backpropagation. Las CNNs están diseñadas para reconocer patrones visuales a partir de los píxeles de las imágenes. LeNet-5 es un conjunto de trabajos realizados por Yann LeCun et al., principalmente para el reconocimiento de caracteres tanto escritos a mano como a máquina.

1995 NIST Special Database 19[19]

Base de datos que contiene materiales de capacitación de la National Institute of Standards and Technology (NIST) para el reconocimiento de caracteres a partir de formularios

[17] https://www.nature.com/articles/323533a0
[18] http://yann.lecun.com/exdb/lenet/
[19] https://www.nist.gov/srd/nist-special-database-19

completados a mano. Está compuesta de textos escritos por 3.600 autores y 810.00 imágenes de caracteres.

La primer edición de esta base fue publicada en Marzo de 1995. En Septiembre de 2016 se publicó una segunda edición.

1998 Gradient-Based Learning Applied to Document Recognition, LeCun et al.[20]

LeCun describe la implementación de una Convolutional Neural Networks (CNNs) para identificar dígitos escritos a mano en cheques bancarios. Este fué el primer avance significativo en la utilización de CNN para la identificación de objetos en imágenes.

1999 SIFT & Object Recognition, David Lowe

Feature-based object recognition: El autor propone que es posible detectar y comparar objetos en función de sus características, ya que es muy complejo determinar si un objeto es similar a otro debido a que estos sufren cambios como pueden ser, el ángulo desde donde se toma la fotografía, el punto de referencia del observador y la luz.

2001 Face Detection, Viola & Jones[21]

Reconocimiento facial en imágenes en tiempo real.

2005 y 2009 Human recognition

- Histogram of Gradients (HoG), Dala & Triggs, 2005
- Deformable Part Model, Felzenswalb, McAllester, Ramanan, 2009

Estos dos papers buscan detectar las distintas partes del cuerpo para así poder identificar al ser humano en imágenes.

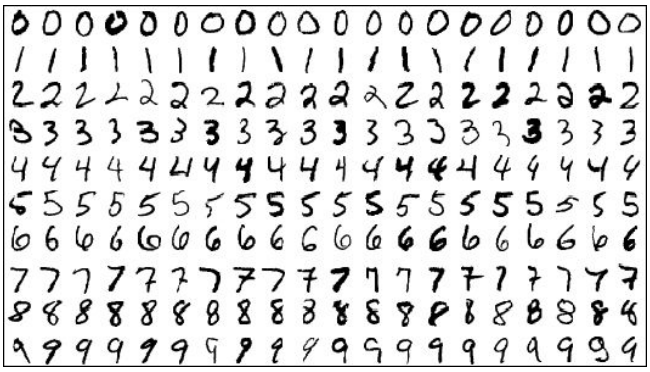
2005 a 2012 PASCAL Visual Object Challenge, Everingham et al.

Competencia de reconocimiento y detección de hasta 20 categorías en un conjunto de imágenes.

[20] http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf
[21] https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

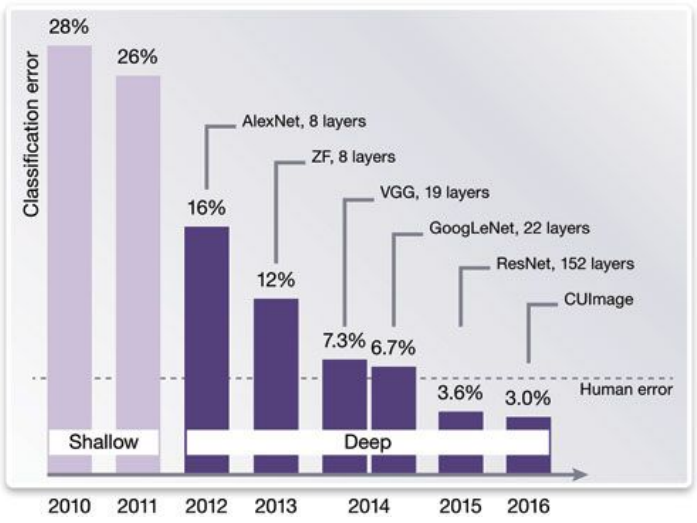
2007 The MNIST database of handwritten digits[22]

Esta base de datos contiene 60.000 ejemplos de entrenamiento y 10.000 ejemplos de prueba de dígitos escritos a mano. Está compuesta por imágenes en blanco y negro de dígitos de 0 a 9 con una resolución de 28x28 píxeles. Es una evolución de la NIST, con ejemplos de entrenamiento de la SD-3 (por empleados de la Census Bureau) y ejemplos de prueba de la SD-1 (por estudiantes de secundaria). Se la considera el "Hello world" de Machine Learning ya que es un excelente prototipo para comenzar a entender los conceptos de Neural Networks. En la imagen vemos una muestra de los distintos dígitos que tiene el conjunto de datos.



2009 ImageNet, Deng, Dong, Socher, Li, Li, & Fei-Fei[23,24]

Competencia que evalúa algoritmos para la localización y detección de objetos en imágenes y videos. Cuenta con 22k categorías sobre un conjunto de 14M de imágenes.



Durante la edición realizada en el 2012 se presentó por primera vez un algoritmo que utilizaba Deep Learning el cual consiguió superar considerablemente al ganador del año anterior. El algoritmo denominado AlexNet, contaba con 8 capas y fué desarrollado por Alex Krizhevsky. En la Figura se puede observar la evolución en la competencia desde de la introducción de Deep Learning a partir de AlexNet en el 2012.

[22] http://yann.lecun.com/exdb/mnist/
[23] http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf
[24] http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture1.pdf

2014 DeepFace: Closing the Gap to Human-Level Performance in Face Verification[25]

Sistema de reconocimiento facial desarrollado por Facebook basado en Deep Learning. Cuenta con 9 layers y más de 120 millones de conexiones. Este algoritmo fue entrenado utilizando la base de datos de imágenes de Facebook con más de 4 millones de imágenes.

2016 AlphaGo, Mastering the game of Go with deep neural networks and tree search[26]

Programa desarrollado por Google DeepMind para jugar al juego de mesa Go. Este juego se considera de suma complejidad para ser implementado utilizando Inteligencia Artificial debido a la cantidad de alternativas posibles y la dificultad de evaluar las posiciones en el tablero.

2017 Capsule Networks (CapsNet), Geoffrey E. Hinton et al.[27]

Geoffrey Hinton (El padrino de Deep Learning) introduce el concepto de cápsulas. Estas se representan como un grupo de neuronas cuya salida expresa diferentes propiedades de la misma entidad. Cada capa está compuesta de múltiples cápsulas. Estas redes buscan resolver las limitaciones de las CNNs. Las CNNs no tienen referencia de espacio, en otras palabras, para que una cara sea válida deberá estar compuesta por ojos, nariz, boca sin importar su ubicación. Las CapsNet almacenarán las referencias necesarias no sólo para determinar estos atributos sino también las referencias de espacio entre ellas. Las CapsNet han sido puestas a prueba sobre el conjunto de MNIST en las cuales se las considera "state-of-the-art".

2018 BERT (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers), Google AI Language[28]

Método de pre-entrenamiento para la representación de lenguajes utilizado en NLP (Natural Language Processing) desarrollado por Google AI Language. El modelo cuenta con 1024 layers y 340M parámetros. El entrenamiento de este modelo requiere de una gran capacidad de cómputo. BERT obtuvo resultados a nivel state-of-the-art para tareas de NLP.

[25] https://www.cs.toronto.edu/~ranzato/publications/taigman_cvpr14.pdf
[26] https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf
[27] https://arxiv.org/pdf/1710.09829.pdf
[28] https://arxiv.org/pdf/1810.04805.pdf

2019 Language Models are Unsupervised Multitask Learners (GPT-2), OpenAI[29]

El modelo llamado GPT-2, sucesor de GPT, con 1.5 billones de parámetros fué entrenado por OpenAI con un dataset especialmente preparado con textos obtenidos de internet. El objetivo del modelo es poder predecir la palabra que le sucede a un texto. Este modelo se encuentra al nivel de state-of-the-art y por este motivo OpenAI decidió no hacer público un modelo pre-entrenado ya que puede ser utilizado por ejemplo, para generar noticias falsas. Dicho modelo tiene la capacidad de escribir más de un párrafo a partir de una primer oración.

[29] https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Machine Learning (ML)

Definición

Se define Machine Learning[30] como una colección de algoritmos y técnicas que le permiten a una máquina aprender a partir de un conjunto de datos. Estos datos están representados numéricamente por vectores y matrices.

Tom Mitchell (1997)[31] define un algoritmo que aprende como: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

Tarea, T

Aprender es obtener la capacidad de realizar una tarea *T* descripta a través de ejemplos. Un ejemplo está compuesto por un conjunto de características (features).

Las tareas más comunes que se pueden representar con ML son:

- **Clasificación (Classification)**, especifica a qué categoría *k* pertenece una determinada entrada (input).
- **Regresión (Regression)**, predice un valor numérico en función de una entrada (input).
- **Traducción (Machine translation)**, la entrada consiste en una secuencia de símbolos en algún lenguaje que deben ser convertidos a una secuencia de símbolos en otro lenguaje.
- **Detección de anomalías (Anomaly detection)**, determina si un evento es excepcional o atípico.

Medición de performance, P

Para poder evaluar la capacidad o performance *P* de un algoritmo de ML es necesario determinar cualitativamente su rendimiento; el cual será específico a la tarea que se quiere realizar. Para evaluar dicha performance, mediremos la precisión del modelo evaluando para cuantos ejemplos obtuvimos una salida correcta. Comúnmente trabajaremos con su equivalente opuesto, la **tasa de error** (error rate).

[30] http://www.deeplearningbook.org/contents/ml.html
[31] https://www.cs.cmu.edu/~tom/mlbook.html

Definiremos la tasa de error como la proporción de ejemplos para los cuales se predijo una salida incorrecta.

Esta medición debe realizarse sobre un **conjunto de pruebas** (test set) y debe estar separado del **conjunto de datos** o ejemplos (dataset) que se utilizó para entrenar el modelo.

Experiencia, E

Los algoritmos de ML se pueden categorizar en no supervisados (unsupervised), supervisados (supervised) o reforzados (reinforcement) según el tipo de experiencia que tienen durante el proceso de aprendizaje.

Unsupervised Learning

Algoritmos que aprenden las propiedades de un conjunto de datos (dataset). Estos datos están compuestos por distintas características (features) y no están etiquetados. En el contexto de Deep Learning, el aprendizaje no supervisado (unsupervised learning), se utiliza para determinar la distribución probabilística del conjunto de datos. También es utilizado para agrupar ejemplos similares dentro de un conjunto de datos, esto se conoce como clustering. Los conjuntos de datos para entrenar son menos costosos de conseguir ya que los propios algoritmos se encargan de buscar los patrones de los conceptos representados.

Supervised Learning

Algoritmos que aprenden las propiedades de un conjunto de datos (dataset). Estos ejemplos están compuestos por distintas características (features) los cuales tienen asociada una etiqueta o valor de verdad. Normalmente los datos son representados como vectores, donde una componente del par son los datos de entrada y la otra, los resultados deseados.

Algoritmos que aprenden a asociar una entrada (input o features) con una salida (output o target) dado un conjunto de entrenamiento (dataset) donde la entrada es **x** y la salida es **y**. En algunos casos la salida **y** es difícil de obtener automáticamente y debe ser proveída por un humano "supervisor".

Reinforcement Learning

Estos algoritmos (también llamados agentes) no cuentan con un conjunto de datos sobre el que puedan aprender, sino que el agente debe interactuar con el ambiente (o mundo exterior) y aprender en base a la experiencia ganada en el mismo.

Reinforcement learning es comúnmente utilizado para entrenar modelos que juegan videojuegos (OpenAI Gym[32]). El agente ejecuta acciones sobre el ambiente y el ambiente

[32] https://github.com/openai/gym

responde a estas acciones con observaciones y recompensas. En base a estas respuestas el agente aprende. Las recompensas son extrínsecas al agente.

Lo que buscarán estos algoritmos es hacer que el agente sea capaz de no quedarse limitado por la escasez de recompensa dada por el ambiente. A partir de eso surge la idea de generar una recompensa intrínseca haciendo que el agente sienta curiosidad. El concepto de curiosidad en un algoritmo de reinforcement learning está probado en el paper Large-Scale Study of Curiosity-Driven Learning[33] de OpenAI. Donde el agente intentará predecir cómo será el ambiente en el futuro y así comparar la predicción con lo que en realidad ocurrió. El error en la predicción de lo que el agente cree que va a pasar es lo que determinará su curiosidad, a mayor error mayor la curiosidad por lo desconocido.

Muchos de los algoritmos desarrollados por OpenAI son OpenSource[34].

Capacidad, Overfitting & Underfitting

El desafío central de los algoritmos de ML es poder responder correctamente ante datos de entrada nunca antes vistos. La habilidad de responder bien a este tipo de datos se denomina **generalización** (generalization).

Cuando entrenamos un modelo lo hacemos utilizando un conjunto de **datos de entrenamiento** (training set) sobre el cuál podemos calcular un **error de entrenamiento** (training error). Lo que buscaremos durante esta etapa es reducir al máximo este error. No solo queremos disminuir dicho error sino también el **error de generalización** (generalization error) o también conocido como **error de pruebas** (test error). El error de generalización es el error esperado ante una nueva entrada y se estima a partir del conjunto de pruebas (test set).

Para poder estimar dichos errores de manera correcta, donde estos tengan un sentido real, tendremos en cuenta una serie de suposiciones. Los ejemplos contenidos en ambos conjuntos de datos deberán:

- ser independientes uno de otros
- estar distribuidos de manera idéntica, es decir, deberán tener la misma distribución de probabilidad.

Estas suposiciones nos permitirán estudiar matemáticamente la relación entre el error de entrenamiento (training error) y el error de generalización (generalization error).

Los factores que determinarán el correcto funcionamiento de un algoritmo de Machine Learning serán la habilidad a:

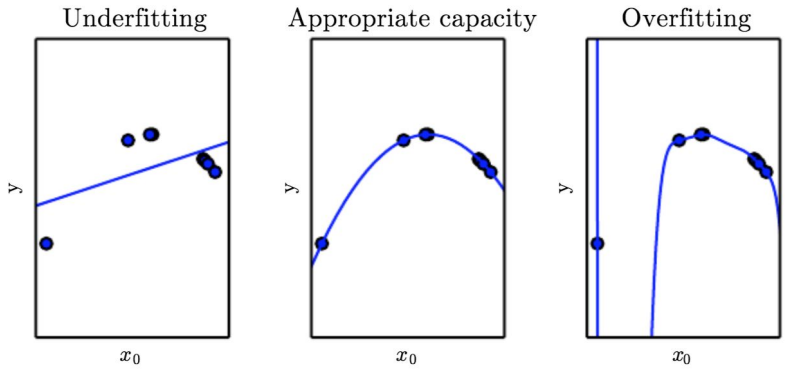
1. Hacer que el error de entrenamiento (training error) sea lo mas chico posible.

[33] https://pathak22.github.io/large-scale-curiosity/resources/largeScaleCuriosity2018.pdf
[34] https://github.com/openai/baselines

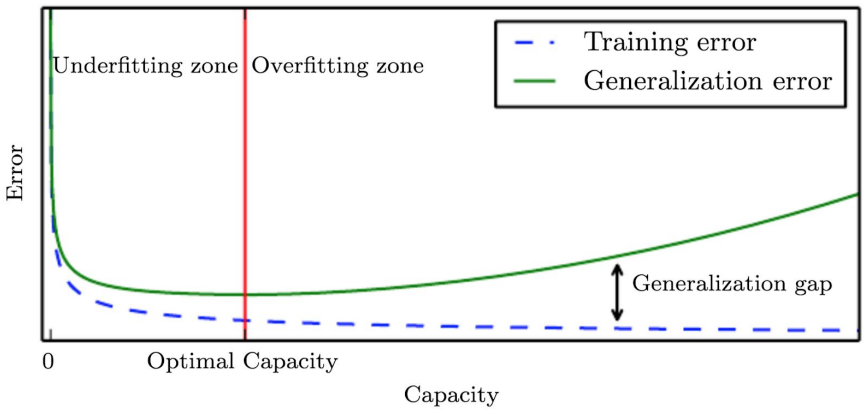
2. Hacer que la diferencia entre el error de entrenamiento (training error) y el error de generalización (generalization error) sea lo más pequeño posible.

Estos dos factores corresponden a los conceptos de **underfitting** (variance) y **overfitting** (bias). Underfitting ocurre cuando un modelo no puede hacer que el error de entrenamiento sea lo suficientemente pequeño. Overfitting ocurre cuando la diferencia entre el error de entrenamiento (training error) y el error de generalización (test error) se hace muy grande.

Es posible controlar si un modelo caerá en underfitting u overfitting modificando su **capacidad** (capacity). La capacidad de un modelo es la habilidad que tiene de soportar una amplia variedad de funciones. Modelos con baja capacidad tendrán problemas para capturar todo el conjunto de entrenamiento y caerán en underfitting. Por el contrario, modelos con alta capacidad podrán caer en overfitting al tener que memorizar propiedades del conjunto de entrenamiento que no serán utilizadas por el conjunto de pruebas.



En la figura se comparan tres modelos donde el primero está representado por una función lineal, el segundo por una función cuadrática y el tercero por un polinomio de grado 9. La función lineal no logra capturar la curvatura y cae en underfitting. El polinomio es capaz de representar el conjunto de datos de manera correcta, pero queda condicionado a la particularidad de los ejemplos de entrenamiento y por lo tanto cae en overfitting. La función cuadrática es la que mejor generaliza el problema.



En la figura se puede ver la relación que existe entre la capacidad y el error. El error de generalización y el de entrenamiento se comportan de manera diferente. A medida que se incrementa la capacidad, el error de entrenamiento se hace más chico pero la diferencia entre los errores de generalización y entrenamiento se hacen más grandes.

Regularización (Regularization)

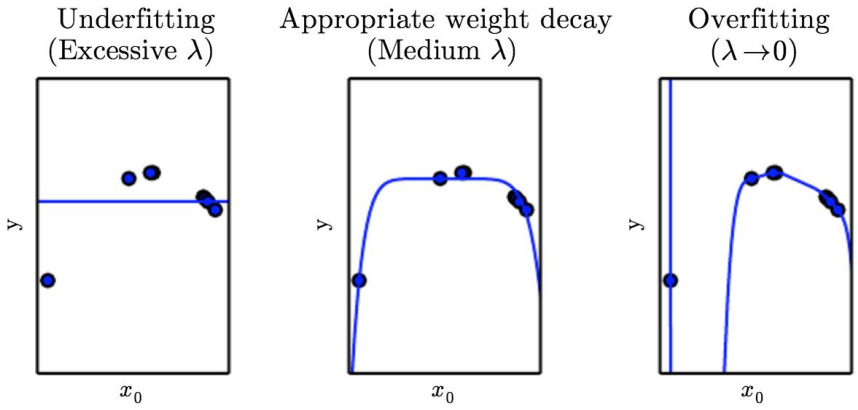
Los algoritmos de ML se diseñan de manera que puedan desempeñar una tarea concreta de manera correcta. Durante el diseño se define un conjunto de preferencias que modelan el comportamiento del algoritmo. Cuando estas preferencias están alineadas al tipo de problema que queremos resolver entonces los algoritmos responden mejor.

Hasta el momento, la única manera de modificar el comportamiento de los algoritmos es a través de la capacidad. Recordemos que la capacidad nos permite modificar la representación de nuestro modelo, agregando o quitando funciones del espacio de soluciones (o hipótesis).

La **regularización** es cualquier modificación que se realiza a un algoritmo con la intención de reducir el error de generalización o pruebas, pero no el error de entrenamiento.

Una forma de regularización es expresar una preferencia por alguna función en particular por sobre otras, en lugar de agregar o quitar funciones de espacio de hipótesis (como sucede con la capacidad). No hay una forma de regularización correcta sino que el tipo de regularización aplicada a un algoritmo dependerá de la tarea que se quiera resolver y por lo tanto será particular a cada solución.

La regularización busca penalizar la complejidad del modelo en lugar de "hacer encajar" el conjunto de entrenamiento.



Hiper-parámetros y conjunto de validación

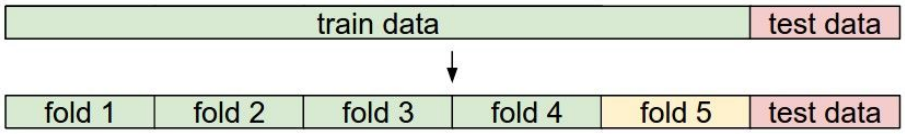
Los hiper-parámetros (Hyperparameters) nos permiten modificar el comportamiento de un algoritmo de Machine Learning. La capacidad y la regularización, por ejemplo, son tipos de hiper-parámetros.

Los parámetros de configuración que suelen ser elegidos como hiper-parámetros son aquellos que un algoritmo no puede aprender por sí mismo ya que son sumamente complejos de optimizar.

Para poder ajustar correctamente los hiper-parámetros es necesario contar con un conjunto de datos que no haya sido observado previamente por el algoritmo durante su entrenamiento. No es posible utilizar el conjunto de pruebas para ajustar los hiper-parámetros ya que este conjunto no debe ser utilizado sino hasta el último momento. Si lo utilizáramos corremos el riesgo de ajustar los hiper-parámetros a nuestro conjunto de pruebas y frente a nuevos datos su performance se verá seriamente afectada. Por esto debemos definir un nuevo conjunto denominado **conjunto de validación** y se puede obtener a partir del conjunto de entrenamiento antes de ser entrenado el algoritmo. Por lo tanto, el conjunto de entrenamiento será utilizado para aprender cuáles serán los hiper-parámetros a usar y el conjunto de validación será utilizado para estimar el error de generalización durante o después del entrenamiento. Se suele destinar entre el 70-90% de los datos al conjunto de entrenamiento y esto estará fuertemente relacionado al número de hiper-parámetros a estimar.

Validación cruzada (cross-validation)

Hay situaciones en las que el tamaño del conjunto de entrenamiento es pequeño y por ende el conjunto de validación también. En estos casos se utiliza una técnica más sofisticada para ajustar los hiper-parámetros y se denomina validación cruzada. En lugar de definir un único conjunto de validación se divide el conjunto de pruebas en distintos conjuntos de validación iterando sobre cada uno de estos y promediando su rendimiento.



En la imagen se puede ver como el conjunto de entrenamiento es dividido en distintos "folds" donde el fold 5 representa el conjunto de validación. Estos folds se van iterando y cada uno de ellos pasa a ser, en algún momento, un conjunto de validación. Al final de la iteración se obtendrán los mejores hiper-parámetros y recién ahí, y por única vez, se utilizará el conjunto de pruebas.

En la práctica se prefiere evitar el uso de cross-validation ya que es computacionalmente más caro que tener un único conjunto de validación. Pero como se mencionó anteriormente esto

será útil cuando el conjunto de entrenamiento es pequeño. Los números de folds utilizados normalmente son: 3-fold, 5-fold ó 10-fold.

Clasificación de imágenes

Podemos definir al problema de clasificación de imágenes como la tarea de asignar una etiqueta (label) a una imagen de entrada (input) a partir de un conjunto fijo de categorías. Este es uno de los problemas centrales en Computer Vision.

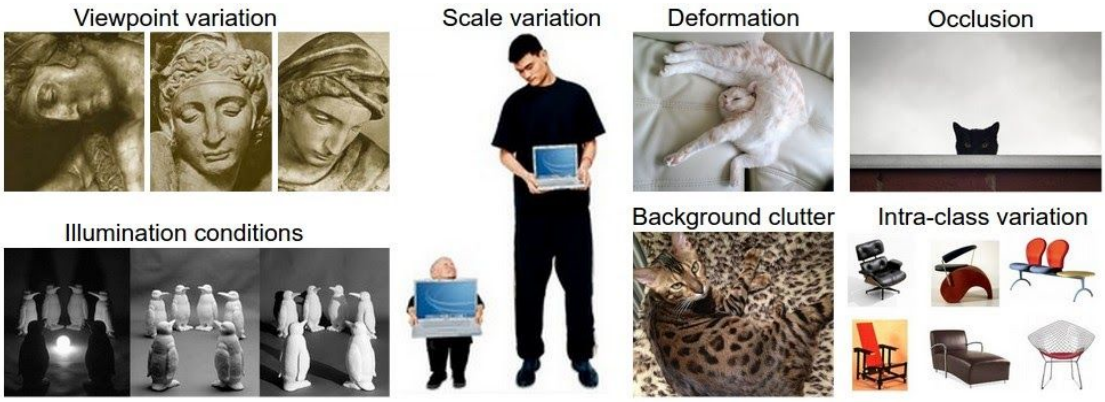
Las imágenes se representan en una matriz o array de números de 3-dimensiones, donde cada dimensión será un canal de color Red, Green y Blue (RGB). Las dimensiones del array estarán definidas según el tamaño de la imagen. Cada número que compone el array de datos será del tipo integer en un rango de 0 (negro) a 255 (blanco).

Si tenemos una imagen de dimensiones 248 x 400 pixels y 3 canales RGB tendremos una matriz con 297.600 números (248x400x3).

Desafíos

La tarea de reconocer conceptos visuales, como por ejemplo un gato, es relativamente trivial para los seres humanos pero bastante más compleja para un algoritmo de Computer Vision. Es importante y necesario destacar algunos de los desafíos que deben resolver estos algoritmos para poder entender donde radica su complejidad.

- **Variación del punto de vista** (Viewpoint variation): una instancia de un objeto puede estar orientado de diferentes maneras con respecto a la cámara.
- **Variación de escala** (Scale variation): un objeto de una misma clase a menudo exhibe variaciones en su tamaño. No solo en términos de la imagen sino en el mundo real.
- **Deformación** (Deformation): Los objetos no son cuerpos rígidos y pueden deformarse en formas extremas.
- **Oclusión u Obstrucción** (Occlusion): Los objetos de interés pueden estar ocultos u obstruidos por otros donde solo una porción del objeto (algunos pixels) es visible.
- **Condiciones de iluminación** (Illumination conditions): los efectos de iluminación pueden modificar drásticamente una imagen.
- **Desorden de fondo** (Background clutter): Los objetos pueden estar mimetizados con el ambiente haciendo muy compleja su identificación.
- **Variación intra-clase** (Intra-class variation): las clases de interés pueden ser relativamente amplias. Estas clases están compuestas de distintas formas, si bien representan el mismo objeto, y por su propiedades puede representarse de muchas maneras, como por ejemplo una silla.

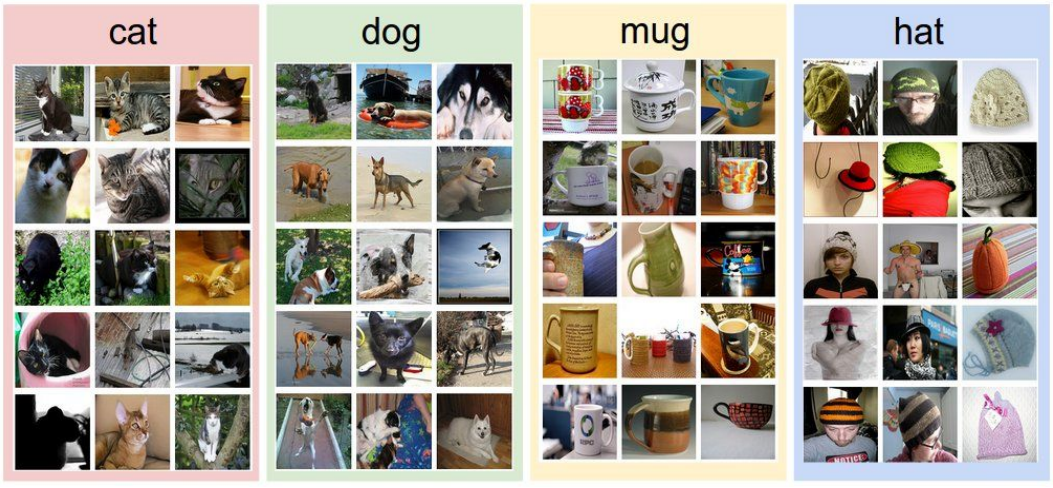


Un buen algoritmo de clasificación de imágenes debe ser capaz de adaptarse a todas estas variaciones manteniendo, al mismo tiempo, la sensibilidad a las variaciones entre la misma clase (inter-class).

Enfoque basado en datos (Data-driven approach)

Debido a que es sumamente complejo escribir un algoritmo que identifique objetos en una imagen se debe utilizar un enfoque distinto. Utilizaremos un algoritmo de Machine Learning al cual se le proveerá de muchos ejemplos. Todos estos ejemplos serán de distintas clases para que aprenda las características visuales y variaciones de cada una de ellas. Este enfoque es considerado "data-driven" ya que se basa primero en acumular un conjunto diverso de datos de entrenamiento, con imágenes etiquetadas para luego poder entrenar un modelo. Si prestamos atención a esta definición vemos que no es más que el concepto de Machine Learning, más precisamente, un problema de clasificación.

A continuación veremos un ejemplo de un conjunto de entrenamiento con cuatro categorías distintas. En la práctica este conjunto tendrá miles de categorías y cientos de miles de imágenes.



Clasificación lineal

La clasificación lineal tiene dos componentes,

- **Función de puntuación** (score function): mapea los datos en bruto (raw data) a las puntuaciones de clase (class scores). En el caso de las imágenes, se relacionarán los valores de los pixels a las puntuaciones de confianza (confidence scores) de cada una de las clases.

- **Función de pérdida** (loss function): cuantifica el acuerdo entre la puntuación de predicción (predicted scores) y las etiquetas de verdad absoluta (ground truth labels).

Debido a que debemos minimizar la función de pérdida con respecto a los parámetros de la función de puntuación, podemos categorizarlo como un problema de optimización.

Supongamos un conjunto de entrenamiento de imágenes representado por $x_i \in R^D$, cada uno asociado a una etiqueta y_i. Donde $i = 1...N$ e $y_i \in 1...K$ y por lo tanto tenemos N ejemplos (cada uno de dimensionalidad D) y K categorías distintas.

Para ejemplificar esta definición utilizaremos el conjunto de datos CIFAR-10[35].

- N = 50.000 imágenes
- D = 32 x 32 x 3 = 3072 pixels
- K = 10 clases distintas

Definiremos el espacio de soluciones de la función de puntuación (score function) como:

$$f : R^D \rightarrow R^K$$

que mapea los pixeles de las imágenes con las clases de puntuación (class scores). Y a su ecuación como:

$$f(x_i, W, b) = Wx_i + b$$

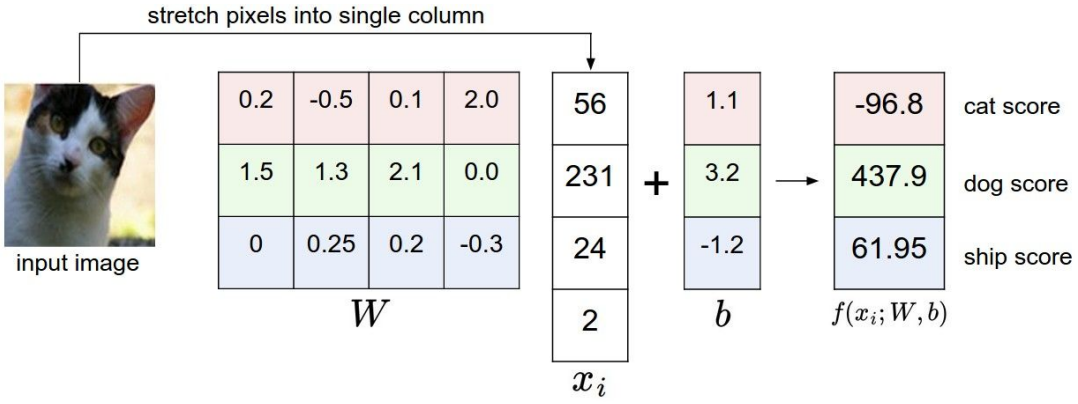
En esta ecuación asumimos que la imagen $\mathbf{x}_i$ tiene todos sus pixeles "aplanados" en un vector columna de forma [D x 1]. La matriz **W** de dimensiones [K x D] y el vector **b** de dimensiones [K x 1], son los parámetros de la función. **W** es comúnmente llamada **pesos** (weights) o parámetros y **b** es llamado **vector bias** por su influencia en las puntuaciones de salida (output scores) pero sin interactuar con los datos x_i.

[35] https://www.cs.toronto.edu/~kriz/cifar.html

Interpretación

La clasificación lineal computará la puntuación de una clase como la suma de los pesos de todos sus píxeles a través de sus 3 canales de color (RGB). Dependiendo de los valores que definamos para los pesos (W) la función tendrá la capacidad de ponderar (dependiendo del signo de cada uno de estos pesos) ciertos colores en ciertas posiciones de la imagen. Por ejemplo un barco tendrá más pesos positivos en el canal de color azul (B) y por el contrario pesos negativos en los canales rojo y verde (RG).

Para dar un ejemplo concreto supongamos que tenemos una imagen de 4 pixels (con un solo canal de color) y 3 clases distintas, gato (rojo), perro (verde) y barco (azul). Los pixeles de las imágenes están aplanados en un vector columna. Este vector es multiplicado por la matriz de pesos W para obtener las puntuaciones para cada clase. En este ejemplo la matriz de pesos W no es la óptima ya que como puede verse le asigna una puntuación muy baja a la imagen del gato. En particular la matriz W está convencida de que está buscando un perro.



Función de pérdida (Loss function)

Esta función nos permite medir nuestra "infelicidad" o insatisfacción de las predicciones sobre el conjunto de entrenamiento. Cuanto más grande sea el resultado de la función de pérdida peor será el trabajo de clasificación que estamos realizando. Por lo tanto, cuando más bajo sea el valor de la función de pérdida mejor estaremos haciendo el trabajo de clasificación.

La función de pérdida puede encontrarse también en distintas bibliografías como función de costo (cost function) o función objetivo (objective). A lo largo de este desarrollo nos referiremos a esta por su primer nombre.

La función de pérdida cuantifica qué tan buena o mala es nuestra matriz W. Esta función analiza los puntajes (scores) generados en función de W, de manera de poder determinar que tan bien o mal responde. Debemos encontrar un procedimiento eficiente para buscar los valores de W en todos los posibles espacios de soluciones, hasta encontrar la que mejor

puntuación arroje. Este es un problema de optimización ya que debemos buscar una matriz W que minimice la pérdida. El concepto de optimización será explicado más adelante.

Dado un conjunto de ejemplos **{(x_i, y_i)} i=1...N**. Donde **x_i** representa los píxeles de una imagen e **y_i** es la etiqueta (de tipo integer). Podemos definir a la función de pérdida como,

$$L(W) = \frac{1}{N}\sum_{i}^{N} L_i(f(x_i, W), y_i)$$

A partir de la imagen **x_i** y los pesos **W** se computa la función **f(x_i, W),** la cual estima una puntuación para la etiqueta. Luego se define una función de pérdida **L_i** la cual toma la puntuación predicha (predicted score) y la compara con el valor real (true target) o **y_i**. Esta función devuelve un resultado cuantitativo de que tan mal están las predicciones para el ejemplo de entrenamiento. El resultado final de **L** es el promedio de la sumatoria de todas las pérdidas para todos los ejemplos que conforman el conjunto.

Multiclass Support Vector Machine (SVM)

Es una generalización de binary SVM que soporta múltiples clases.

Sea **i** un elemento del conjunto de imágenes, **x_i** los píxeles de la imagen e **y_i** la etiqueta (la predicción que queremos que haga nuestro algoritmo). La función de puntuación recibe los píxeles de la imagen y computa el vector **f(x_i, W)** con las clases de puntuación, al que llamaremos **s** (score). Tendremos un score **s_j** para la predicción y otro score **s_{yi}** para la clase real (true class), es decir, la clase que deberíamos predecir.

Podemos formalizar la función Multiclass SVM loss para el elemento **i**, como:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

La notación **$j \neq y_i$** representa que se deberán sumar todas las clases menos la clase real (true class) ya que el score para la clase real será igual a 0.

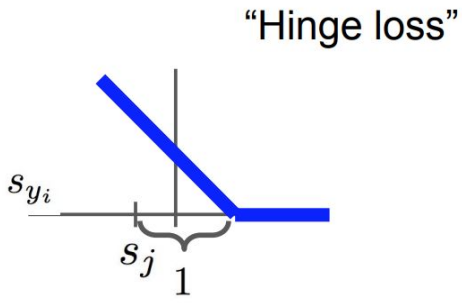
El valor del hiper-parámetro **Δ** es arbitrario y su objetivo es garantizar que el puntaje correcto sea mayor al incorrecto por una diferencia mayor a **Δ**. Si esto no se cumple acumularemos pérdida. Generalmente utilizaremos **Δ = 1.0** por lo que podemos reescribir la función,

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

La pérdida total, para todo el conjunto de ejemplos, será el promedio de la sumatoria de las pérdidas para todos los elementos:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i$$

El límite en cero **max(0, -)** es comunmente llamado **Hinge loss**.



Para resumir el comportamiento de Multiclass SVM daremos un ejemplo concreto. Supongamos que tenemos 3 clases con puntuaciones (scores) **s = [13, -7, 11]** y la primera clase es la clase real (true class). También asumiremos el valor de delta como **Δ = 10**. El cálculo de la pérdida será,

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10)$$

$$L_i = \max(0, -10) + \max(0, 8) = 8$$

El primer término computó una pérdida negativa de -20 (-7 - 13) ya que el score de la clase correcta (13) es mayor al score de la clase incorrecta (-7) por un margen mayor a 10. De hecho la diferencia fue de 20 que es mucho más grande que el delta definido en 10.

El segundo término computó una pérdida de 8, a pesar de que el score para la clase correcta es mayor a la clase incorrecta (13 > 11). Pero la diferencia es de solo 2 y no fué más grande que el margen deseado de 10. Por este motivo la pérdida es de 8.

Si hubiésemos utilizado como valor de delta 1, el resultado hubiera sido el mismo.

Multiclass Support Vector Machine buscará que el score de la clase correcta sea más grande que el resto de los scores por lo menos por un margen mayor o igual a delta **Δ**. Si alguna clase tiene una puntuación dentro de la zona roja (o mayor), entonces acumularemos pérdida. Por el contrario si la puntuación es menor la pérdida será cero. El objetivo será encontrar todos los pesos (weights) de tal manera que todos los ejemplos del conjunto de datos de entrenamiento cumplan con esta premisa, obteniendo una pérdida total lo más baja posible (tienda a cero).

Regularización

La regularización busca resolver uno de los problemas de la función de pérdida. Supongamos que tenemos un conjunto de datos y los parámetros W que clasifican de manera correcta a cada uno de los ejemplos del conjunto ($L_i = 0$ para todos los i). El problema radica en que W puede no ser necesariamente único, es decir, que puede haber W similares que también clasifican todos los elementos del conjunto de manera correcta. Si contamos con un W que clasifica correctamente los ejemplos entonces cada múltiplo de W también lo hará, $\lambda W \ donde \ \lambda > 1$.

Es posible extender la función de pérdida con una penalidad de regularización (regularization penalty) $R(W)$. La regularización más común es la norma **L2** que desalienta grandes pesos a través de una penalización cuadrática sobre todos los parámetros:

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

En la expresión anterior se suman todos los valores cuadrados de W. Vale la pena destacar que la función de regularización está basada solamente en los pesos W. Esta regularización completa la función Multiclass SVM loss ya que ahora estará compuesta por dos componentes, el primero es la pérdida de información (data loss) y el segundo es la pérdida de regularización (regularization loss). Por lo tanto la función Multiclass SVM loss se convierte en:

$$L = \underbrace{\frac{1}{N} \sum_i^N L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

Donde N es el número de ejemplos de entrenamiento y λ es el hiper-parámetro de regularización. No hay una manera sencilla de definir este parámetro, pero como vimos anteriormente es posible determinarlo utilizando validación cruzada. Este hiper-parámetro es uno de los más importantes a tener en cuenta durante el entrenamiento.

Expandiendo la función en su forma completa,

$$L = \frac{1}{N} \sum_i^N \sum_{j \neq y_i} [\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2$$

Hasta este punto, todo lo que debemos hacer es encontrar los pesos que minimicen la pérdida.

Hay otros tipos de regularización que son comúnmente utilizados,

- L1 regularization

- Elastic net (L1 + L2)
- Max norm regularization
- Dropout

Clasificador Softmax (Multinomial Logistic Regression)

En el contexto de Deep Learning, Softmax loss es uno de los más utilizados pese a que la performance entre SVM y Softmax es muy similar.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Optimización

Para explicar el concepto de optimización utilizaremos la analogía con un valle, con montañas, y arroyos. Comenzaremos a caminar y recorrer este valle. Cada punto del valle se corresponde con alguna de las configuraciones de W. Nuestro trabajo será encontrar los caminos que nos llevan cuesta abajo.

El objetivo de la función de optimización es encontrar un W que minimice la función de pérdida (loss function).

Gradiente

En funciones de 1-dimensión la pendiente es la tasa de cambio de la función en un punto de interés. El gradiente es una generalización de la pendiente pero para vectores. Por lo tanto, el gradiente es un vector compuesto por pendientes (también denominadas derivadas) para cada dimensión. Recordemos que la expresión matemática para la derivada de una función en una dimensión es:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Calcularemos derivadas parciales cuando las variables de entrada de la función sean vectores en lugar de números.

Hay dos maneras de calcular el gradiente:

- Gradiente numérico: aproximado, lento y fácil de escribir
- Gradiente analítico: exacto, rápido, propenso a errores (error prone)

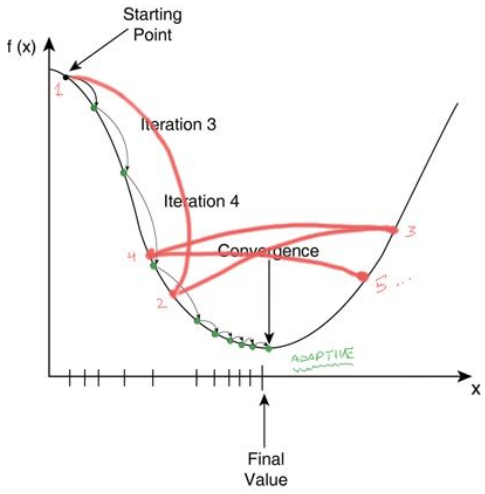
En la práctica siempre utilizaremos el gradiente analítico (analytic gradient) pero verificaremos la implementación con el gradiente numérico (numerical gradient). Esto se denomina, verificación por gradiente (gradient check).

El gradiente nos dará la pendiente de la función de pérdida por cada una de sus dimensiones.

Descenso del Gradiente (Gradient Descent)

Inicializa W con valores aleatorios, luego computa la pérdida, el gradiente y actualiza los pesos en sentido opuesto a la dirección del gradiente. Recordemos que el gradiente apunta en el sentido en que crece la función. Por lo tanto, tomaremos pequeños pasos en sentido opuesto a la dirección del gradiente y repetiremos esto hasta que W converja. El tamaño de los pasos (step size) es un hiper-parámetro y nos dice que tan lejos nos iremos (o que tan grandes serán nuestros pasos) en esa dirección, cada vez que calculemos el gradiente. Llamaremos a este concepto **learning rate**. Este uno de los hiper-parámetros más importantes a configurar al entrenar el modelo. Elegir un learning rate muy chico hará que avance de manera consistente pero lenta. Si por el contrario elegimos uno muy grande, descenderá más rápido pudiendo sobrepasarse.

Como podemos ver en la imagen a continuación[36], un learning rate mas chico (líneas grises) alcanza el objetivo de forma más lenta a través de varias iteraciones. Por el contrario un learning rate muy grande (líneas rojas) se acerca más rápido al objetivo pero sin llegar a converger nunca al verdadero mínimo.



Stochastic Gradient Descent (SGD)

Definimos la pérdida como la ponderación de que tan mal estamos clasificando cada ejemplo. Definiremos también a la pérdida total como el promedio de la pérdida de cada uno de los ejemplos de entrenamiento. Pero en la práctica la cantidad de ejemplos (N) suele ser muy grande y computar esta pérdida no sólo demorará mucho tiempo y sino que será computacionalmente costosa.

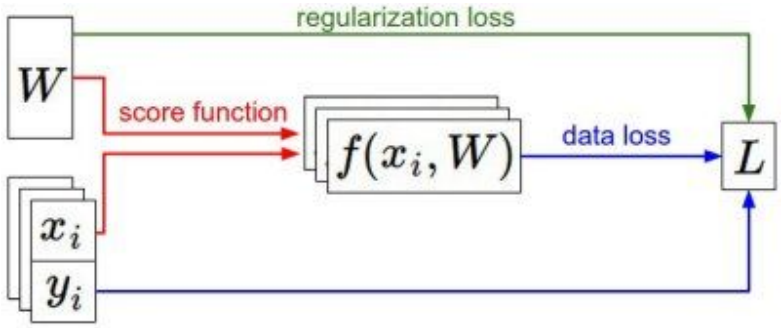
Por este motivo, utilizaremos Stochastic Gradient Descent (SGD) donde en lugar de computar la pérdida y el gradiente sobre todo el conjunto de entrenamiento, tomaremos un conjunto de ejemplos de entrenamiento (**mini-batch**) en cada iteración. Este mini-batch se utilizará para

[36] https://qph.fs.quoracdn.net/main-qimg-f3972c89625c0451f0ea92a4c0ea0728

estimar el valor de la suma total (true gradient). Por convención, el tamaño del mini-batch es una potencia de 2 (32, 64, 128) ya que en la práctica muchas de las operaciones con vectores son más ágiles cuando sus entradas son potencia de 2. Por ejemplo, en "state of the art ConvNets" se utilizan normalmente 256 ejemplos para conformar el mini-batch sobre un conjunto de entrenamiento de unos 1.2 millones de elementos (ILSVRC).

Puntuación, pérdida, regularización y optimización

Para terminar de comprender cómo estos conceptos se complementan, podemos analizar el siguiente diagrama de flujo,



- Se provee de un conjunto de ejemplos expresados por el par **(x, y)**.
- La matriz W (weights) comienza con valores aleatorios y va cambiando.
- La función de puntuación calcula la puntuación de las clases, las cuales son almacenadas en el vector **f**.
- La función de pérdida **L** tiene dos componentes:
    - Data loss: calcula la compatibilidad entre las puntuaciones **f** y las etiquetas **y**.
    - Regularization loss: es solo en función de los pesos **W**.
- Se calculan los gradientes en función de **W** y se utilizan para actualizar los parámetros durante la aplicación de Gradient Descent.

Backpropagation

Este concepto se introdujo por primera vez en la década del 70 pero su importancia no fue apreciada sino hasta 1986 a partir del paper "Learning representations by backpropagating errors"[37] por Rumelhart, D., Hinton, G. & Williams, R.. Es una manera de calcular el gradiente a partir de la aplicación recursiva de la regla de la cadena (chain rule). Recordemos que el gradiente es utilizado para minimizar el error producido por cada neurona.

Imaginemos una red neuronal como una cadena de responsabilidades donde cada nodo es una neurona especializada en una tarea determinada. A su vez, cada nodo afectará el resultado final para bien o para mal. Por lo tanto, debemos analizar toda la cadena de responsabilidades

[37] https://www.nature.com/articles/323533a0

buscando aquellas neuronas que hayan afectado negativamente el resultado. Si encontramos una neurona cuya influencia es negativa, debemos responsabilizarla con parte del error. De esta manera el error se repartirá a cada neurona según el nivel de responsabilidad. El análisis de qué responsabilidad tiene cada neurona en el error tiene sentido realizado hacia atrás, desde la señal de error hasta la primer capa. Esto es así ya que el error de las capas anteriores depende directamente de las capas posteriores. Por lo tanto, mediante la retropropagación (backpropagation) de errores podremos determinar que parte de culpa tiene cada neurona en el error del resultado final.

El algoritmo de backpropagation permite determinar el error generado por cada neurona con un único pase hacia adelante y otro hacia atrás. Estos errores son los que utilizaremos para calcular las derivadas parciales de cada parámetro de la red, conformando así el vector gradiente.

Para comprender de manera gráfica backpropagation podemos analizar la representación de una función utilizando su gráfico computacional (computational graph).

Computational Graph

Podremos utilizar estos gráficos para representar cualquier función, donde los nodos del grafo estarán definidos por las operaciones de la función.

Dada la función,

$$f(x,y,z) = \underbrace{(x+y)}_{\text{q}} z = q.z$$

La misma puede ser desglosada en las siguientes expresiones,

$$q = x + y \qquad\qquad f = q.z$$

Utilizando estas expresiones podemos computar sus derivadas de manera simple,

$$\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1 \qquad\qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Para **x** e **y** es distinto ya que no están "conectados" directamente con **f** sino que esto se hace a través de un nodo intermedio **q**. Por este motivo debemos aplicar la regla de la cadena (chain rule),

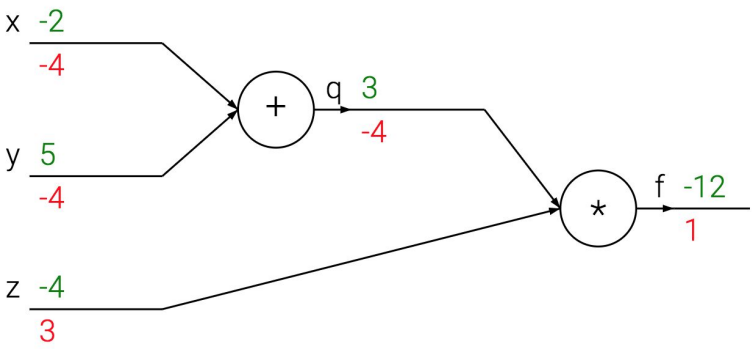
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \qquad\qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z$$

No estamos necesariamente interesados en los gradientes intermedios, como el de **q**. Por el contrario nos enfocaremos en aquellos gradientes de **f** con respecto a sus variables de entrada, **x**, **y**, **z**.

Para poder dar una representación más visual asignaremos valores a las variables de entrada y armaremos su gráfico computacional. Haremos un "forward-pass" utilizando los valores de entrada para computar la salida de la función. Supongamos que,

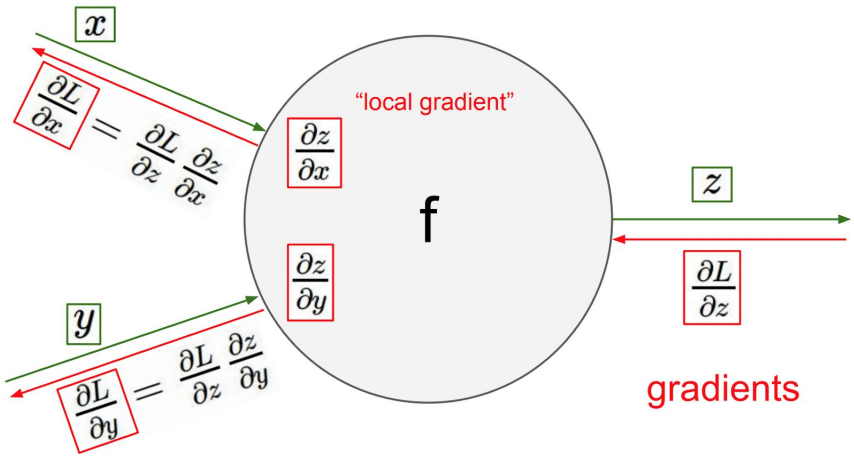
$$x = -2; y = 5; z = -4$$

En verde se representan los valores calculados para obtener la salida de la función. En rojo los resultados de los cálculos de las derivadas y por lo tanto los gradientes.



$$\frac{\partial f}{\partial x} = z = -4 \qquad \frac{\partial f}{\partial y} = z = -4 \qquad \frac{\partial f}{\partial z} = q = x + y = 3 \qquad \frac{\partial f}{\partial q} = z = -4$$

Es posible agrupar los nodos en uno solo, de mayor complejidad, siempre que sea posible determinar el gradiente local (local gradient).



Métricas para evaluar modelos de clasificación

La precisión (accuracy) es una buena medida cuando las distintas clases del conjunto de datos están equilibradas ya que nos dará la proporción entre las predicciones correctas que ha hecho el modelo sobre el total de predicciones. Sin embargo, cuando las distintas clases están desequilibradas no es posible distinguir los errores del tipo falso positivo y falso negativo. Podemos definir la precisión como el equivalente de restar el error de una unidad: ***1 - Error***.

$$\text{Precisión} = Accuracy = \frac{clasificaciones\,correctas}{total\,de\,observaciones} = 1 - Error$$

La precisión es la métrica más común y a la que se le suele dar mayor importancia pero no es suficiente para entender cómo se comportará el modelo frente a casos desconocidos.

Matriz de Confusión (Confusion matrix)

La Matriz de Confusión[38] es utilizada para tener una visión más completa durante el análisis de la performance de un modelo. Es una de las métricas más intuitivas y sencillas que se utiliza para evaluar la precisión y exactitud del modelo. La matriz comparará el valor de verdad (actual class) de las etiquetas con el valor predicho (predicted class) para cada una de ellas. Podemos definir la matriz de confusión como:

		Predicted class	
		+	-
Actual class	+	**TP** True Positives	**FN** False Negatives Type II error
	-	**FP** False Positives Type I error	**TN** True Negatives

- **Verdaderos Positivos** (True Positives - TP): los datos reales son 1 (Verdadero) y la predicción también es 1 (Verdadero).
- **Verdaderos Negativos** (True Negatives - TN): los datos reales son 0 (Falso) y el pronóstico también es 0 (Falso).
- **Falsos Positivos** (False Positives - FP): los datos reales indican que es 0 (Falso) y la predicción indica que es 1 (Verdadero), es decir, la predicción ha sido errónea (Type I error).

[38] https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks

- **Falsos Negativos** (False Negatives – FN): los datos reales indican que es 1 (Verdadero) y el pronóstico es 0 (Falso), es decir, la predicción ha sido errónea (Type II error).

Sensibilidad: también llamada en inglés, **Recall** o **True Positive Rate**. Nos da la probabilidad de que, dada una observación realmente positiva, el modelo la clasifique así. Nos provee información de la performance con respecto a los Falsos Negativos, por lo que si queremos minimizarlos debemos enfocarnos en esta métrica.

$$Sensibilidad = Recall = TPR = \frac{TP}{TP + FN}$$

Especificidad: también llamada en inglés Specificity o tasa de verdaderos negativos. Nos da la probabilidad de que, dada una observación realmente negativa, el modelo la clasifique así.

$$Especificidad = \frac{TN}{TN + FP}$$

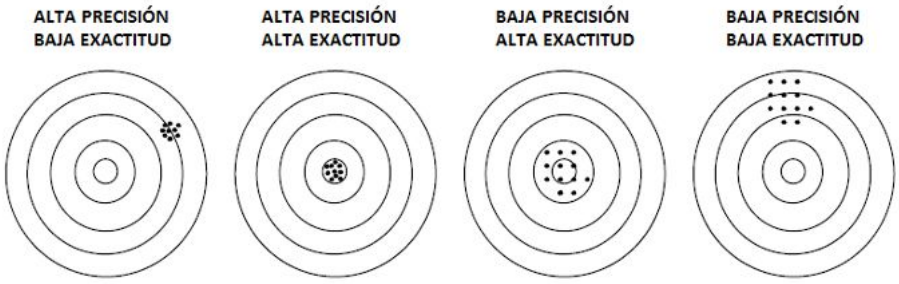
Exactitud: también llamada en inglés, **Precision** o valor de predicción positiva. Nos da la probabilidad de que, dada una predicción positiva, la realidad sea positiva también. Nos provee información de la performance con respecto a los Falsos Positivos, por lo que si queremos minimizarlos debemos enfocarnos en esta métrica.

$$Exactitud = Precision = \frac{TP}{TP + FP}$$

Tasa de Falsos Positivos (False Positive Rate):

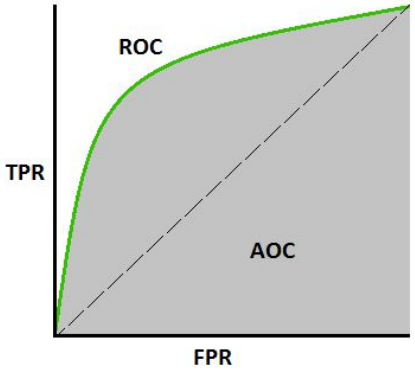
$$FPR = 1 - Especificidad = \frac{FP}{TN + FP}$$

Generalmente surge la confusión entre las métricas Precisión y Exactitud ya que podemos tener modelos que sean exactos y no precisos o viceversa. En el siguiente gráfico podemos ver una comparativa entre ambas métricas.



Curva ROC (Receiver Operating Characteristics)

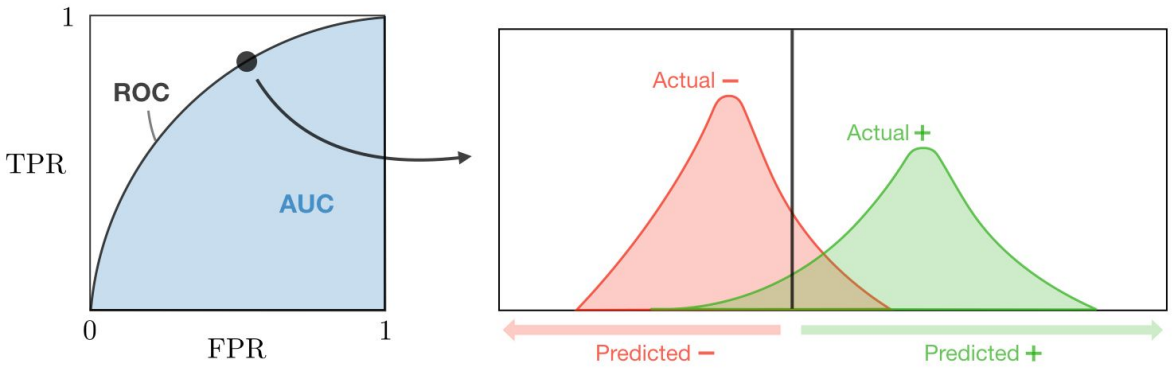
Muestra el rendimiento de un modelo de clasificación. Esta curva representa la comparación entre el True Positive Rate (TPR) y el False Positive Rate (FPR).



AUC: Área bajo la curva ROC

Representa la medida o grado de separación y nos dirá que tan capaz es el modelo de distinguir entre clases. A mayor AUB mejor será el modelo prediciendo 0 como 0 y 1 como 1. Cuanto más se acerca la curva a 1 mejores serán las predicciones y por analogía, cuando más se acerque la curva a 0 peores serán las predicciones.

- AUC = 1, las predicciones son perfectas
- AUC = 0.5, es el peor de los escenarios ya que no podrá discriminar entre clases
- AUC = 0, el modelo predecirá los casos negativos como positivos y viceversa.



Neural Networks

Al hablar de redes neuronales la primer analogía que hacemos es con las neuronas de nuestros cerebros. De hecho, las redes neuronales que desarrollaremos a continuación están inspiradas en las nuestras, pero dada su complejidad estamos aún muy lejos de que se parezcan. Sin embargo, podemos introducir las redes neuronales sin hacer una analogía directa a nuestros cerebros.

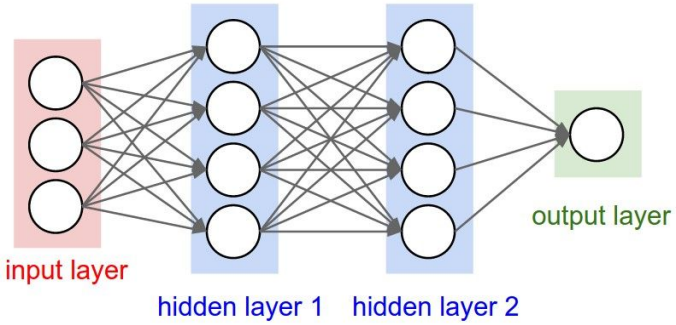
Definiremos una neurona como la unidad básica de procesamiento. Las neuronas tienen conexiones de entrada a través de las que reciben estímulos externos, que llamaremos valores de entrada (inputs). Con estos valores las neuronas realizarán un cálculo interno y generarán un valor de salida (output). Matemáticamente podemos interpretar una neurona como una función.

Internamente la neurona utiliza todos los valores de entrada para realizar una suma ponderada. La ponderación de cada una de sus entradas viene dada por el peso (W_i) que se le asigna. Cada conexión que llega a la neurona tendrá asociada un valor que servirá para definir con qué intensidad cada variable de entrada afecta la neurona. Esto modificará positiva o negativamente el valor de salida.

Una neurona puede modelarse de tal manera que se comporte como una compuerta AND o una OR. Pero no sucede lo mismo con una compuerta XOR ya que no es posible representarla separando linealmente una nube de puntos. Para solucionar este problema debemos combinar las neuronas, de forma que puedan separar ambas clases, es decir, tendremos 2 líneas rectas. Esto demuestra que es posible representar información más compleja combinando neuronas.

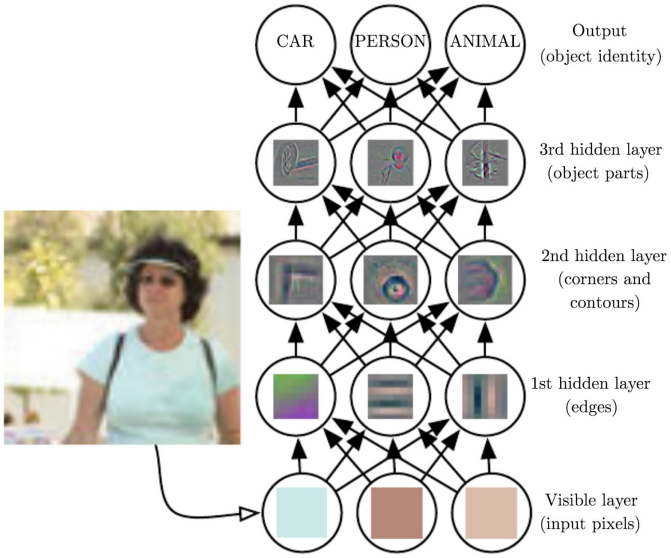
Podremos combinar neuronas de distintas maneras:

- En la misma capa (layer), recibirán la misma información de entrada de la capa anterior y los cálculos realizados pasarán a la siguiente capa.
    - Capa de Entrada (Input layer): la primera capa.
    - Capa oculta (Hidden layer): capas intermedias.
    - Capa de Salida (Output layer): la última capa.
- En forma secuencial, recibirán la información procesada por la neurona anterior.



La ventaja de poder combinar las capas es que ahora las redes podrán aprender conocimiento jerarquizado. Las primeras capas aprenderán los conceptos más básicos y las capas subsiguientes conceptos o características más específicas, es decir, que entre más capas añadimos más complejo será ser el conocimiento. Esta profundidad de capas se conoce como **Deep Learning o Conocimiento Profundo**.

Es complejo para una máquina comprender el sentido de la información que le proveemos. Deep Learning resuelve este problema partiendo la complejidad en pequeños problemas, a través de distintos layers, donde cada uno aprende un conjunto de características. La entrada (input) de la red contiene variables que representan la información que queremos observar. Luego tenemos las capas ocultas (hidden layers) que extraen de manera incremental las propiedades o características (features) de la información que le presentamos. El modelo, a través de estas capas ocultas, es quién deberá determinar qué conceptos son relevantes para poder explicar la relación que tiene la información observada. Dada una imagen el primer layer podrá identificar los bordes (edges), el 2do layer podrá identificar conceptos como contornos y esquinas (corners and contours) y el 3er layer podrá identificar partes de objetos (object parts). Todo esto se hará de manera incremental en cada uno de los distintos layers hasta llegar al último layer que mapeara la salida de los layers anteriores en un resultado concreto (vector de activaciones).



El trabajo que realiza cada una de estas neuronas se puede reducir a un problema de regresión lineal. Matemáticamente estaríamos concatenando varias operaciones de regresión lineal. El problema que tiene esto es que el efecto de sumar muchas operaciones de regresión lineal (o líneas rectas) equivale a solamente haber hecho una única operación, es decir, da como

resultado una línea recta. Este mismo problema visto desde el concepto de una red neuronal hace que toda la estructura colapse a una única neurona. Para solucionar este problema debemos hacer que la suma resulte en algo distinto a una línea recta y para eso necesitamos que cada recta sufra algún tipo de deformación no lineal. La aplicación de esta deformación se denomina **función de activación**.

La aplicación de una función de activación y la combinación de neuronas, hará que podamos representar estructuras de datos complejas. Matemáticamente se puede expresar como la intersección de distintas superficies.

Retomando el concepto de clasificación lineal, donde computamos las puntuaciones para las distintas categorías utilizando la fórmula **s = Wx**, donde **W** es una matriz y **x** el vector columna que contiene todos los píxeles de la imagen que queremos procesar. Podemos extender este concepto para definir una red neuronal (neural network) calculando las puntuaciones como,

$$s = W_2 \max(0, W_1 x)$$

Donde, como podemos ver, la salida de una neurona será la entrada de la neurona siguiente y por lo tanto tendremos una red de 2 capas (layers). Encontramos también, la función **max()** que se aplica a todos los elementos de salida de la primer neurona. Esta función, es la función de activación y matemáticamente se conoce como ReLU. Es comúnmente utilizada para definir los límites de activación haciendo que todas aquellas estimaciones menores a cero sean igual a cero. Los parámetros $\mathbf{W_1}$ y $\mathbf{W_2}$ se aprenden utilizando el algoritmo de Stochastic Gradient Descent (SGD) y sus gradientes se derivan de la aplicación de la regla de la cadena (calculados utilizando backpropagation).

Siguiendo la misma lógica, una neural network de 3-layers se define como,

$$s = W_3 \max(0, W_2 \max(0, W_1 x))$$

En este ejemplo los parámetros que debemos aprender serán: $\mathbf{W_1}$, $\mathbf{W_2}$, y $\mathbf{W_3}$.

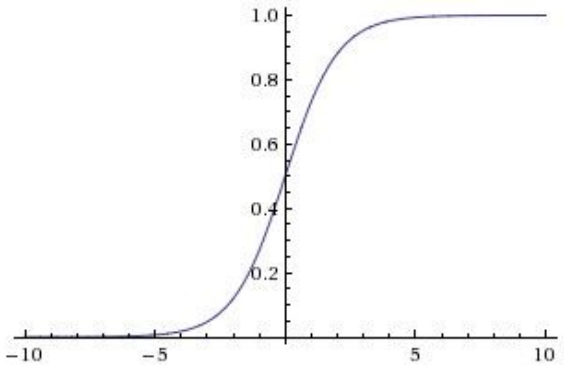
Funciones de activación

Las funciones de activación distorsionan el valor de salida de una neurona añadiendo deformaciones no lineales. De esta manera, podemos encadenar de forma efectiva la computación de varias neuronas. Las deformaciones dependerán del tipo de función de activación que se elija.

Sigmoid

La distorsión que produce hace que los números grandes converjan a 1 y los valores pequeños a 0. Con esta función también podemos representar probabilidades ya que sus valores van en el rango de 0 a 1.

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

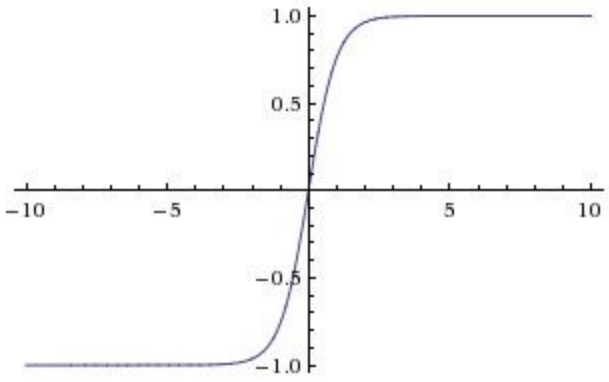


Generalmente utilizaremos este tipo de activación cuando queremos obtener una probabilidad en modelos con múltiples clases.

Tanh

La tangente hiperbólica tiene forma similar a Sigmoid, pero sus valores se mueven en el rango del -1 al 1.

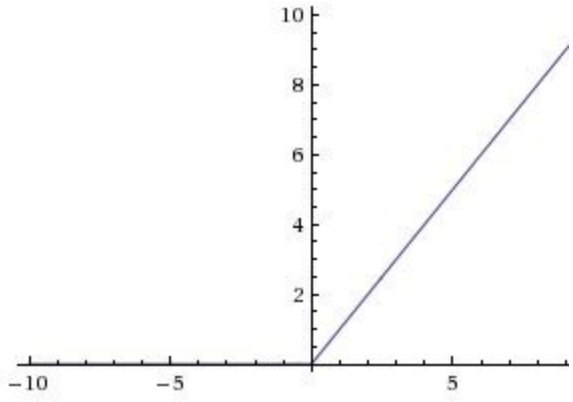
$$f(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



ReLU (Rectified Linear Unit)

La Unidad Rectificada Lineal se comporta como una función lineal cuando es positiva y constante a 0 cuando el valor de entrada es negativo.

$$f(x) = \max(0, x)$$

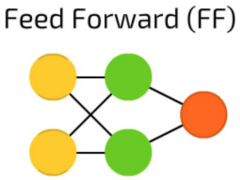


Generalmente utilizaremos este tipo de activación cuando la salida es binaria.

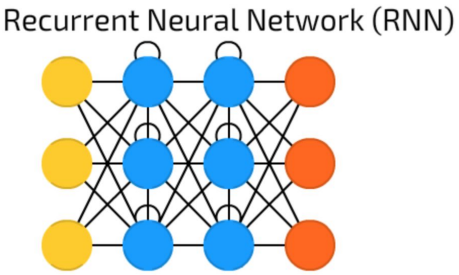
Arquitecturas de Redes Neuronales

Las arquitecturas más comunes que podemos encontrar son:

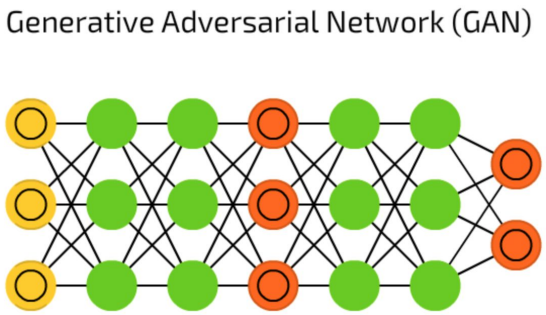
- **Perceptron**: la neurona más básica, utilizada en algoritmos supervisados de clasificación binaria.
- **Feed Forward Network (FF)**: También llamadas multilayer perceptrons (MLPs), porque podemos verlas como un conjunto de Perceptrons. Se utilizan en problemas de clasificación binaria. La información fluye en una única dirección desde la entrada a la salida.



- **Recurrent Neural Network (RNN):** Redes neuronales con "memoria" cuyas decisiones están influenciadas por lo aprendido en el pasado. Las Long/Short Term Memory (LSTM) son un tipo especial de las RNNs. Generalmente utilizadas para Natural language processing (NLP) por su capacidad de memoria lo que permite, a partir del análisis de las secuencias anteriores predecir las subsiguientes (traducción de textos, generación de textos, etc)



- **Generative Adversarial Network (GAN)**: Redes que son capaces de aprender la estructura de los datos para generar nuevos datos. A partir de los datos de entrada podrán generar nuevos datos de salida. Este modelo cuenta con 2 tipos de redes distintas en la cual la primera se encargará de aprender a generar datos y la segunda se encargará de aprender a evaluar si los datos son verdaderos o generados por la primer red. De ahí su nombre de redes adversarias. La primer red deberá aprender a "engañar" a la segunda red, haciendo que los datos generados sean cada vez mejores.



- **Convolutional Neural Network (CNN)**: Neuronas utilizadas para el procesamiento de imágenes y serán las neuronas que desarrollaremos a continuación.

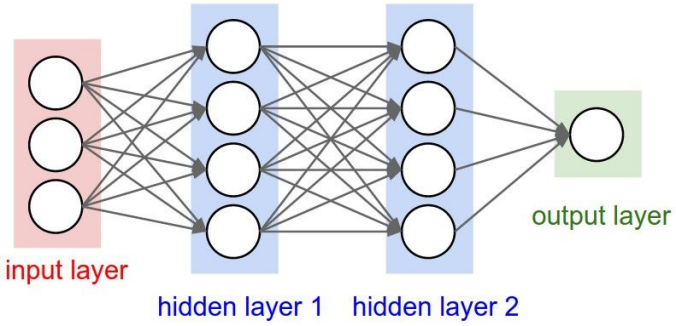
Convolutional Neural Networks (CNNs / ConvNets)

Las redes convolucionales o ConvNets son un tipo de red neuronal utilizada para el procesamiento de imágenes. Estas redes reciben el nombre de convolucionales ya que emplean una operación matemática llamada convolución. La convolución es un tipo de operación lineal. Las ConvNets son aquellas redes neuronales que utilizan la operación de convolución en, por lo menos, alguna de sus capas.

La diferencia entre las Neural Networks regulares y las ConvNets es que estas últimas asumen que las entradas (inputs) de la red serán imágenes, lo que permite que ciertas propiedades puedan ser codificadas en la arquitectura de la red. A su vez, esto permite que la función de avance (forward function) sea más eficiente y reduce el número de parámetros de la red.

Arquitectura de una ConvNet

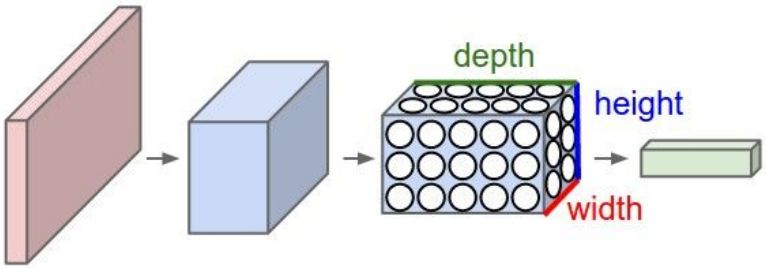
Cómo vimos anteriormente las Neural Networks reciben una entrada (un único vector) que es transformada a través de una serie de capas ocultas (hidden layers). Cada una de las capas ocultas está compuesta por un conjunto de neuronas, donde cada neurona está conectada a todas las neuronas de la capa anterior (fully-connected). Las neuronas de una capa funcionan independientemente unas de otras y no comparten ninguna conexión. La última capa (fully-connected) es llamada "capa de salida" (output layer) y en configuraciones de clasificación representa las puntuaciones de la clase de salida.



El problema de estas redes es que no escalan adecuadamente con imágenes. Recordemos el ejemplo de CIFAR-10, donde las imágenes tienen un tamaño de 32x32x3 (ancho, alto y 3 canales de color). La primer capa oculta tendrá 32*32*3 = 3072 pesos (weights). Si extendemos el tamaño de la imagen a procesar, la cantidad de pesos será significativamente mayor.

Las Convolutional Neural Networks dan por hecho que la entrada será una imagen y modelan la arquitectura de manera diferente. Las capas de estas redes se adecuan a la estructura de una imagen y tendrán neuronas dispuestas en 3 dimensiones, representadas por el ancho, la altura y la profundidad (width, height, depth). El objetivo de estas capas es preservar el espacio dimensional (spatial size) de las imágenes. Retomando el ejemplo de CIFAR-10, las imágenes

de entrada estarán representadas por un volumen de entrada de activaciones cuya dimensión será 32x32x3 (width, height, depth). Las neuronas de una capa solo estarán conectadas a una pequeña región de la capa anterior. Además, la capa de salida tendrá una dimensión de 1x1x10 ya que al final de la red reduciremos la imagen a un único vector de clases de puntuación dispuestas a lo largo de la dimensión de profundidad.



Layers

Las ConvNets están conformadas por distintas capas (layers) agrupadas de manera secuencial, en donde cada layer transforma un volumen de activaciones en otro a través una función diferencial. Dentro de las arquitecturas de las ConvNets podemos encontrar 3 tipos de layers:

- **Convolutional Layer**
- **Pooling Layer**
- **Fully-Connected Layer** (el mismo que encontramos en las Neural Networks regulares).

Estos layer podrán ser apilados (stack) para formar una ConvNet y conceptualmente los podemos imaginar como "building blocks".

Una arquitectura común de ConvNet estará compuesta por distintos CONV-RELU layers, seguidos de un POOL layer, repitiendo este patrón hasta que la imagen se fusione espacialmente en una de menor tamaño. El último FC (fully-connected) layer almacenará la salida (generalmente las clases de puntuación).

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

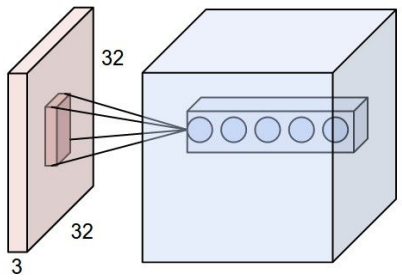
Convolutional Layer (CONV)

Es el layer que realiza la mayor parte del trabajo computacional en la red. Los parámetros de esta capa consisten en un conjunto de filtros que aprenden. Estos filtros son de pequeñas dimensiones tanto a lo ancho como a lo alto (width y height) y se extienden a través de toda la profundidad del volumen de entrada. Por ejemplo, en CIFAR-10 tendremos una imagen de

entrada de 32x32x3 y las dimensiones del filtro de 5x5x3, donde 5 pixeles representan tanto el ancho como alto y el 3 representa los distintos canales de color (la profundidad se conserva).

Durante la recorrida "hacia adelante" de la red (forward pass), cada filtro recorrerá la imagen a través de toda su superficie (ancho y alto), calculando el producto escalar (dot product) entre las entradas del filtro y las distintas posiciones de entrada de la imagen. A medida que el filtro se desplaza sobre el ancho y alto del volumen de entrada se producirá un mapa de activaciones (activation map) de 2-dimensiones.

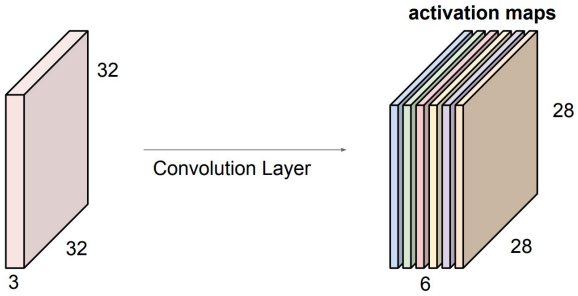
Cuando trabajamos con entradas (inputs) de múltiples dimensiones, como es el caso de las imágenes, no es práctico ni escalable conectar las neuronas a todas las neuronas del volumen anterior. Lo que haremos será conectar cada neurona sólo a una región local del volumen de entrada. La extensión espacial (spatial extent) de esta conectividad es un hiper-parámetro llamado campo receptivo (receptive field) de una neurona y es equivalente al tamaño del filtro. Este hiper-parámetro será representado por la letra F (spatial extent filter). Es importante destacar la asimetría en cómo trataremos las dimensiones espaciales (width y height) y la dimensión de profundidad (depth): las conexiones serán locales en espacio, pero completas en profundidad.



Hasta este punto hemos explicado la conectividad entre las neuronas y el volumen de entrada pero todavía no vimos cuántas neuronas tendremos en el volumen de salida ni cómo estarán organizadas. Existen 3 hiper-parámetros que controlarán el tamaño del volumen de salida y son: **Depth, Stride** y **Zero-padding.**

Depth (K)

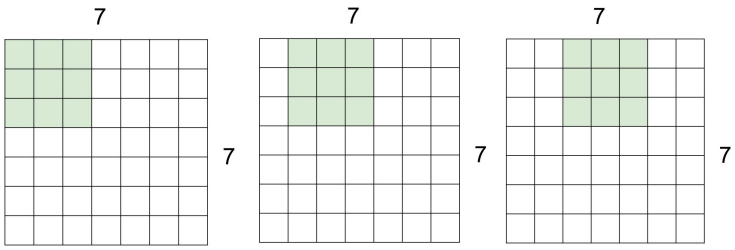
Es un hiper-parámetro que controla la profundidad del volumen de salida a partir del número de filtros que usaremos. Definiremos una columna de profundidad (depth column) para el conjunto de neuronas que observan la misma región de entrada de la imagen.



La imagen muestra la salida luego de aplicar 6 filtros de 5x5, dando como resultado 6 mapas de activaciones (activation maps) de 28x28x1 cada uno.

Stride (S)

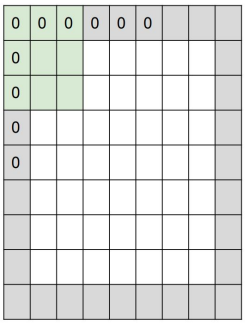
Determina los pasos que dará el filtro cuando se deslice por la imagen. Cuando el stride es 1, el filtro se moverá de a 1 pixel. Cuando el stride sea de 2 entonces el filtro saltará 2 pixeles cada vez que se mueva. Esto producirá volúmenes de salida más pequeños espacialmente.



Las imágenes muestran los primeros 3 movimientos de un filtro de 3x3 con un stride de 1.

Zero-padding (P)

Hay veces que será conveniente rellenar el volumen de entrada con ceros alrededor del borde. El tamaño de este borde será un hiper-parámetro y nos permitirá controlar el tamaño espacial de los volúmenes de salida.



La imagen muestra el comportamiento de zero pad de 1 con un input de 7x7, un filtro de 3x3 y un stride de 1.

Resumiendo, el comportamiento de los CONV layers, el mismo estará definido por los siguientes hiper-parámetros:

- K: número de filtros.

- F: dimensiones del filtro (spatial extent).
- S: stride (movimientos del filtro).
- P: el número de zero-padding.

La aplicación de un CONV layer a un input de dimensiones W_1 x H_1 x D_1 producirá un output de dimensiones W_2 x H_2 x D_2, donde:

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

$$D_2 = K$$

Los pesos (weights) por filtro están definidos por F * F * D_1, con un total de (F * F * D_1) * K.

Ejemplo

Continuando con el ejemplo de CIFAR-10, para una imagen de 32x32x3, con 10 filtros de 5x5, stride 1 y zero-padding 2.

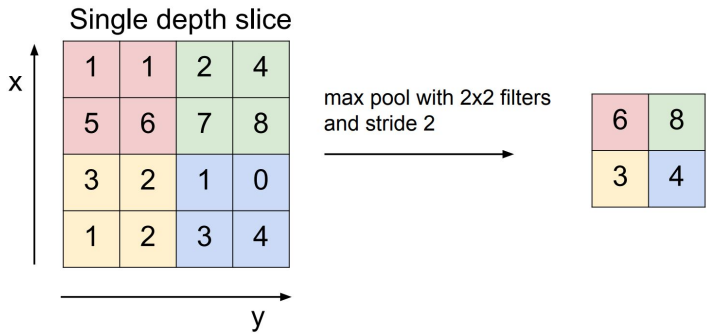
$$W_2 = H_2 = \frac{32 - 5 + 2 * 2}{1} + 1 = 32$$

Por lo tanto la salida será de 32x32x10. A su vez podemos determinar la cantidad de parámetros que tendrá este layer. Cada filtro tendrá 5*5*3 + 1 = 76 parámetros (+1 por el bias). Sabiendo que tenemos 10 filtros, 76*10 = 760 parámetros en total.

Pooling Layer (POOL)

El Pooling Layer (POOL) es una operación de reducción (downsampling) que generalmente se aplica luego de una operación de convolución. Estos layers hacen que las representaciones sean más pequeñas y más manejables. Podemos destacar dos tipos de de operación de pooling:

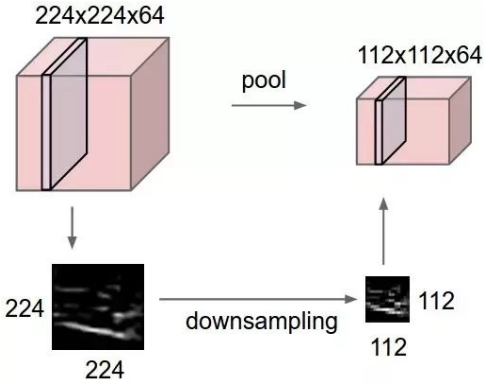
- **Max Pooling:** aplicará la función max() al filtro.
- **Average Pooling:** aplicará la función avg() al filtro.



Aplicación de Max Pooling con un stride de 2.

Resumiendo, un Pooling Layer,

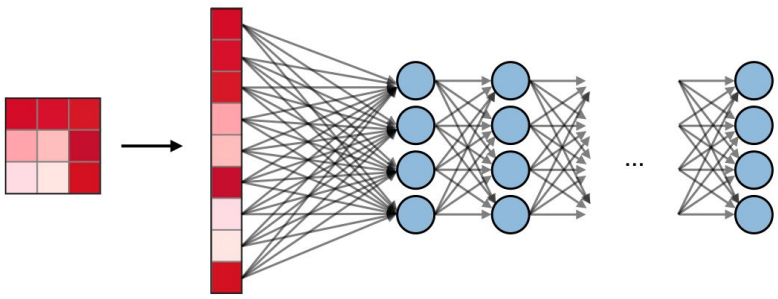
- Transforma la representación en algo más pequeño y manejable (downsampling).
- Opera independientemente sobre cada mapa de activación (activation map).



Downsampling de una imagen de 224x224x64 con un filtro de 2x2 y stride 2, en otra de 112x112x64.

Fully Connected Layer (FC)

Contiene neuronas que se conectan a todo el volumen de entrada (input volume) como en las Neural Networks regulares. Se las suele encontrar al final de las CNN y se utilizan para optimizar el objetivo, como las clases de puntuación (class scores).



Arquitecturas Convolucionales

Las arquitecturas convolucionales más comunes son:

- **LeNet**: La primer aplicación exitosa de una ConvNet desarrollada por Yann LeCun[39] en los 90's. La implementación más conocida es la que se utilizó para interpretar los números de los dígitos postales.

- **AlexNet**: El primer trabajo que hizo populares a estas redes en el campo de Computer Vision. Desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoff Hinton. Esta arquitectura terminó primera en la competencia ImageNet ILSVRC que se llevó a cabo en el 2012. Esta implementación es similar a LeNet pero mas profunda, grande y contaba con capas convolucionales apiladas una encima de la otra (anteriormente era común tener solo una capa CONV seguida inmediatamente una otra capa POOL).

- **ZFNet**: Esta arquitectura fué la ganadora del ImageNet ILSVRC 2013. Desarrollada por Matthew Zeiler y Rob Fergus. Es conocida como ZFNet[40] por las iniciales de sus autores (Zeiler & Fergus Net). La arquitectura está basada en una mejora realizada a AlexNet a partir de un ajuste en los hiper-parámetros, en particular, expandiendo el tamaño de las capas convolucionales intermedias y reduciendo el tamaño del filtro y el stride de la primer capa.

- **GoogLeNet (Inception)**: Arquitectura ganadora del ImageNet ILSVRC 2014, desarrollado por Szegedy et al.[41] de Google. La contribución más importante de esta arquitectura fue la introducción de un "Inception Module" que redujo drásticamente el número de parámetros de la red (4M comparado con los 60M de AlexNet). Adicionalmente la implementación utiliza Average Pooling en lugar de Fully Connected layers, eliminando gran cantidad de parámetros innecesarios. A esta primer versión, desarrollada en el 2014, le han seguido varias actualizaciones más, con la última lanzada en el 2016 llamada "Inception-v4"[42].

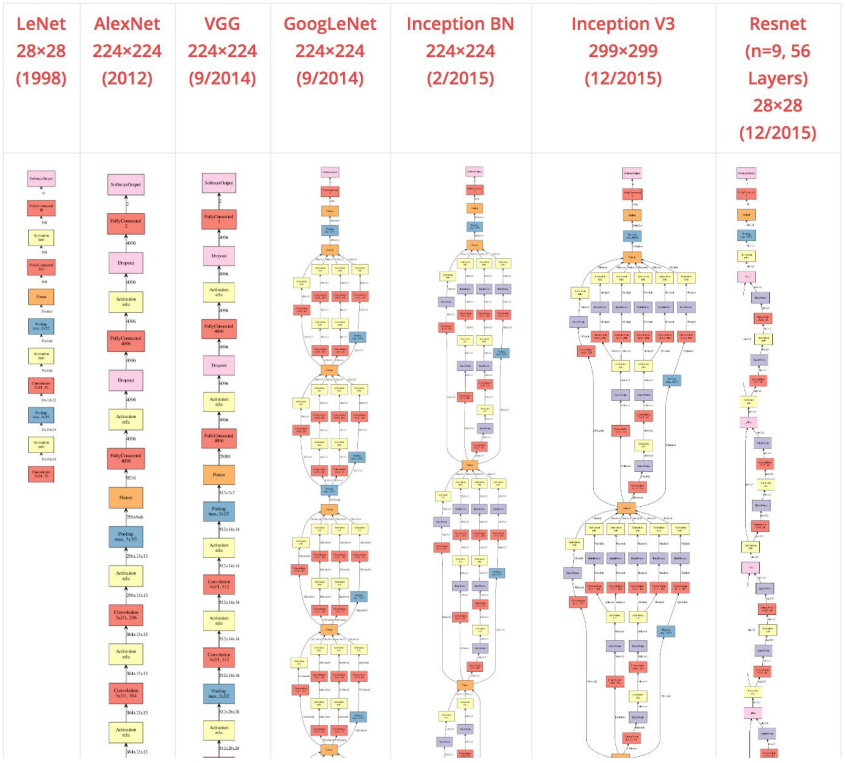
- **VGGNet**: Esta arquitectura también fue revolucionaria en la competencia ILSVRC 2014 obteniendo el segundo puesto, detrás de GoogLeNet. Los autores de la red son Karen Simonyan and Andrew Zisserman[43] de la universidad de Oxford y demostraron que la profundidad de una red es un componente crítico para una buena performance. La mejor implementación de esta red tiene 16 capas CONV/FC y presenta una arquitectura extremadamente homogénea que solo realiza convoluciones de 3x3 y pooling de 2x2,

[39] http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf
[40] http://arxiv.org/abs/1311.2901
[41] http://arxiv.org/abs/1409.4842
[42] https://arxiv.org/abs/1602.07261
[43] http://www.robots.ox.ac.uk/~vgg/research/very_deep/

desde el principio hasta el final. El problema de VGGNet es que es más costosa de evaluar ya que usa más memoria y cuenta con 140M de parámetros.

- **ResNet**: Residual Network[44] desarrollado por Kaiming He et al. y fué la arquitectura ganadora del ILSVRC 2015. Hace uso intensivo de batch normalization y skip connections. Esta arquitectura también omite los fully connected layers al final de la red. En la práctica esta arquitectura se considera el state-of-the-art en las ConvNets.

Joseph Paul Cohen comparte en su blog el artículo "Visualizing CNN architectures side by side with mxnet"[45] donde puede visualizarse claramente una comparativa entre estas arquitecturas y la cantidad de capas que tiene cada una. No es posible adjuntar dicha visualización en este trabajo ya que llevaría unas cuantas páginas pero a continuación se adjunta una imagen donde puede verse la magnitud de lo que representa cada arquitectura.



[44] http://arxiv.org/abs/1512.03385
[45] https://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/

Transfer Learning

Se desarrollará el concepto de transferencia de aprendizaje (Transfer Learning) para las Convolutional Networks ya que son las redes neuronales que utilizaremos frecuentemente para el análisis de imágenes. De todas maneras, esta técnica puede ser utilizada también para cualquier otro tipo de red neuronal.

En la práctica muy poca gente entrena una Convolutional Network desde cero (utilizando inicializaciones aleatorias) ya que es muy raro contar con un dataset del tamaño apropiado y normalmente entrenar una ConvNet puede demorar por lo menos entre 2 y 3 semanas utilizando varios GPUs y por ende suele ser caro también. Por este motivo se suelen utilizar modelos pre-entrenados a partir un gran conjunto de datos (como por ejemplo ImageNet[46]). Los modelos pre-entrenados pueden utilizarse como inicializadores del nuevo modelo o para extraer las características fijas (fixed feature extractor) del nuevo dataset.

Los tres grandes escenarios donde se utiliza la técnica de Transfer Learning son:

- **ConvNet as fixed feature extractor**: Tomemos una ConvNet entrenada utilizando ImageNet, debemos remover el último fully-connected layer. La salida de este layer será el conjunto de puntuaciones (scores) de las 1000 clases. Luego podremos tratar el resto de la ConvNet para extraer las características (fixed feature extractor) del nuevo conjunto de datos. Llamaremos a estas características (features), **CNN codes**. Es importante para la performance del modelo que estos códigos tengan un límite inferior en cero utilizando ReLU. Una vez extraídos los códigos para todas las imágenes entrenaremos un clasificador lineal (Linear SVN o Softmax) para el nuevo conjunto de datos.

- **Fine-tuning the ConvNet**: En esta estrategia no solo se remplazará y volverá a entrenar el clasificador por encima de la ConvNet en el nuevo dataset sino que también se hará fine-tuning de los pesos de la red pre-entrenada utilizando backpropagation. Se podrá realizar fine-tuning de todos los pesos o también se podrán dejar los primeros layers fijos (para evitar overfitting) aplicando solo fine-tuning a los layers de mayor nivel. Esto está motivado a que los primeros layers aprenden características más genéricas y los últimos layer aprenden características más específicas.

- **Pretrained models**: Debido al tiempo que lleva entrenar una ConvNet es común compartir los modelos entrenados para luego ser extendidos a conjuntos más específicos.

[46] http://www.image-net.org/about-overview

Cómo y cuándo realizar fine-tuning?

Hay varios factores que influirán directamente en esto pero los dos más importantes son:

- El tamaño del nuevo conjunto de datos (pequeño o grande).
- La similitud del nuevo conjunto de datos con el conjunto original.

En función de estos dos factores podemos distinguir cuatro escenarios posibles:

1. **El nuevo dataset es más chico y similar al dataset original:** Debido a que el nuevo dataset es pequeño no es buena idea realizar fine-tuning de la ConvNet para no caer en overfitting. Ya que los ejemplos son similares a los originales, esperaríamos que los layers de mayor nivel sean relevantes al nuevo conjunto. Por lo tanto la mejor idea será entrenar un clasificador lineal en los CNN codes (removiendo el último layer).
2. **El nuevo dataset es más grande y similar al dataset original:** Ya que contamos con más información podremos asegurarnos que no caeremos en overfit si intentáramos realizar fine-tuning sobre toda la red.
3. **El nuevo dataset es más chico y muy distinto al dataset original:** Debido a que el nuevo dataset es pequeño la mejor idea será entrenar un clasificador lineal. El problema es que al contar con ejemplos muy diferentes con respecto al dataset original el modelo contará con características relevantes al dataset original. Para esto será conveniente entrenar un clasificador SVM desde las activaciones de los primeros layers de la red.
4. **El nuevo dataset es más grande y muy distinto al dataset original:** Ya que el nuevo dataset es más grande podremos entrenar la ConvNet desde cero. De todas maneras, en la práctica se utilizan los pesos (weights) del modelo pre-entrenado para inicializar el nuevo modelo. En este caso tendremos suficiente información y seguridad para realizar fine-tuning de toda la red.

Implementación

FaceNet

Introducción

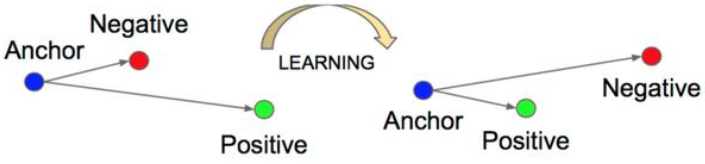
El paper de FaceNet[47] propone una solución al problema de verificación y reconocimiento de rostros. Para esto los autores utilizan una arquitectura de redes convolucionales (CNNs) que aprenden a mapear un rostro en un espacio Euclidiano. Una vez producido este espacio, tanto el reconocimiento, la verificación y el clustering de rostros puede ser fácilmente implementado a través de un algoritmo de clasificación utilizando los "embeddings" como vectores (feature vectors).

Conceptualmente podemos entender un "embedding" como el mapeo de los atributos de entrada (input features) a un vector. En este caso el atributo de entrada del modelo será la representación matricial de un rostro y la salida un vector numérico. Calculando la distancia vectorial podremos identificar qué tan similares son dos rostros y hasta podremos agruparlos por similitud (clustering).

El modelo utiliza como función de pérdida o loss function, "triplet loss". Dicha función minimiza la distancia de los casos positivos y al mismo tiempo maximiza la distancia de los ejemplos negativos, ya que los rostros de una misma identidad aparecerán más cerca que los rostros de distintas identidades.

$$\sum_{i}^{N} \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]_+$$

Triplet loss equation



Triplet loss Learning

[47] https://arxiv.org/pdf/1503.03832.pdf

Desarrollo

La implementación está basada en el paper de FaceNet[48] a partir del código Open Source de David Sandberg[49] hosteado en GitHub. El autor provee 2 modelos pre-entrenados a partir de distintos conjuntos de datos, CASIA-WebFace y VGGFace2.

Dichos modelos podrán ser reutilizados utilizando la técnica de Transfer Learning, reduciendo el tiempo de entrenamiento y el error de generalización de nuestra red. La arquitectura de la CNN utilizada es "Inception Resnet V1" y devolverá como salida un embedding de 128-dimensiones por cada imagen de entrada.

De las distintas estrategias de transfer learning, usaremos la extracción de características (feature extraction). Utilizaremos el modelo pre-entrenado a partir de VGGFace2[50] para generar los embeddings y a continuación entrenaremos un clasificador lineal SVM utilizando dichos embeddings como entrada.

Si bien el código original está escrito en Tensorflow reduciremos la complejidad de la implementación agregando un nuevo layer de abstracción a través de Keras. Para esto nos apoyaremos también en la implementación de Hiroki Taniai[51] que provee el código para traducir un modelo pre-entrenado de Tensorflow a Keras.

Conjunto de datos

El armado y la preparación del conjunto de datos es uno de los pasos más complejos de la implementación de un modelo de Machine Learning. Si bien existen múltiples conjuntos de datos disponibles al público decidimos construir uno propio. La limitante de trabajar con un conjunto de datos de este tipo es que los resultados planteados en este trabajo no podrán ser replicados sin acceso a dicho conjunto y tampoco podrán ser comparados con experiencias previas de otros autores. De todas maneras, todo lo realizado en este trabajo podrá ser replicado utilizando cualquier otro conjunto de datos.

Algunas de las incertidumbres que surgen al plantear el desarrollo de un conjunto de datos son:

- Cuántas imágenes se deben tener en cuenta como mínimo o máximo durante el entrenamiento?
- Cómo evitar caer en overfitting?
- Que tipo de imágenes afectan positivamente o negativamente las predicciones?
- Un mayor número de clases mejorarán la generalización?
- Mejorarán las predicciones si añadimos imágenes de una persona desde recién nacido?

[48] https://arxiv.org/pdf/1503.03832.pdf
[49] https://github.com/davidsandberg/facenet
[50] https://www.robots.ox.ac.uk/~vgg/data/vgg_face2/
[51] https://github.com/nyoki-mtl/keras-facenet

Para conformar el dataset se utilizaron las siguientes fuentes de datos:

- Redes sociales:
    - Facebook
    - Instagram
    - YouTube
- Whatsapp
- Imágenes sacadas con el celular desde el 2013
- Digitalización de fotos impresas (particularmente para aquellos nacidos antes del año 2000)

Debemos tener en cuenta que durante el entrenamiento del modelo utilizaremos un k-fold de 10 por lo que el número mínimo de imágenes por persona debe ser igual a 10. Aunque contaremos con un caso que no cumpla con esta premisa.

Estructura del conjunto

El conjunto seguirá la misma estructura planteada por el dataset LFW (Labeled Faces in the Wild)[52] donde cada persona estará representada por un directorio que contendrá las imágenes con los rostros de dicho individuo. Estos archivos estarán numerados con 4 dígitos comenzando del 0000. La numeración de cada imagen no representará ningún tipo de orden cronológico.

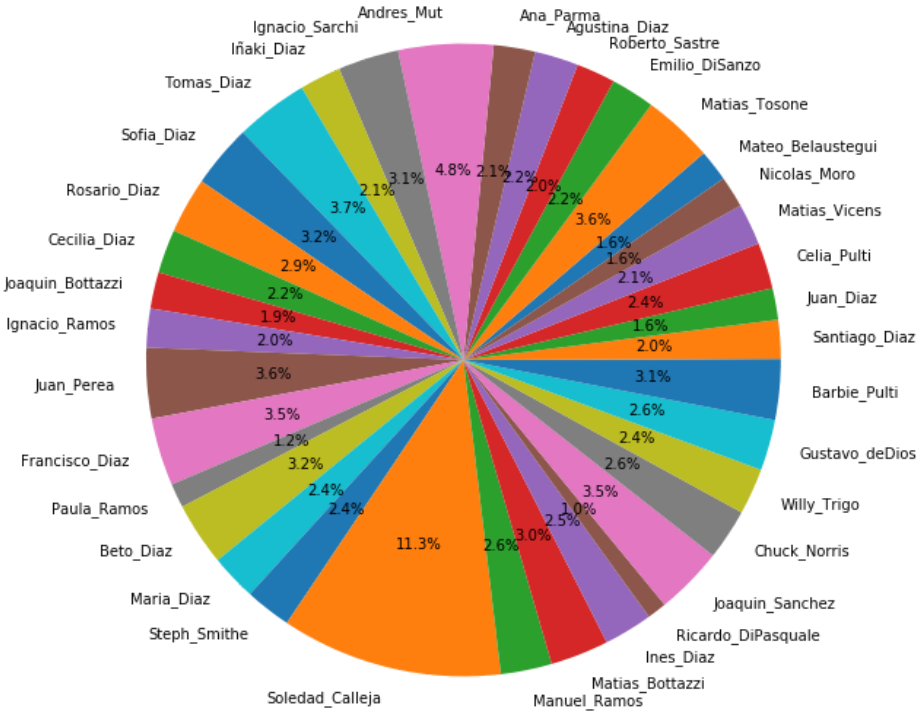
Composición del conjunto

El conjunto está conformado por un total de 805 fotografías de 36 personas donde 12 son mujeres, 24 son hombres y la media de edad son los 35 años. De este conjunto tenemos,

- Media: 22.36 imágenes por clase
- Desvío estándar: 13.03 imágenes por clase
- Mínimo: 8 imágenes en una de las clases
- Máximo: 90 imágenes en una de las clases
- Total: 805 imágenes

Prestando atención a los valores mínimos y máximos y analizando el gráfico de torta representado a continuación, podremos notar que hay 2 clases que contienen menos del 1.5% de los ejemplos y hay una en particular que tiene el 11% de los ejemplos totales. Por lo tanto, podemos definir el conjunto de datos como un conjunto desbalanceado (imbalanced class distribution), ya que el número de elementos que pertenecen a una clase es significativamente menor o mayor al resto de las clases. Durante el desarrollo analizaremos cómo esto afecta o contribuye a la performance del modelo.

[52] http://vis-www.cs.umass.edu/lfw/



Pre-procesamiento del conjunto de datos

Como vimos anteriormente, al describir los distintos desafíos del reconocimiento de imágenes, es necesario realizar un preprocesamiento del conjunto de datos con el que trabajaremos. El procesamiento consistirá en:

- Identificar los rostros en las imágenes
- Alinear y recortar los rostros de la imagen
- Redimensionar los rostros a 160x160 pixels

Los rostros en las imágenes se identificarán utilizando la implementación de MTCNN en Keras por Iván de Paz Centeno[53]. De ahora en más, cuando hagamos referencia al conjunto de datos estaremos hablando del conjunto de imágenes que hayan pasado por el proceso de pre-procesamiento. Dicho proceso agregará un "-N" al nombre de la imagen donde N representa el número de rostro detectado en la imagen, de esta manera si hay alguna imágen con múltiples rostros no estaremos sobreescribiendo la imagen con el último detectado. La alternativa que se suele usar ante múltiples rostros es utilizar el rostro más grande de la imagen, pero en nuestro conjunto original no siempre será la persona que querramos etiquetar.

[53] https://github.com/ipazc/mtcnn

Margenes

El modelo MTCNN devolverá las coordenadas (x, y, w, h) de cada rostro identificado en la imagen. Con estas coordenadas podremos recortar los rotos de las imágenes haciendo un resize a 160x160px que, como se mencionó anteriormente, es el input que espera el modelo pre-entrenado de Facenet. A las coordenadas de cada uno de estos rostros podemos adicionar un margen brindando así más contexto al modelo. Para entender si la definición del margen afecta las predicciones, utilizaremos un pequeño conjunto de datos con 7 personas y entrenaremos el modelo usando 1 sola imagen por identidad. Las predicciones serán validadas con un conjunto de test de 64 imágenes (~9 por persona).

Margen (pixels)	Performance del modelo
0	0.9782608695652174
32	***1***
40	0.9565217391304348
80	0.7608695652173914

En el desarrollo de Facenet, David Sandberg comenta que, durante el proceso de alineación de rostros[54], utiliza un valor de 32 pixels para los márgenes. Menciona también que esto no fué estudiado con suficiente profundidad como para afirmar que es el valor que mejores resultados arrojará. En función de esta recomendación y las pruebas realizadas, definiremos un margen de 32px al recortar cada rostro.

Entrenamiento

Ya procesadas las imágenes podremos entrenar el modelo. Aquí es donde utilizaremos la técnica de transfer learning para calcular los embeddings de cada uno de los rostros. Los embeddings estarán calculados con el modelo pre-entrenado de FaceNet. Cada embedding representarán un vector de 128-dimensiones que podremos ubicar en un espacio vectorial con el resto de los embeddings calculados. Esto nos permitirá computar la distancia euclidiana entre vectores, ya que rostros similares tenderán a estar más cerca que rostros de distintas personas. Podremos también utilizar la distancia para aplicar técnicas de clustering, pero esto quedará fuera del alcance de este trabajo.

Para el tamaño de nuestro conjunto de datos utilizaremos un algoritmo de aprendizaje supervisado (supervised learning) que nos permita clasificar múltiples ejemplos en N dimensiones. El algoritmo que soporta estas características es SVM (Support Vector Machine) y podremos utilizar la implementación de Sklearn, particularmente C-Support Vector Classification (SVC)[55].

[54] https://github.com/davidsandberg/facenet/wiki/Triplet-loss-training#face-alignment
[55] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Evaluaremos el score del entrenamiento utilizando la técnica de cross-validation lo que nos permitirá identificar si durante este proceso introducimos overfitting. Agregar este tipo de validación demandará mayor tiempo de entrenamiento y uso de recursos de hardware pero nos dará métricas más concretas de la performance del modelo.

Parámetros de configuración para el entrenamiento:

- SVC hiper-parámetros:
    - Kernel: lineal.
    - Epochs: el número de iteraciones estará controlada por la propia implementación y finalizará cuando converja.
    - C (penalización): 10.
- Cross validation: k-fold en 10.
- Dataset:
    - 75% de las imágenes para el conjunto de entrenamiento.
    - 25% de las imágenes para el conjunto de pruebas.

El valor de C definirá el margen que dejaremos entre los conjuntos en el hiperplano. A valores más chicos de C el margen será mayor y a valores más grandes será menor. Para la definición del valor de C (por default es 1), se utilizó cross-validation con distintos valores para los cuales se obtuvo:

C	Performance del modelo
0.01	0.11 (+/- 0.01)
0.25	0.88 (+/- 0.08)
0.5	0.92 (+/- 0.06)
0.75	0.93 (+/- 0.04)
1	0.93 (+/- 0.04)
1.25	0.93 (+/- 0.04)
1.5	0.93 (+/- 0.04)
1.75	0.93 (+/- 0.04)
2	0.94 (+/- 0.04)
10	**0.95 (+/- 0.03)**
100	0.95 (+/- 0.03)

A partir de un *C: 10* nuestro modelo se hará *0.95 (+/- 0.03)* más preciso, por lo que este será el valor que utilizaremos como parámetro de entrenamiento. Si bien la ganancia es mínima, el esfuerzo estuvo solo en modificar un parámetro de configuración.

La distribución del conjunto datos se realizó con la función *train_test_split*[56] de Scikit-learn lo que garantiza que los subconjuntos generados sean conformados de manera aleatoria.

[56] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Predicciones

Una vez que contamos con el modelo entrenado podremos comenzar a realizar predicciones. Para esto, utilizaremos el conjunto de pruebas, que cuenta con datos nunca antes vistos por el modelo. A partir de los resultados arrojados, analizaremos si estamos en situación de underfitting, overfitting o nuestro modelo responde correctamente.

Como mencionamos durante el entrenamiento, la performance (accuracy) del modelo entrenado y validado contra el conjunto de entrenamiento es de 0.95 por lo que al tener un bajo error de entrenamiento (0.5) podremos descartar la situación de underfitting.

Si bien utilizamos la técnica de cross-validation para detectar a tiempo tanto underfitting como overfitting, es importante que sigamos monitoreando la performance. Las predicciones sobre el conjunto de pruebas arrojan una performance de 0.931, por lo que al ser muy baja la diferencia entre el valor de entrenamiento y el de pruebas podremos descartar también la situación de overfitting.

De 202 imágenes analizadas, 188 fueron correctamente identificadas por el modelo quedando solo 14 imágenes en el conjunto de error.

Análisis de resultados

Por cada predicción, el modelo devolverá un valor de probabilidad entre 0 a 1 por cada una de las clases en las que haya sido entrenado. Es decir, para nuestro conjunto de datos devolverá la probabilidad de cada elemento sobre los 36 miembros del conjunto.

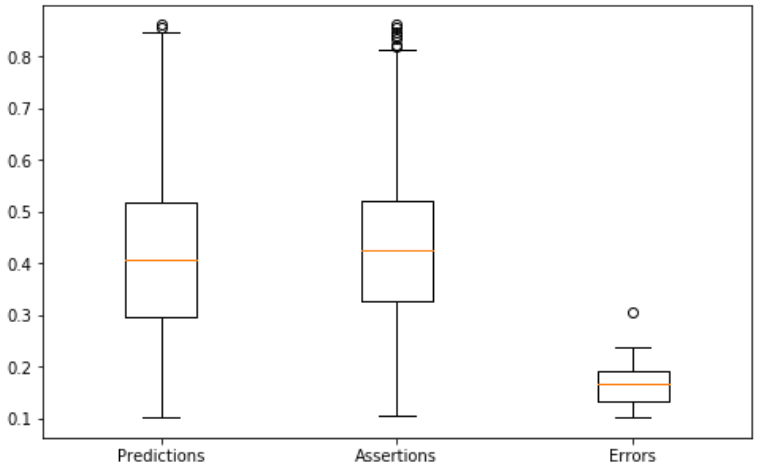
De todas las predicciones tenemos que,

- Media: 0.43
- Desvío estándar: 0.18
- Mínimo: 0.10
- Máximo: 0.86
- Total: 188 ejemplos

Si bien el modelo nunca llegó a realizar una estimación por encima del 0.86, los resultados fueron en el 93.1% de casos, correctos.

Para las predicciones incorrectas tenemos que,

- Media: 0.17
- Desvío estándar: 0.05
- Mínimo: 0.10
- Máximo: 0.31
- Total: 14 ejemplos



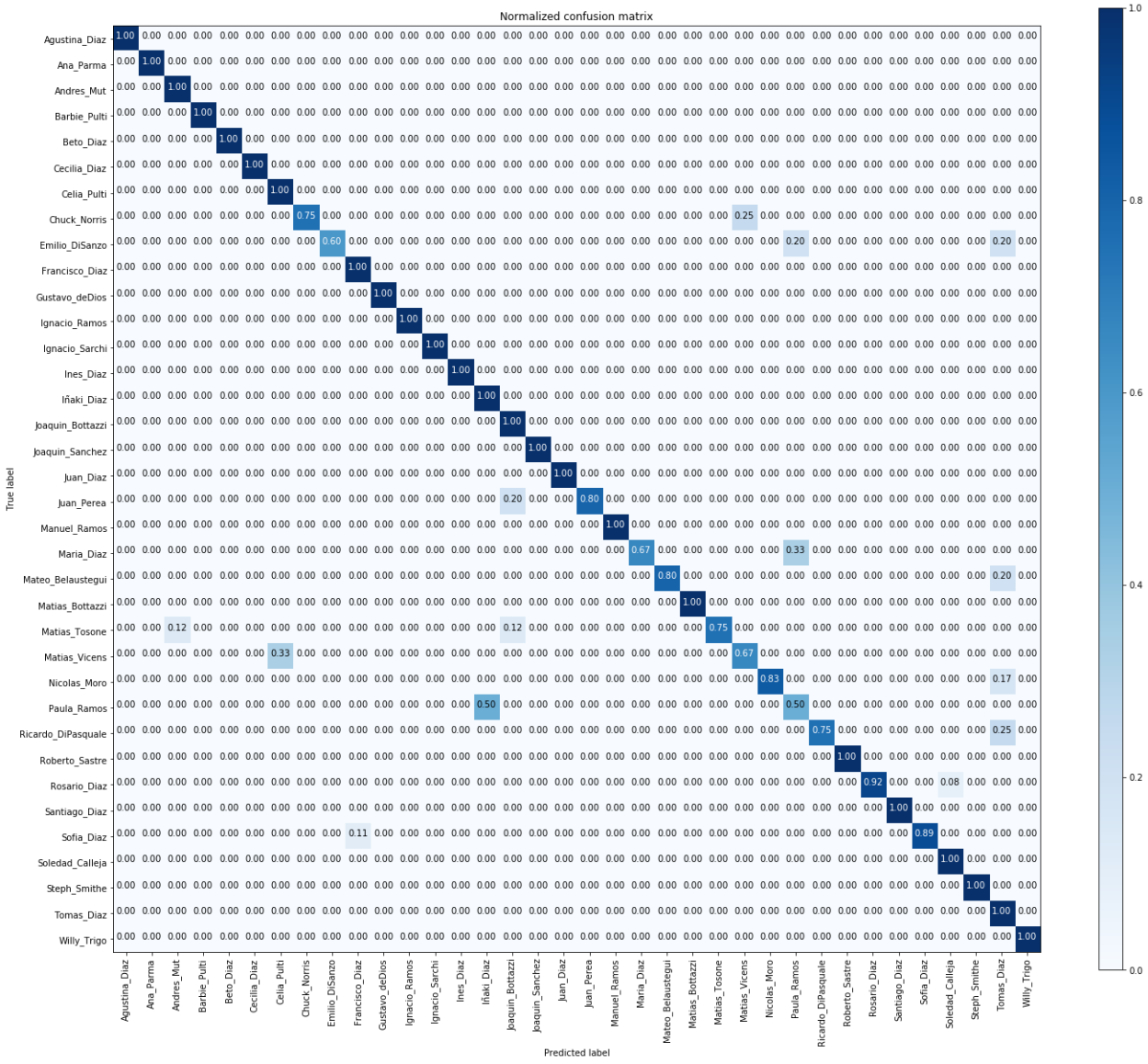
Distribución de las predicciones totales, los aciertos y los errores.

Del conjunto de errores podemos considerar sólo uno como anomalía ya que está fuera de los límites de la distribución con un valor de 0.30688606. Podemos ver también que hay casos correctos cuya distribución se solapa con la distribución de los errores. Esto último será un problema si queremos definir un umbral mínimo a partir del cual consideramos que nuestras predicciones son correctos ya que estaremos descartando estos casos por tener baja probabilidad.

Como los errores son pocos, podemos analizarlos e intentar entender bajo qué situaciones pueden darse. Para comenzar con este análisis necesitaremos el soporte de la Matriz de Confusión y las métricas de Precision y Recall.

Matriz de confusión

Comenzaremos este análisis a partir de la matriz de confusión donde compararemos los valores predichos con los reales. La matriz nos ayudará a ver de manera rápida y gráfica cuáles son las clases con problemas. Como soporte de la matriz, usaremos un mapa de calor (heat map) que nos permitirá utilizar colores para ayudar la visualización. Los valores de la matriz están normalizados de 0 a 1 ya que las clases no cuentan con la misma cantidad de ejemplos.



De las 36 clases con la que cuenta nuestro conjunto de datos tenemos 12 que fueron predichas de manera incorrecta y en la matriz podremos identificarlas como las que no tienen un valor igual a 1 en la diagonal (azul oscuro). En el eje y encontramos los valores de verdad y en el eje x tenemos las predicciones para cada una de las clases.
Leyendo el gráfico desde los valores de verdad (eje y) si la diagonal no es igual a 1 veremos los casos en los que dicha clase se predijo de manera incorrecta y bajando al eje x encontramos la predicción que se realizó. Pero si leemos a partir del eje de las predicciones, en los casos que la diagonal no es igual a 1, veremos la predicción que se realizó y cuál era en realidad su valor de verdad.
Por ejemplo, leyendo desde los valores de verdad, Matías Vicens se predijo un 33% de las veces como Celia Pulti. Si bien 33% es un valor grande hay que tener en cuenta que esta clase contaba con 3 ejemplos en el conjunto de pruebas. A su vez, si leemos la misma clase desde los valores predichos, para Matías Vicens tenemos que Chuck Norris fué predicho como Matías

el 25% de las veces. Para Chuck Norris teníamos 4 ejemplos en el conjunto de datos de pruebas, de manera que en ambos casos una sola imagen fué predicha de manera errónea.

Matias Vicens

Chuck Norris

Precision y Recall

Dejaremos en la tabla todos aquellos casos en los que, tanto el Precision como el Recall son distintos de 1. Si comparamos la tabla con la matriz de confusión, tendremos los 12 ejemplos que fueron predichos con error además de todas aquellas clases involucradas en dichos errores.

Class	Precision	Recall	Support
Andres_Mut	0.857143	1.000000	6
Celia_Pulti	0.800000	1.000000	4
Chuck_Norris	1.000000	0.750000	4
Emilio_DiSanzo	1.000000	0.600000	5
Francisco_Diaz	0.833333	1.000000	5
Iñaki_Diaz	0.666667	1.000000	2
Joaquin_Bottazzi	0.600000	1.000000	3
Juan_Perea	1.000000	0.800000	5
Maria_Diaz	1.000000	0.666667	3
Mateo_Belaustegui	1.000000	0.800000	5
Matias_Tosone	1.000000	0.750000	8
Matias_Vicens	0.666667	0.666667	3
Nicolas_Moro	1.000000	0.833333	6
Paula_Ramos	0.333333	0.500000	2
Ricardo_DiPasquale	1.000000	0.750000	4
Rosario_Diaz	1.000000	0.916667	12
Sofia_Diaz	1.000000	0.888889	9
Soledad_Calleja	0.962963	1.000000	26
Tomas_Diaz	0.666667	1.000000	8
Accuracy			0.930693
Macro average	0.927410	0.914506	202
Weighted average	0.948910	0.930693	202

Para ponerle una cara a cada una de nuestras predicciones erróneas,

Esto requerirá un análisis más profundo, pero podríamos afirmar que el algoritmo falla generalmente en imágenes de perfil, con el rostro parcialmente cubierto y en perfiles que hayan tenido ejemplos de pequeños.

Límites de la solución

Los límites de la implementación dependerán mayormente de los casos de uso que se busquen resolver y deberíamos realizar pruebas con miles de imágenes, posiblemente utilizando el conjunto de datos de CelebFaces Attributes (CelebA) Dataset[57]. Para el conjunto de datos presentado en este trabajo tenemos una performance por debajo de valores state-of-the-art (99%) pero aún así podemos considerar que una performance del 95% para el conjunto de entrenamiento y 93% para el conjunto de pruebas son muy buenos resultados. También hay que tener en cuenta que la necesidad de recursos de procesamiento para entrenar el modelo SVM es relativamente baja.
Desde el punto de vista de escalabilidad la solución no escala ya que el procesamiento y entrenamiento es lineal. Tanto el procesamiento, el entrenamiento y las inferencias realizadas en este trabajo se hicieron en una computadora personal. Es posible extender la implementación para que la misma escale utilizando servicios como Amazon SageMaker[58].

Recursos utilizados

Como referencia conceptual y para tener un entendimiento de tiempos y recursos se describirán las características del hardware utilizado.

Se utilizó una computadora personal con las siguientes características:

- MacBook Pro (Retina, 13-inch, Mid 2014)
- Procesador: 3 GHz Intel Core i7
- Memoria: 16 GB 1600 MHz DDR3
- Disco: SSD

[57] http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
[58] https://aws.amazon.com/es/sagemaker/

Resultados

Si bien la performance del modelo entrenado a partir de nuestro conjunto de datos es buena (0.93) todavía hay posibilidad de mejorarla. La implementación se puede extender para incluir:

- Alineación de rostros utilizando "Deep Funneling"
- Augmentation

Alineación de rostros: Luego de analizar la implementación en Keras de MTCNN, utilizada para el reconocimiento de rostros en las imágenes, encontramos que no está realizando de manera automática la alineación de los rostros. A su vez, dicho código tiene un bug que deja de reconocer rostros que se encuentran a más de 45 grados. Si bien estos puntos son mejorables, la recomendación para mejorar la performance del modelo es a través de implementación de la alineación de las imágenes utilizando "Deep funneling" esta técnica está desarrollada en el paper "Learning to Align from Scratch"[59] de Gary B. Huang, Marwan Mattar, Honglak Lee, y Erik Learned-Miller.

Augmentation: Para aquellos casos en los que no contamos con suficientes ejemplos para entrenar el modelo podremos utilizar técnicas de augmentation como la modificación de colores, rotaciones, distorsiones, etc. En el paper "The Effectiveness of Data Augmentation in Image Classification using Deep"[60] de Jason Wang y Luis Perez, se extiende el concepto a través del uso de GANs (Generative Adversarial Network). Si bien la introducción de GANs es interesante, se puede comenzar aplicando técnicas tradicionales con Keras, que nos provee una API de ImageDataGenerator[61].

AWS Rekognition

Introducción

AWS Rekognition es un servicio de reconocimiento de imágenes que utiliza Deep Learning para detectar objetos, escenas y rostros; extrae texto, reconoce a personas famosas e identifica contenido inapropiado en imágenes. También permite realizar búsquedas y comparar rostros. Rekognition Image se basa en la misma tecnología de Deep Learning desarrollada por los científicos de visión informática de Amazon para analizar miles de millones de imágenes al día para Prime Photos. El servicio devuelve una puntuación de fiabilidad de todos los elementos que identifica para poder tomar decisiones acerca de cómo utilizar los resultados. Además, todos los rostros detectados se devuelven con coordenadas (x, y, w, h) que abarca todo el rostro y se puede utilizar para localizar la posición del rostro en la imagen. El servicio nos provee una API con la cual podemos interactuar sin tener ningún conocimiento de Deep

[59] http://vis-www.cs.umass.edu/papers/nips2012_deep_congealing.pdf
[60] http://cs231n.stanford.edu/reports/2017/pdfs/300.pdf
[61] https://keras.io/preprocessing/image/

Learning. En otras palabras, solo debemos interactuar con una API y procesar los resultados para tomar una decisión.

Desarrollo

Al ser Rekognition un servicio público, pago y utilizado por miles de usuarios a nivel mundial, realizaremos el análisis del servicio con la premisa de que todos los ejemplos que hayan sido bien identificados por nuestra implementación de Facenet lo serán también por este servicio. Por lo tanto, nos enfocaremos en todos aquellos casos que no hayan tenido buenas probabilidades en su predicción. El servicio ofrece más funcionalidades de las que evaluaremos en este trabajo ya que nos enfocaremos solo en la comparación de rostros contra una lista de rostros conocidos por el sistema.

Conjunto de datos

Tomaremos como conjunto de datos el mismo que utilizamos para entrenar el modelo de Facenet, usando los mismos elementos para el conjunto de entrenamiento y de pruebas. Por este motivo la identificación de rostros ya estará resuelta y no necesitaremos de Rekognition para hacerlo pese a que el servicio nos provee esta funcionalidad. La API para identificación de rostros nos devolverá también información cómo la edad, si la persona tiene lentes, barba, bigote, los ojos abiertos o cerrados y hasta un análisis de sentimiento. Todas las imágenes serán almacenadas en un bucket de S3 y el servicio recibirá el path para acceder a cada una de ellas.

Pre-procesamiento del conjunto de datos

El servicio no requiere de ningún tipo de preprocesamiento ya que aceptará imágenes tanto PNG como JPEG con un tamaño máximo de 15 MB. El tamaño mínimo de la imagen deberá ser de 80x80 píxeles donde el rosto tendrá que ocupar por lo menos 40x40 píxeles. En el caso de tener una imágen con múltiples rostros, se utilizará el rostro más grande y se lo comparará con el resto de los rostros almacenados para determinar su identidad.

Las imágenes de nuestro conjunto de datos ya se encuentran recortadas con una resolución de 160x160 píxeles y utilizaremos las mismas para no brindar más información que la necesaria donde se pueda aprovechar el contexto para fortalecer las predicciones.

Entrenamiento

Si bien en Rekognition no es necesario entrenar un modelo, debemos definir lo que se conoce como "Collection", y es donde se almacenará la información de cada uno de los rotos detectados. La documentación del servicio[62] menciona que las imágenes de los rostros no son almacenadas en las distintas colecciones sino que el algoritmo de detección de rostros extrae los atributos de un rostro en un vector (feature vector) y esto es lo que se termina almacenando en la colección (conceptualmente funciona exactamente igual que nuestro algoritmo de

[62] https://docs.aws.amazon.com/rekognition/latest/dg/collections.html

Facenet). Cada colección podrá almacenar hasta 20 millones de rostros. A su vez, las colecciones se asocian a una versión[63] específica del modelo que se utiliza para realizar el reconocimiento. Por defecto al crear una nueva colección se asocia a la última versión del modelo disponible y no es posible migrar una colección de una versión de un modelo a otra ya que distintas versiones de modelos no son compatibles entre ellas. La única manera de migrar de una versión del modelo a una más nueva es reindexando todas las imágenes nuevamente, por lo que es importante mantener las imágenes almacenadas.

Crearemos una colección utilizando la API CreateCollection[64] y luego indexaremos los rostros de nuestro conjunto de entrenamiento utilizando la API IndexFaces[65]. Es necesario que las imágenes que utilicemos con la API de IndexFaces contengan solo 1 rostro. Esta API nos devolverá un ID único por rostro y será nuestra responsabilidad asociar dicho ID al nombre de la persona. Por lo tanto tendremos tantos IDs por persona como número de imágenes indexadas para esa persona. En la siguiente tabla vemos que las distintas imágenes para una misma persona tienen distintos IDs.

Rekognition Face ID	Etiqueta	Imagen
3879e682-99e5-4ad0-aada-e1c801f593b4	Joaquin_Sanchez	Joaquin_Sanchez_0003-0.jpg
e630fa1a-2ae5-45b6-b36a-c395bc1b0421	Soledad_Calleja	Soledad_Calleja_0077-0.jpg
7fe5c245-dbac-4831-b4fc-58b301de0ef2	Soledad_Calleja	Soledad_Calleja_0007-0.jpg
d75a0042-0f88-449b-9458-e243dcbc325b	Maria_Diaz	Maria_Diaz_0000-0.jpg
8753c551-95b7-47de-b88f-f71527827808	Matias_Tosone	Matias_Tosone_0024-0.jpg
44217609-6a6f-4fe0-b8d0-3fdfae310ba9	Soledad_Calleja	Soledad_Calleja_0097-0.jpg

Durante la indexación de las imágenes del conjunto de entrenamiento Rekognition descartó la siguiente imagen por no tener ningún rostro. Más allá de ser una imagen compleja por tener parte del rostro cubierto, es un falso negativo y de alguna manera deberíamos reflejar esto como un error ya que Facenet pudo reconocerla, aunque no sabemos si esta imagen afecta positiva o negativamente algún otra predicción.

Predicciones

Las predicciones realizadas fueron sorprendentes, arrojando una performance de 0.995 con 1 solo caso erróneo. Este error también se encontró dentro del conjunto de error para las predicciones realizadas por Facenet.

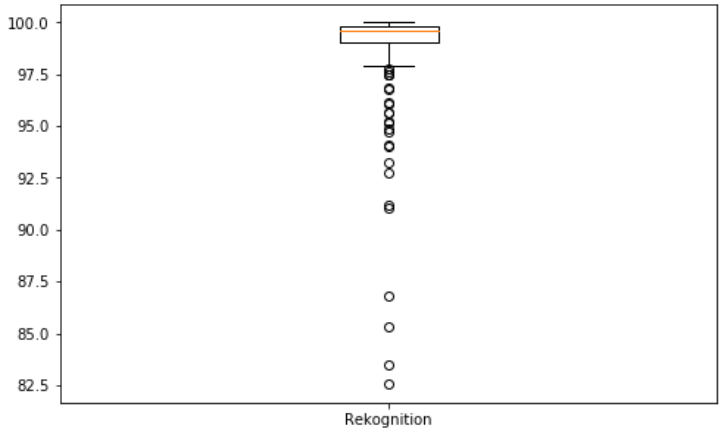
[63] https://docs.aws.amazon.com/rekognition/latest/dg/face-detection-model.html
[64] https://docs.aws.amazon.com/rekognition/latest/dg/API_CreateCollection.html
[65] https://docs.aws.amazon.com/rekognition/latest/dg/API_IndexFaces.html

Facenet	AWS Rekognition
	

Claramente es un caso complejo ya que más del 50% del rostro está cubierto y no es posible determinar con claridad la posición de los ojos que generalmente se utiliza durante el proceso de alineación de la imagen.

Rekognition devuelve las probabilidades en porcentajes de 0 a 100. De todas las predicciones realizadas tenemos que,

- Media: 98.75
- Desvío estándar: 2.52
- Mínimo: 82.54
- Máximo: 100.00

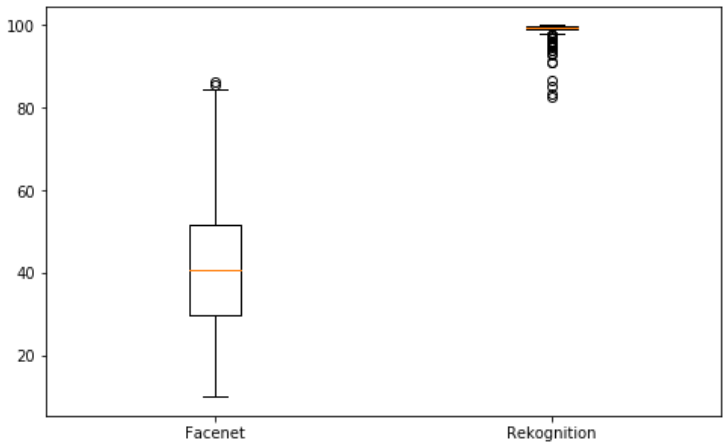


Distribución de las predicciones para Rekognition

Al tener 1 solo caso de error no tiene mucho sentido analizar la matriz de confusión, ni métricas como Precision o Recall. En consecuencia, compararemos las mejores predicciones y los casos de error de Facenet con los resultados de Rekognition.

Distribución y definición de umbral

Para entender mejor cómo están distribuidas las predicciones realizaremos un box-plot con las distintas predicciones tanto para Facenet como Rekognition, de manera que podamos comparar ambas distribuciones en un mismo gráfico.



Distribución de las predicciones Facenet vs Rekognition

La distribución de predicciones de Facenet es más pareja que la de Rekognition en cambio para Rekognition la dispersión es muy baja y los valores se encuentran mayormente entre 100 y 97.5. También podemos ver que las predicciones de Rekognition tienen mayor cantidad de outliers (o extremos), unos 22 casos menores a 97.5.

Es importante entender la dispersión de nuestras predicciones ya que para que los modelos puedan responder a casos de uso en la vida real, debemos definir el umbral que deberán superar nuestras predicciones para ser consideradas certeras.

Definir un umbral para Rekognition es más simple que para Facenet ya que tenemos solo 1 caso de error y es el extremo inferior de las predicciones (82.5). Los ejemplos descartados representarán aquellos correctamente clasificados que no estaremos aceptando como correctos, es decir, en cuanto estaríamos incrementando el conjunto de error.

Umbral	Ejemplos descartados	
85	1	0.45%
90	3	1.36%
95	11	5.00%
98.75 (media)	41	18.18%
99	49	22.27%

Para Facenet es algo más complejo ya que el conjunto de ejemplos correctamente clasificados y el de error se superponen. Recordemos también que el conjunto de error de Facenet contiene un outlier en 0.306 por lo que estaremos obligados a descartar todos los ejemplos a partir de este valor.

Umbral	Ejemplos descartados	
0.31	42	19.09%
0.43 (media general)	96	43.63%
0.45 (media correctos)	106	48.18%

La definición del umbral nos dará la tasa de error de cada solución. Para Facenet, si definimos un umbral de 0.31 tendremos un 20% de predicciones descartadas por error. Para Rekognition si tomamos un umbral de 90 el 1.5% de las predicciones serán descartadas.

Límites de la solución

Resolución de imagen

Amazon Rekognition admite una gran variedad de resoluciones de imagen. Para obtener los mejores resultados, recomendamos el uso de una resolución VGA (640x480) o superior. Si la resolución es inferior a QVGA (320x240), hay bastantes probabilidades de que no puedan identificarse rostros, objetos o contenido inapropiado, aunque Amazon Rekognition acepta cualquier imagen que tenga al menos 80 píxeles en ambas dimensiones.

Rostros por imagen

Con Amazon Rekognition, se pueden detectar hasta 100 rostros en una imagen.

Cantidad de rostros por Collection

Cada colección podrá almacenar hasta 20 millones de rostros

Costos

Amazon Rekognition Image, tiene dos tipos de costos:

1. **Análisis de imagen**: Rekognition cobra cada vez una imagen es analizada a través de llamadas a la API. Ejecutar múltiples llamados a la API con una única imagen cuenta como procesamiento múltiple de imágenes.

Tipo de costo	Precios	Precio por 1000 imágenes
Primer millón de imágenes procesadas al mes	0,001 USD por imagen	1,00 USD
Siguientes 9 millones de imágenes procesadas al mes	0,0008 USD por imagen	0,80 USD
Siguientes 90 millones de imágenes procesadas al mes	0,0006 USD por imagen	0,60 USD

Más de 100 millones de imágenes procesadas al mes	0,0004 USD por imagen	0,40 USD

2. **Almacenamiento de metadatos de rostros**: Para habilitar la búsqueda de rostros, se deberán almacenar metadatos de rostros en una Collection para poder buscar coincidencias. Los cargos de almacenamiento se aplican mensualmente y se prorratean en el caso de meses parciales. El costo de almacenamiento es de 0,00001 USD/metadatos de rostros al mes.

Los precios variarán según la región que se utilice, todas las pruebas realizadas en este trabajo se hicieron en el Norte de Virginia (us-east-1).

Resultados

Si bien la solución escala prácticamente sin ningún tipo de esfuerzo, si nuestro conjunto de datos crece mucho en el tiempo no podremos beneficiarnos de volver a entrenar nuestro propio modelo haciéndolo aún más certero a nuestros casos de uso, sino que dependeremos de que AWS actualice su modelo. A su vez, AWS tenderá a mejorar su modelo con miles de usuarios más que aportarán imágenes beneficiandonos nosotros también.
Los resultados arrojados por la solución son verdaderamente impresionantes y el esfuerzo de la implementación es mínimo. Vale aclarar también, que como ya mencionamos, no se requiere ningún conocimiento de Machine Learning lo que simplifica las cosas aún más. De todas maneras la experiencia previa ganada con la implementación de Facenet agilizó aún más el uso de la solución, sobre todo desde el lado analítico.

Conclusión

El trabajo comienza con el desarrollo de la teoría necesaria para poder comprender cómo funciona un modelo de Deep Learning orientado al reconocimiento de imágenes. Estos temas nos ayudaron a comprender la arquitectura de Resnet usada por Facenet para calcular los embeddings de cada rostro y así sortear los problemas que se presentaron durante la implementación del modelo. Realizar una comparación directa entre nuestra implementación de Facenet y AWS Rekognition es algo injusta, no solo porque Rekognition es un servicio probado por miles de usuarios sino también porque nuestra implementación no es lo suficientemente robusta para alcanzar semejante performance. Más allá de este punto, las predicciones de Facenet fueron muy buenas logrando un 93% de aciertos y estamos muy conformes con los resultados obtenidos. Vale la pena aclarar también que aún hay espacio a introducir mejoras que ayuden a incrementar los aciertos de nuestro modelo como técnicas de alineación de rostros y augmentation. Como futura implementación es interesante pensar en complementar ambas soluciones pudiendo realizar inferencias locales con Facenet y utilizando el servicio de Rekognition cuando la probabilidad de acierto es baja.

Traducciones a múltiples idiomas en tiempo real, autos que conducen solos (Tesla[66] Autopilot y Weymo[67]), asistentes personales que establecen conversaciones (Alexa[68] y Ok Google) y algoritmos que ganan competencias de juegos (Dota 2[69] y AlphaStar[70]). Todas estas tecnologías están impulsando cambios muy fuertes en cómo interactuamos con nuestro entorno y comprenderlas nos ayudará, no solo a promover el cambio, sino también a poder adelantarnos a las oportunidades que las mismas nos presenten.

[66] https://www.tesla.com/autopilotAI
[67] https://waymo.com/tech/
[68] https://www.amazon.jobs/en/teams/alexa-ai
[69] https://openai.com/projects/five/
[70] https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

Bibliografía

Salvo en aquellos casos en el que concepto es referenciado, la teoría de este trabajo está basada en las clases abiertas "CS231n: Convolutional Neural Networks for Visual Recognition" de la Universidad de Stanford[71]. La implementación práctica está inspirada principalmente en el desarrollo de Facenet por David Sandberg[72]. El código utilizado en el desarrollo de este trabajo se encuentra en un repositorio en Github[73].

[71] http://cs231n.stanford.edu
[72] https://github.com/davidsandberg/facenet
[73] https://github.com/franciscodiazydiaz/keras-facenet