Esta obra está bajo una
Licencia Creative Commons Atribución-NoComercial-SinDerivar 4.0 Internacional

Módulo 8. Minería de Datos II

Curso Modular Big Data y Data Science Aplicados a la Economía y a la Administración y Dirección de Empresas

Pablo Sánchez Cabrera Ángel Rodríguez Chicote

Alfonso Carabantes Álamo

2024-07-01

Índice

1 Introducción **4**

2 Deep Learning **6**
2.1 Introducción . . . 6
2.2 Revisión de las Redes Neuronales . . . 7
2.2.1 Principales arquitecturas de Deep Learning . . . 12
2.3 Redes Neuronales Convolucionales . . . 15
2.3.1 Introducción . . . 15
2.3.2 Pooling . . . 19
2.3.3 Padding . . . 19
2.3.4 Stride . . . 20
2.3.5 Redes convolucionales con nombre propio . . . 21
2.4 Redes Nueronales Recurrentes . . . 24
2.5 Autoencoders . . . 24
2.6 Arquiecturas Preentranadas . . . 30
2.6.1 Detección de objetos . . . 30
2.6.2 Tratamiento de audios . . . 30
2.6.3 Tareas sobre textos . . . 30
2.7 Aprendizaje por Refuerzo . . . 30
2.7.1 Introducción . . . 30
2.7.2 Historia del Aprendizaje por Refuerzo . . . 30
2.7.3 Formalismo Matemático . . . 30
2.7.4 Taxonomía de Algoritmos . . . 35
2.7.5 Q-Learning (value) . . . 37
2.7.6 DQN (Deep Q-Learning) . . . 39
2.7.7 Listado Algoritmos . . . 41
2.8 Redes Generativas Adversarias . . . 45
2.9 Actualidad y algunos conceptos relacionados con el Deep Learning . . . 45
2.10 Software para aplicar Deep Learning . . . 45

3 Análisis Causal y Modelos Gráficos Probabilísticos **46**
3.1 Relaciones y Modelos Causales . . . 46
3.2 Redes bayesianas: aprendizaje y clasificadores . . . 46
3.2.1 Hipótesis MAP y Naive-Bayes . . . 46
3.2.2 Modelos bayesianos . . . 46
3.3 Modelos Ocultos de Markov . . . 46
3.3.1 Tipos de cadenas de Markov . . . 46
3.3.2 Aplicaciones de MOMs . . . 46
3.4 Causal Machine Learning . . . 46
3.4.1 Efectos causales. ATE y CATE . . . 46
3.4.2 Uplift . . . 46

4 Algoritmos Genéticos **47**
4.1 Introducción . . . 47
4.2 Fundamentos teóricos (conceptos) . . . 49
4.2.1 Codificación de los datos . . . 49
4.2.2 Algoritmo . . . 50
4.2.3 Otros Operadores . . . 53
4.2.4 Parámetros necesarios al aplicar Algoritmos Genéticos . . . 54
4.3 Conclusiones . . . 54
4.4 Ejemplos prácticos . . . 55
4.4.1 Algoritmos genéticos con R . . . 55
4.4.2 Algoritmos genéticos en Python . . . 59

5 Lógica Difusa **61**
5.1 Conceptos clave. . . . 61
5.1.1 Conjuntos difusos y funciones de membresía . . . 61
5.1.2 Inferencia y Modelamiento difuso . . . 61
5.2 Ejemplos prácticos . . . 61
5.2.1 Clustering Difuso . . . 61
5.2.2 Herramientas de Diagnóstico . . . 61

Bibliografia **62**

Appendices **63**

A Anexo 1 **63**

1 Introducción

Anotación[1] Diagrama Conceptual Módulo 8: Minería de Datos II

Referecnia bibliografica (Knuth 1984)

Referencia gráfico Figure 1.1 Otra referencia a gráfico 1.1

Note

Note that there are five types of callouts, including: `note`, `warning`, `important`, `tip`, and `caution`.

Warning

Atención!!!

Important

Importante

Tip with Title

This is an example of a callout with a title.

Expand To Learn About Collapse

This is an example of a 'folded' caution callout that can be expanded by the user. You can use `collapse="true"` to collapse it by default or `collapse="false"` to make a collapsible callout that is expanded by default.

[1] Anotación de prueba

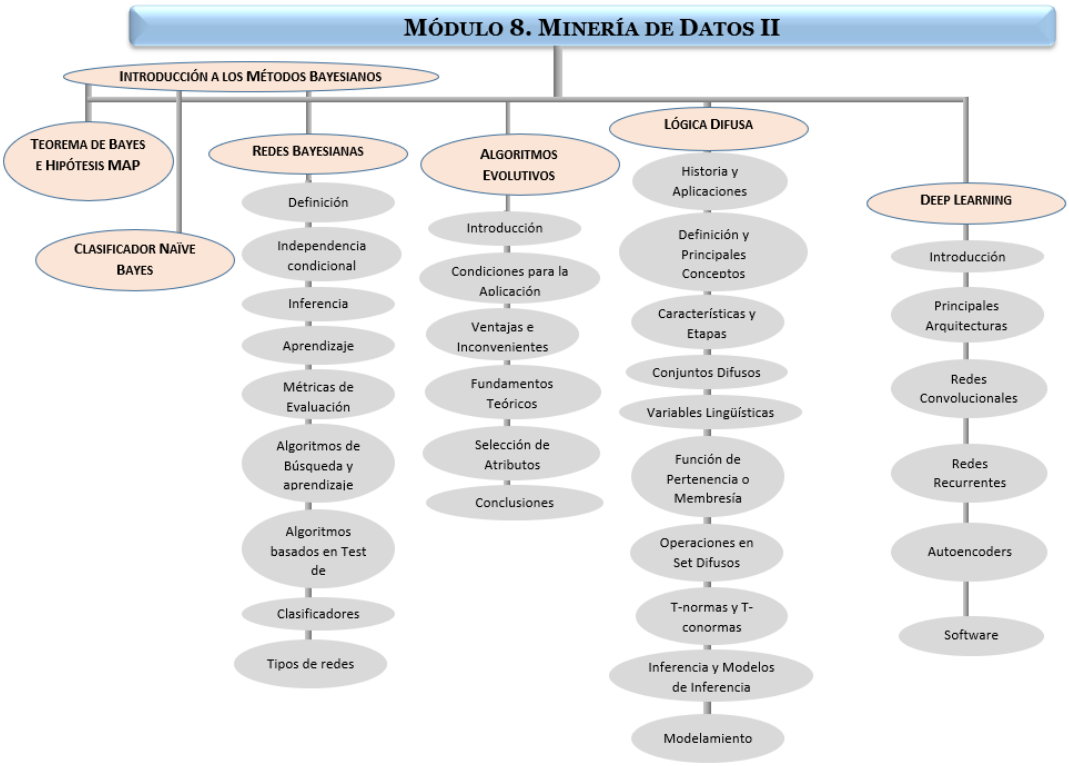


Imagen 1.1: Diagrama Conceptual

2 Deep Learning

2.1 Introducción

El **deep learning o aprendizaje profundo** es un concepto amplio, un término de moda que atiende a una realidad que cada día está más presente entre nosotros, y que no tiene una única definición que se pueda considerar veraz, aunque podemos generalizar la definición afirmando que surge de la idea de imitar el cerebro a partir de la utilización de hardware y software, para crear inteligencia artificial. Este intento de imitar al cerebro se ha plasmado en lo que se conoce como redes neuronales artificiales (RNA) que están dotadas de una capacidad de abstracción jerárquica: representan los datos de entrada en varios "niveles", a través de arquitecturas en varias capas, cada capa aprende patrones más complejos según la profundidad de ésta, para conseguir información útil para el aprendizaje.

Podemos afirmar que el principio general del funcionamiento de una arquitectura profunda es guiar el entrenamiento de las capas intermedias utilizando aprendizaje no supervisado. Algunos de los prototipos de arquitecturas y/o algoritmos desarrollados se basan en otras arquitecturas "más simples", modificándolas y obteniendo una arquitectura "más profunda", lo que quiere decir que en sus capas y neuronas se separan las características de un conjunto de datos de entrada de una forma jerarquizada.

Otra definición dada del aprendizaje profundo es que es un subconjunto de los algoritmos usados en machine learning, y se caracterizan por tener una **arquitectura en capas** donde cada capa aprende patrones más complejos según la profundidad de ésta. En este sentido, existen ya muchos prototipos de redes neuronales que tienen esta característica, por ejemplo, las ya muy conocidas redes convolucionales o las redes recurrentes. El Perceptron Multicapa de una sola capa explicado en el módulo 5 no es considerado un método de aprendizaje profundo. El Deep Learning es una técnica reciente donde aún se sigue investigando y desarrollándose y que surgió a partir del año 2006, año en el que se considera fue retomada su investigación. El principal problema por el que se había ralentizado su progreso es que el entrenamiento basado en el método del gradiente para redes neuronales profundas supervisadas, según muchos investigadores, se estancaba en lo que se llama mínimo local aparente, lo que significa que una red con más capas ocultas, es decir una red neuronal más profunda, obtiene peores resultados que una red con sólo una capa oculta. Los investigadores abordaron este tema planteado a través de diferentes ópticas y se descubre que se obtienen resultados más precisos si cada capa de la red es entrenada a través de un algoritmo de preentrenamiento no supervisado (el método del gradiente requiere que sea supervisado) Para el entrenamiento de las capas, diversos autores sugieren utilizar autoencoders o máquinas de Boltzmann y, una vez se hayan preentrenado las diferentes capas, se puede aplicar un criterio supervisado. En estos últimos años se ha producido un rápido crecimiento en la cantidad de arquitecturas y/o algoritmos de entrenamiento en una RNA, incluso, variantes de ellos, las cuales siguen el concepto planteado anteriormente.

2.2 Revisión de las Redes Neuronales

Vamos a hacer una revisión de las redes neuronales para posteriormente poder abordar los diferentes tipos de redes neuronales que se utilizan en Deep Learning. Algunos de los avances más recientes en varios de los diferentes componentes que forman parte de las redes neuronales están recopilados en (Gu et al. 2017) Las redes neuronales artificiales tienen sus orígenes en el Perceptrón, que fue el modelo creado por Frank Rosenblatt en 1957 y basado en los trabajos que previamente habían realizado Warren McCullon (neurofisiólogo) y Walter Pitts (matemático). El Perceptrón está construido por una neurona artificial cuyas entradas y salida pueden ser datos numéricos, no como pasaba con la neurona de McCulloch y Pitts (eran sólo datos lógicos). Las neuronas pueden tener pesos y además se le aplica una función de activación Sigmoid (a diferencia de la usada anteriormente al Paso binario). En esta neurona nos encontramos que se realizan los siguientes cálculos:

$$z = \sum_{i=1}^{n} w_i x_i + b_i$$

$$\hat{y} = \delta(z)$$

donde representan los datos numéricos de entrada, son los pesos, es el sesgo (bias), es la función de activación y finalmente es el dato de salida. El modelo de perceptrón es el más simple, en el que hay una sola capa oculta con una única neurona. El siguiente paso nos lleva al Perceptrón Multicapa donde ya pasamos a tener más de una capa oculta, y además podemos tener múltiples neuronas en cada capa oculta. Cuando todas las neuronas de una capa están interconectadas con todas las de la siguiente capa estamos ante una red neuronal densamente conectada. A lo largo de las siguientes secciones nos encontraremos con redes en las que no todas las neuronas de una capa se conectan con todas de la siguiente. Veamos como describiríamos ahora los resultados de las capas donde representan los datos de la neurona en la capa (siendo los valores de entrada), son los pesos en la capa , es el sesgo (bias) en la capa , es la función de activación en la capa (puede que cada capa tenga una función de activación diferente), es el número de neurona de la capa anterior que conectan con la y finalmente es el dato de salida de la capa . Es decir, en cada capa para calcular el nuevo valor necesitamos usar los valores de la capa anterior.

Aplicaciones de las Redes Neuronales

Cada día las redes neuronales están más presentes en diferentes campos y ayudan a resolver una gran variedad de problemas. Podríamos pensar que de forma más básica una red neuronal nos puede ayudar a resolver problemas de regresión y clasificación, es decir, podríamos considerarlo como otro modelo más de los existentes que a partir de unos datos de entrada somos capaces de obtener o un dato numérico (o varios) para hacer una regresión (calcular en precio de una vivienda en función de diferentes valores de la misma) o que somos capaces de conseguir que en función de los datos de entrada nos deje clasificada una muestra (decidir si conceder o no una hipoteca en función de diferentes datos del cliente). Si los datos de entrada son imágenes podríamos estar usando las redes neuronales como una forma de identificar esa imagen: • Identificando que tipo de animal es • Identificando que señal de tráfico es • Identificando que tipo de fruta es • Identificando que una imagen es de exterior o interior de una casa • Identificando que es una cara de una persona • Identificando que una imagen radiográfica represente un tumor maligno • Identificando que haya texto en una imagen Luego podríamos pasar a

revolver problemas más complejos combinando las capacidades anteriores: • Detectar los diferentes objetos y personas que se encuentran en una imagen • Etiquedado de escenas (aula con alumnos, partido de futbol, etc…) Después podríamos dar el paso al video que lo podríamos considerar como una secuencia de imágenes: • Contar el número de personas que entran y salen de una habitación • Reconocer que es una carretera • Identificar las señales de tráfico • Detectar si alguien lleva un arma • Seguimiento de objetos • Detección de estado/actitud de una persona • Reconocimiento de acciones (interpretar lenguaje de signos, interpretar lenguaje de banderas) • Vehículos inteligentes Si los datos de entrada son secuencias de texto • Sistemas de traducción • Chatbots (resolución de preguntas a usuarios) • Conversión de texto a audio Si los datos de entrada son audios • Sistemas de traducción • Altavoces inteligentes • Conversión de audio a texto

A continuación, pasamos a revisar diferentes elementos de las redes neuronales que suelen ser comunes a todos los tipos de redes neuronales.

Datos

Cuando se trabaja con redes neuronales necesitamos representar los valores de las variables de entrada en forma numérica. En una red neuronal todos los datos son siempre numéricos. Esto significa que todas aquellas variables que sean categóricas necesitamos convertirlas en numéricas. Además, es muy conveniente normalizar los datos para poder trabajar con valores entre 0 y 1, que van a ayudar a que sea más fácil que se pueda converger a la solución. Es importante que los datos seán números en coma flotante, sobre todo si se van a trabajar con GPUs (Graphics Process Units), ya que permitirán hacer un mejor uso de los multiples cores que les permiten operar en coma flotante de forma paralela. Actualmente, hay toda una serie de mejoras en las GPUs que permite aumentar el rendimiento de las redes neuronales como son el uso de operaciones en FP16 (Floating Point de 16 bits en lugar de 32) de forma que pueden hacer dos operaciones de forma simultánea (el formato estándar es FP32) y además con la reducción de memoria (punto muy importante) al meter en los 32 bits 2 datos en lugar de sólo uno. También se han añadido técnicas de Mixed Precision (Narang et al. 2018), los Tensor Cores (para las gráficas de NVIDIA) son otra de las mejoras que se han ido incorporando a la GPUs y que permiten acelerar los procesos tanto de entrenamiento como de predicción con las redes neuronales.

El primer objetivo será convertir las variables categóricas en variables numéricas, de forma que el AE pueda trabajar con ellas. Para realizar la conversión de categórica a numérica básicamente tenemos dos métodos para realizarlo: • Codificación one-hot. • Codificación entera. La codificación one-hot consiste en crear tantas variables como categorías tenga la variable, de forma que se asigna el valor 1 si tiene esa categoría y el 0 si no la tiene.

La codificación entera lo que hace es codificar con un número cada categoría. Realmente esta asignación no tiene ninguna interpretación numérica ya que en general las categorías no tienen porque representar un orden al que asociarlas. Normalmente se trabaja con codificación one-hot para representar los datos categóricos de forma que será necesario preprocesar los datos de partida para realizar esta conversión, creando tantas variables como categorías haya por cada variable. Si nosotros tenemos nuestra muestra de datos de que tiene variables de forma que , y son variables categóricas que tienen número de categorías respectivamente, tendremos finalmente las siguientes variables sólo numéricas:

De esta forma, se aumentarán el número de variables con las que vamos a trabajar en función de las categorías que tengan las variables categóricas. Normalmente nos encontramos que en una red neuronal las variables de salida son: • un número (regresión) • una serie de números (regresión múltiple) • un dato binario (clasificación binaria) • una serie de datos binarios que representa una categoría de varias (clasifiación múltiple)

Arquitectura de red

Para la construcción de una red neuronal necesitamos definir la arquitectura de esa red. Esta arquitectura, si estamos pensando en una red neuronal densamente conectada, estará definida por la cantidad de capas ocultas y el número de neuronas que tenemos en cada capa. Más adelante veremos que dependiendo del tipo de red neuronal podrá haber otro tipo de elementos en estas capas. Función de coste y pérdida Otro de los elementos clave que tenemos que tener en cuenta a la hora de usar nuestra red neuronal son las funciones de pérdida y funciones de coste (objetivo). La función de pérdida va a ser la fución que nos dice cómo de diferente es el resultado del dato que nosotros queríamos conseguir respecto al dato original. Normalmente se suelen usar diferentes tipos de funciones de pérdida en función del tipo de resultado con el que se vaya a trabajar. La función de coste es la función que vamos a tener que optimizar para conseguir el mínimo valor posible, y que recoge el valor de la función de pérdida para toda la muestra. Tanto las funciones de pérdida como las funciones de coste, son funciones que devuelven valores de .

Si tenemos un problema de regresión en el que tenemos que predecir un valor o varios valores numéricos, algunas de las funciones a usar son: • Error medio cuadrático () [120] , es el valor real e es el valor predicho • Error medio absoluto () [121] , es el valor real e es el valor predicho Para los problemas de clasifiación: • Binary Crossentropy (Sólo hay dos clases) [122] es el valor real e es el valor predicho • Categorical Crosentropy (Múltiples clases representadas como one-hot) [123] es el valor real para la clase e es el valor predicho para la clase • Sparse Categorical Crossentropy (Múltiples clases representadas como un entero) [124] es el valor real para la clase e es el valor predicho para la clase • Kullback-Leibler Divergence Esta función se usa para calcular la diferencia entre dos distribuciones de probabilidad y se usa por ejemplo en algunas redes como Variational Autoencoders (Doersch 2016) o Modelos GAN (Generative Adversarial Networks) [125] [126] [127] • Hinge Loss [128] Las correspondientes funciones de coste que se usarían, estarían asociadas a todas las muestras que se estén entrenando o sus correpondientes batch, así como posibles términos asociados a la regularización para evitar el sobreajuste del entrenamiento. Es decir, la función de pérdida se calcula para cada muestra, y la función de coste es la media de todas las muestras. Por ejemplo, para el Error medio cuadrático () tendríamos el siguiente valor: [129]

Optimizador

El Descenso del gradiente es la versión más básica de los algoritmos que permiten el aprendizaje en la red neuronal haciendo el proceso de backpropagation (propagación hacia atrás). A continuación veremos una breve explicación del algoritmo así como algunas variantes del mismo recogidas en (Ruder 2017) Recordamos que el descenso del gradiente nos permitirá actualizar los parámetros de la red neuronal cada vez que demos una pasada hacia delante con todos los datos de entrada, volviendo con una pasada hacia atrás. [130] donde es la función de coste, es el parámetro de ratio de aprendizaje que permite definir como de grandes se quiere que sean los pasos en el aprendizaje. Cuando lo que hacemos es actualizar los parámetros para cada pasada hacia delante de una

sola muestra, estaremos ante lo que llamamos Stochastic Gradient Descent (SGD). En este proceso convergerá en menos iteraciones, aunque puede tener alta varianza en los parámetros. [131] donde e son los valores en la pasada de la muestra . Podemos buscar un punto intermedio que sería cuando trabajamos por lotes y cogemos un bloque de datos de la muestra, les aplicamos la pasada hacia delante y aprendemos los parámetros para ese bloque. En este caso lo llamaremos Mini-batch Gradient Descent [132] donde son los valores de ese batch . En general a estos métodos nos referiremos a ellos como SGD. Sobre este algoritmo base se han hecho ciertas mejoras como: **Learning rate decay** Podemos definir un valor de decenso del ratio de aprendizaje, de forma que normalmente al inicio de las iteraciones de la red neuronal los pasos serán más grandes, pero conforme nos acercamos a la solución optima deberemos dar pasos más pequeños para ajustarnos mejor. [133] donde ahora se irá reduciendo en función del valor del decay Momentum El **momentum** se introdujo para suavizar la convergencia y reducir la alta varianza de SGD. [134] [135] donde es lo que se llama el vector velocidad con la dirección correcta. **NAG (Nesterov Accelerated Gradient)** Ahora daremos un paso más con el NAG, calculando la función de coste junto con el vector velocidad. [136] [137] donde ahora vemos que la función de coste se calcula usando los parámetros de sumado a

Veamos algunos algoritmos de optimización más que, aunque provienen del SGD, se consideran independientes a la hora de usarlos y no como parámetros extras del SGD. **Adagrad (Adaptive Gradient)** Esta variante del algoritmo lo que hace es adaptar el ratio de aprendizaje para cada uno de los pesos en lugar de que sea global para todos. [138] donde tenemos que es una matriz diagonal donde cada elemento es la suma de los cuadrados de los gradientes en el paso , y es un término de suavizado par evitar divisiones por 0.

RMSEProp (Root Mean Square Propagation) En este caso tenemos una variación del Adagrad en el que intenta reducir su agresividad reduciendo monotonamente el ratio de aprendizaje. En lugar de usar el gradiente acumulado desde el principio de la ejecución, se restringe a una ventana de tamaño fijo para los últimos n gradientes calculando su media. Así calcularemos primero la media en ejecución de los cuadros de los gradientes como: [139] y luego ya pasaremos a usar este valor en la actualización [140]

AdaDelta

Aunque se desarrollaron de forma simultánea el AdaDelta y el RMSProp son muy parecidos en su primer paso incial, llegando el de AdaDelta un poco más lejos en su desarrollo. [141] y luego ya pasaremos a usar este valor en la actualización [142] [143] **Adam (Adaptive Moment Estimation)** [144] [145] [146] donde y son estimaciones del primer y segundo momento de los gradientes respectivamente, y y parámetros a asignar. [147] [148] [149] **Adamax** [150] [151] [152] [153] donde y son estimaciones del primer y segundo momento de los gradientes respectivamente, y y parámetros a asignar. [154] [155] **Nadam (Nesterov-accelerated Adaptive Moment Estimatio)** Combina Adam y NAG. [156] [157] [158]

Función de activación Las funciones de activación dentro de una red neuronal son uno de los elementos clave en el diseño de la misma. Cada tipo de función de activación podrá ayudar a la convergencia de forma más o menos rápida en función del tipo de problema que se plantee. En un AE las funciones de activación en las capas ocultas van a conseguir establecer las restricciones no lineales al pasar de una capa a la siguiente, normalmente se evita usar la función de activación lineal en las capas intermedias ya que queremos conseguir transformaciones no lineales. A continuación, exponemos las

principales funciones de activación que mejores resultados dan en las capas ocultas: • Paso binario (Usado por los primeros modelos de neuronas) [159] • Identidad [160] • Sigmoid (Logística) [161] • Tangente Hiperbólica (Tanh) [162] • Softmax [163]

- `ReLu ( Rectified Linear Unit)`
  `[164]`

- `LReLU (Leaky Rectified Linear Unit)`
  `[165]`
- `PReLU (Parametric Rectified Linear Unit)`
  `[166]`
- `RReLU (Randomized Rectified Linear Unit)`
  `[167]`

*La diferencia entre LReLu, PReLu y RRLeLu es que en LReLu el parámetro es uno que se asigna fijo, en el caso de PReLu el parámetro también se aprende durante el entrenamiento y finalmente en RReLu es un parámetro con valores entre 0 y 1, que se obtiene de un muestreo en una distribución normal. Se puede profundizar en este grupo de funciones de activación en (Xu et al. 2015) • ELU (Exponential Linear Unit) [168] FIGURA nº 64: COMPARACIÓN ENTRE LAS FUNCIONES ReLU, LReLU/PReLU, RReLU y ELU

FUENTE: Jiuxiang, G. et al (2019)

Función de activación en salida En la capa de salida tenemos que tener en cuenta cual es el tipo de datos final que queremos obtener, y en función de eso elegiremos cual es la función de activación de salida que usaremos. Normalmente las funciones de activación que se usarán en la última capa seran: • Lineal con una unidad, para regresión de un solo dato numérico [169] donde es un valor escalar. • Lineal con multiples unidades, para regresión de varios datos numéricos [170] donde es un vector. • Sigmoid para clasifiación binaria [171] • Softmax para calsifiación múltiple [172]

Regularización Las técnicas de regularización nos permiten conseguir mejorar los problemas que tengamos por sobreajuste en el entrenamiento de nuestra red neuronal. A continuación, vemos algunas de las técnicas de regularización existentes en la actualidad: • Norma LP Básicamente estos métodos tratan de hacer que los pesos de las neuronas tengan valores muy pequeños consiguiendo una distribución de pesos más regular. Esto lo consiguen al añadir a la función de pérdida un coste asociado a tener pesos grandes en las neuronas. Este peso se puede construir o bien con la norma L1 (proporcional al valor absoluto) o con la norma L2 (proporcional al cuadrado de los coeficientes de los pesos). En general se define la norma LP) [173] [174] Para los casos más habituales tendríamos la norma L1 y L2. [175] [176]

Dropout Una de las técnicas de regularización que más se están usando actualmente es la llamada Dropout, su proceso es muy sencillo y consiste en que en cada iteración de forma aleatoria se dejan de usar un porcentaje de las neuronas de esa capa, de esta forma es más dificil conseguir un sobreajuste porque las neuronas no son capaces de memorizar parte de los datos de entrada. **Dropconnect** El Dropconnect es otra técnica que va un poco más allá del concepto de Dropout y en lugar de usar en cada capa de forma aleatoria una serie de neuronas, lo que se hace es que de forma aleatoria se ponen los

pesos de la capa a cero. Es decir, lo que hacemos es que hay ciertos enlaces de alguna neurona de entrada con alguna de salida que no se activan.

Inicialización de pesos

Cuando empieza el entrenamiento de una red neuronal y tiene que realizar la primera pasada hacia delante de los datos, necesitamos que la red neuronal ya tenga asignados algún valor a los pesos. Se pueden hacer inicializaciones del tipo: • Ceros Todos los pesos se inicializan a 0. • Unos Todos los pesos se inicializan a 1. • Distribución normal Los pesos se inicializan con una distribución normal, normalmente con media 0 y una desviación alrededor de 0,05. Es decir, valores bastante cercanos al cero. • Distribución normal truncada Los pesos se inicializan con una distribución normal, normalmente con media 0 y una desviación alrededor de 0,05 y además se truncan con un máximo del doble de la desviación. Los valores aun són más cercanos a cero. • Distribución uniforme Los pesos se inicializan con una distribución uniforme, normalmente entre el 0 y el 1. • Glorot Normal (También llamada Xavier normal) Los pesos se inicializan partiendo de una distribución normal truncada en la que la desivación es donde es el número de unidades de entrada y fanout es el número de unidades de salida. Ver (Glorot and Bengio 2010) • Glorot Uniforme (También llamada Xavier uniforme) Los pesos se inicializan partiendo de una distribución uniforme done los límites son done y es el número de unidades de entrada y fanout es el número de unidades de salida. Ver (Glorot and Bengio 2010)

Batch normalization Hemos comentado que cuando entrenamos una red neuronal los datos de entrada deben ser todos de tipo numérico y además los normalizamos para tener valores "cercanos a cero", teniendo una media de 0 y varianza de 1, consiguiendo uniformizar todas las variables y conseguir que la red pueda converger más fácilmente. Cuando los datos entran a la red neuronal y se comienza a operar con ellos, se convierten en nuevos valores que han perdido esa propiedad de normalización. Lo que hacemos con la normalización por lotes (batch normalization) (Ioffe and Szegedy 2015) es que añadimos un paso extra para normalizar las salidas de las funciones de activación. Lo normal es que se aplicara la normalización con la media y la varianza de todo el bloque de entrenamiento en ese paso, pero normalmente estaremos trabajando por lotes y se calculará la media y varianza con ese lote de datos.

2.2.1 Principales arquitecturas de Deep Learning

Actualmente existen muchos tipos de estructuras de redes neuronales artificiales dado que logran resultados extraordinarios en muchos campos del conocimiento. Los primeros éxitos en el aprendizaje profundo se lograron a través de las investigaciones y trabajos de Geoffre Hinton (2006) que introduce las Redes de Creencia Profunda en cada capa de la red de una Máquina de Boltzmann Restringida (RBM) para la asignación inicial de los pesos sinápticos. Hace tiempo que se está trabajando con arquitecturas como los Autoencoders, Hinton y Zemel (1994), las RBMs de Hinton y Sejnowski (1986) y las DBNs (Deep Belief Networks), Hinton et al. (2006) y otras como las redes recurrentes y convolucionales. Estas técnicas constituyen en sí mismas arquitecturas de redes neuronales, aunque también algunas de ellas, como se ha afirmado en la introducción, se están empleando para inicializar los pesos de arquitecturas profundas de redes neuronales supervisadas con conexiones hacia adelante. Las principales arquitecturas de deep learning se resumen en la siguiente figura. Figura 65. modelos de redes neuronales según tipo de aprendizaje

Vamos a describir de forma somera las principales arquitecturas dado que posteriormente se desarrollan más ampliamente en este epígrafe o se tratan en docuemento adjunto que se acompaña en este módulo **Convolutional Neural Network** Tal vez los modelos más utilizados actualmente en el campo del Deep Learning sean las redes neuronales convolucionales, denominadas en inglés Convolutional Neural Networks (CNN). El objetivo de las redes CNN es aprender características de orden superior utilizando la operación de convolución. Estas estructuras de redes neuronales son especialmente eficaces para clasificar y segmentar imágenes, en general, son notablemente eficaces en tareas de visión artificial. Las CNN son una modificación del perceptrón multicapa explicado en un módulo anterior. Este modelo es muy similar al trabajo que ejecuta el cerebro humano: las neuronas se corresponden a campos receptivos similares a como lo realizan las neuronas en la corteza visual de nuestro cerebro. Esta arquitectura ya se utilizó en 1990 donde la empresa AT & T las aplicó para crear un modelo de lectura de cheques, desarrollándose posteriormente muchos sistemas OCR basados en CNN. Las redes de convolución tienen una estructura de varias capas: las capas de Convolución que transforman los datos de entrada a través de una operación matemática llamada Convolución y la capa de pooling, que trata de sintetizar y condensar la información de la capa de convolución. Finalmente, se transforman los datos para aplicar una red densamente conectada que nos ofrece el resultado final en relación con el objetivo que se busca. Estas redes neuronales artificiales se desarrollaron al abrigo de los concursos denominados ILSVRC (Large Scale Visual Recognition Challenge) donde aparecieron las principales aportaciones efectuadas en las redes convolucionales y que hoy podemos utilizar todos los investigadores. Algunos de las estructuras más novedosas y que son modelos ya preentrenados, con estructuras de capas más numerosas y que podemos integrar en nuestras aplicaciones, se denominan: LeNet-5, AlexNet, VGG, GoogLeNet y Resnet.

Autoencoder Los Autoencoders (AE) son uno de los tipos de redes neuronales que caen dentro del ámbito del Deep Learning, en la que nos encontramos con un modelo de aprendizaje no supervisado. Ya se empezó a hablar de AE en la década de los 80 (Bourlard and Kamp 1988), aunque es en estos últimos años donde más se está trabajando con ellos. La arquitectura de un AE es una Red Neuronal Artificial (ANN por sus siglas en inglés) que se encuentra dividida en dos partes, encoder y decoder (Charte et al. 2018), (Goodfellow, Bengio, and Courville 2016). El encoder va a ser la parte de la ANN que va codificar o comprimir los datos de entrada, y el decoder será el encargado de regenerar de nuevo los datos en la salida. Esta estructura de codificación y decodificación le llevará a tener una estructura simétrica. El AE es entrenado para ser capaz de reconstruir los datos de entrada en la capa de salida de la ANN, implementando una serie de restricciones (la reducción de elementos en las capas ocultas del encoder) que van a evitar que simplemente se copie la entrada en la salida. Algunas de sus principales aplicaciones sobre las que se está investigando son: • Reducción de dimensiones / Compresión de datos • Búsqueda de imágenes • Detección de Anomalías • Eliminación de ruido **Redes recurrentes** En la actualidad las **redes neuronales recurrentes (Recurrent Neural Networks)** han logrado un puesto destacado en machine learning. Estas redes que no disponen de una estructura de capas, sino que permiten conexiones arbitrarias entre todas las neuronas, incluso creando ciclos. En esta arquitectura se permiten conexiones recurrentes lo que aumenta el número de pesos o de parámetros ajustables de la red, lo que incrementa la capacidad de representación, pero también la complejidad del aprendizaje. Las peculiaridades de esta red permiten incorporar a la red el concepto de temporalidad, y también que la red tenga memoria, porque los números que introducimos en un momento dado en las neuronas de entrada son trans-

formados, y continúan circulando por la red. Existen diferentes planteamientos de redes recurrentes, por ejemplo, son muy populares por sus aplicaciones las redes de Elmann y de Jordan. En los últimos años se han popularizado las redes recurrentes denominadas Long-Short Term Memory (LSTM) que son una extension de las redes recurrentes y su caracterítica principal es que amplian su memoria para registrar experiencias que han ocurrido hace mucho tiempo. Normalmente contienen tres puertas que determinan si se permite o no una nueva entrada o se elimina la información que llega dado que se considera no importante. Son análogas a una función sigmoide lo que implica que van de 0 a 1, lo que permite incorporarlas al proceso de backpropagation. También se encuentran entre las redes recurrentes, las denominadas GRU (Gated Recurrent Unit) que aparecieron en 2014 y simplifican a las LSMT: son computacionalmente menos costosas y más eficientes.

Boltzmann Machine y Restricted Boltzmann Machine El aprendizaje de la denominada máquina de Boltzmann (BM) se realiza a través de un algoritmo estocástico que proviene de ideas basadas en la mecánica estadística. Este prototipo de red neuronal tiene una característica distintiva y es que el uso de conexiones sinápticas entre las neuronas es simétrico. Las neuronas son de dos tipos: visibles y ocultas. Las neuronas visibles son las que interactúan y proveen una interface entre la red y el ambiente en el que operan, mientras que las neuronas actúan libremente sin interacciones con el entorno. Esta máquina dispone de dos modos de operación. El primero es la condición de anclaje donde las neuronas están fijas por los estímulos específicos que impone el ambiente. El otro modo es la condición de libertad, donde tanto las neuronas ocultas como las visibles actúan libremente sin condiciones impuestas por el medio ambiente. Las maquinas restringidas de Boltzmann (RBM) solamente toman en cuenta aquellos modelos en los que no existen conexiones del tipo visible-visible y oculta-oculta. Estas redes también asumen que los datos de entrenamiento son independientes y están idénticamente distribuidos. Una forma de estimar los parámetros de un modelo estocástico es calculando la máxima verosimilitud. Para ello, se hace uso de los Markov Random Fiels (MRF), ya que al encontrar los parámetros que maximizan los datos de entrenamiento bajo una distribución MRF, equivale a encontrar los parámetros que maximizan la verosimilitud de los datos de entrenamiento, Fischer e Igel (2012). Maximizar dicha verosimilitud es el objetivo que persigue el algoritmo de entrenamiento de una RBM. A pesar de utilizar la distribución MRF, computacionalmente hablando se llega a ecuaciones inviables de implementar. Para evitar el problema anterior, las esperanzas que se obtienen de MRF pueden ser aproximadas por muestras extraídas de distribuciones basadas en las técnicas de Markov Chain Monte Carlo Techniques (MCMC). Las técnicas de MCMC utilizan un algoritmo denominado muestreo de Gibbs con el que obtenemos una secuencia de observaciones o muestras que se aproximan a partir de una distribución de verosimilitud de múltiples variables aleatorias. La idea básica del muestreo de Gibss es actualizar cada variable posteriormente en base a su distribución condicional dado el estado de las otras variables. Deep Belief Network Una red Deep Belief Network tal como demostró Hinton se puede considerar como un “apilamiento de redes restringidas de Boltzmann”. Tiene una estructura jerárquica que como sabemos es una de las características del deep learning. Como en el anterior modelo, esta red también es un modelo en grafo estocástico, que aprende a extraer una representación jerárquica profunda de los datos de entrenamiento. Cada capa de la RBM extrae un nivel de abstracción de características de los datos de entrenamiento, cada vez más significativo; pero para ello, la capa siguiente necesita la información de la capa anterior lo que implica el uso de las variables latentes. Estos modelos caracterizan la distribución conjunta hk entre el vector de observaciones x y las

capas ocultas, donde x=h0, es una distribución condicional para las unidades visibles limitadas sobre las unidades ocultas que pertenecen a la RBM en el nivel k, y es la distribución conjunta oculta visible en la red RBM del nivel superior o de salida. El entrenamiento de esta red puede ser híbrido, empezando por un entrenamiento no supervisado para después aplicar un entrenamiento supervisado para un mejor y más óptimo ajuste, aunque pueden aplicarse diferentes tipos de entrenamiento, Bengio et al. (2007) y Salakhutdinov (2014) Para realizar un entrenamiento no supervisado se aplica a las redes de creencia profunda con Redes restringidas de Boltzmann el método de bloque constructor que fue presentado por Hinton (2006) y por Bengio (2007)

2.3 Redes Neuronales Convolucionales

2.3.1 Introducción

Esta arquitectura de redes de neuronas convolucionales, CNN, Convolutional Neural Networks es en la actualidad el campo de investigación más fecundo dentro de las redes neuronales artificiales de Deep learning y donde los investigadores, empresas e instituciones están dedicando más recursos e investigación. Para apoyar esta aseveración, en google trend se observa que el término convolutional neural network en relación con el concepto de artificial neural network crece y está por encima desde el año 2016. Es en este último lustro donde el Deep learning ha tomado una importancia considerable.

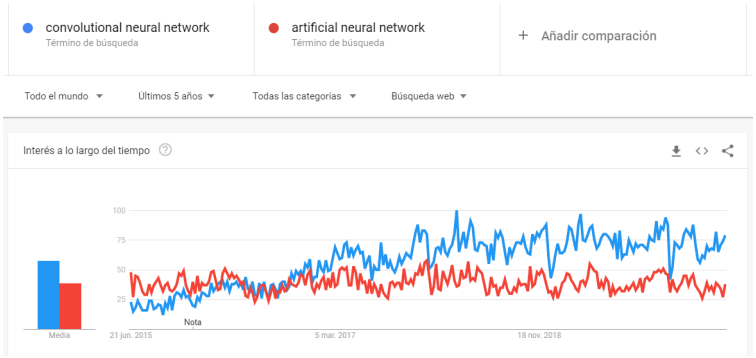


Imagen 2.1: búsqueda de términos de redes neuronales en google trend

Fuente: Google Trend En este modelo de redes convolucionales las neuronas se corresponden a campos receptivos similares a las neuronas en la corteza visual de un cerebro humano. Este tipo de redes se han mostrado muy efectivas para tareas de detección y categorización de objetos y en la clasificación y segmentación de imágenes. Por ejemplo, estas redes en la década de 1990 las aplicó AT & T para desarrollar un modelo para la lectura de cheques. También más tarde se desarrollaron muchos sistemas OCR basados en CNN. En esta arquitectura cada neurona de una capa no recibe conexiones entrantes de todas las neuronas de la capa anterior, sino sólo de algunas. Esta estrategia favorece que una neurona se especialice en una región del conjunto de números (píxeles) de la capa anterior, lo que disminuye notablemente el número de pesos y de operaciones a realizar. Lo más normal es que neuronas consecutivas de una capa intermedia se especialicen en regiones solapadas de la capa anterior. Una forma intuitiva para entender cómo trabajan estas redes neuronales es ver cómo nos representamos y vemos las imágenes. Para reconocer una cara primero tenemos que tener una imagen interna de lo

que es una cara. Y a una imagen de una cara la reconocemos porque tiene nariz, boca, orejas, ojos, etc. Pero en muchas ocasiones una oreja está tapada por el pelo, es decir, los elementos de una cara se pueden ocultar de alguna manera. Antes de clasificarla, tenemos que saber la proporción y disposición y también cómo se relacionan la partes entre sí. Para saber si las partes de la cara se encuentran en una imagen tenemos que identificar previamente líneas bordes, formas, texturas, relación de tamaño, etcétera. En una red convolucional, cada capa lo que va a ir aprendiendo son los diferentes niveles de abstracción de la imagen inicial. Para comprender mejor el concepto anterior hemos seleccionado esta imagen de Raschka y Mirjalili (2019) donde se observa como partes del perro se transforman en neuronas del mapa de características

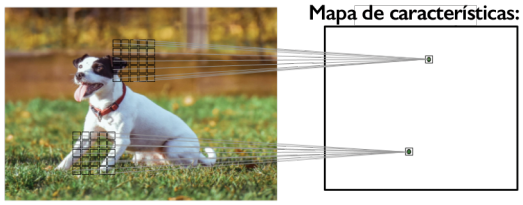


Imagen 2.2: Correspondencia de zonas de la imagen y mapa de características

Fuente: Raschka y Mirjalili (2019)

El objetivo de las redes CNN es aprender características de orden superior utilizando la operación de convolución. Puesto que las redes neuronales convolucionales pueden aprender relaciones de entrada-salida (donde la entrada es una imagen), en la convolución, cada pixel de salida es una combinación lineal de los pixeles de entrada. La convolución consiste en filtrar una imagen utilizando una máscara. Diferentes máscaras producen distintos resultados. Las máscaras representan las conexiones entre neuronas de capas anteriores. Estas capas aprenden progresivamente las características de orden superior de la entrada sin procesar. Las redes neuronales convolucionales se forman usando dos tipos de capas: convolucionales y pooling. La capa de convolución transforma los datos de entrada a través de una operación matemática llamada convolución. Esta operación describe cómo fusionar dos conjuntos de información diferentes. A esta operación se le suele aplicar una función de transformación, generalmente la RELU. Después de la capa o capas de convolución se usa una capa de pooling, cuya función es resumir las respuestas de las salidas cercanas. Antes de obtener el output unimos la última capa de pooling con una red densamente conectada. Previamente se ha aplanado (Flatering) la última capa de pooling para obtener un vector de entrada a la red neural final que nos ofrecerá los resultados.

Las redes neuronales convolucionales debido a su forma de concebirse son aptas para poder aprender a clasificar todo tipo de datos donde éstos estén distribuidos de una forma continua a lo largo del mapa de entrada, y a su vez sean estadísticamente similares en cualquier lugar del mapa de entrada. Por esta razón, son especialmente eficaces para clasificar imágenes. También pueden ser aplicadas para la clasificación de series de tiempo o señales de audio. En relación con el color y la forma de codificarse, en las redes convolucionales se realiza en tensores 3D, dos ejes para el ancho (width) y el alto (height) y el otro eje llamado de profundidad (depht) que es el canal del color con valor tres si trabajamos con imágenes de color RGB (Red, Green y Blue) rojo, verde y azul. Si disponemos de imágenes en escala de grises el valor de depht es uno. La base de datos MNIST (National Institute of Standards and Technology database) con la que

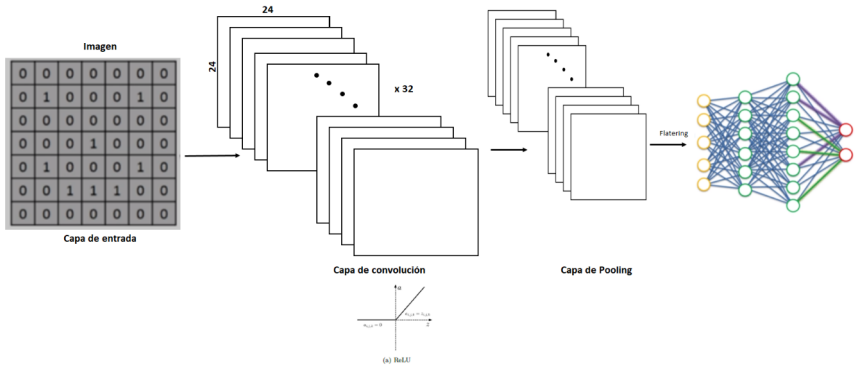


Imagen 2.3: Arquitectura de una CNN

trabajaremos en este epígrafe contiene imágenes de 28 x 28 pixeles, los valores de height y de widht son ambos 28, y al ser una base de datos en blanco y negro el valor de depht es 1. Las imágenes son matrices de píxeles que van de cero a 255 y que para la red neuronal se normalizan para que sus valores oscilen entre cero y uno. 7.4.2. Convolución En las redes convolucionales todas las neuronas de la capa de entrada (los píxeles de las imágenes) no se conectan con todas las neuronas de la capa oculta del primer nivel como lo hacen las redes clásicas del tipo perceptrón multicapa o las redes que conocemos de forma genérica como redes densamente conectadas. Las conexiones se realizan por pequeñas zonas de la capa de entrada.

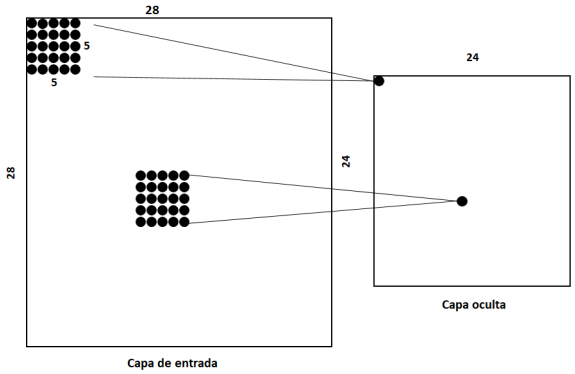


Imagen 2.4: Conexión de las neuronas de la capa de entrada con la capa oculta

Veamos un ejemplo para la base de datos de los dígitos del 1 a 9. Vamos a conectar cada neurona de la capa oculta con una región de 5 x 5 neurona, es decir, con 25 neuronas de la capa de entrada, que podemos denominarla ventana. Esta ventana va a ir recorriendo todo el espacio de entrada de 28 x 28 empezando por arriba y desplazándose de izquierda a derecha y de arriba abajo. Suponemos que los desplazamientos de la ventana son de un paso (un pixel) aunque este es un parámetro de la red que podemos modificar (en la programación lo llamaremos stride). Para conectar la capa de entrada con la de salida utilizaremos una matriz de pesos (W) de tamaño 3 x 3 que recibe el nombre de filtro (filter) y el valor del sesgo. Para obtener el valor de cada neurona de la capa oculta realizaremos el producto escalar entre el filtro y la ventana de la capa de entrada. Utilizamos el mismo filtro para obtener todas las neuronas de la capa oculta, es decir en todos los productos escalares siempre utilizamos la misma matriz, el mismo filtro. Se definen matemáticamente estos productos escalares a través de la siguiente expresión: [177]

Como en este tipo de red un filtro sólo nos permite revelar una característica muy concreta de la imagen, lo que se propone es usar varios filtros simultáneamente, uno para cada característica que queramos detectar. Una forma visual de representarlo (si suponemos que queremos aplicar 32 filtros) es como se muestra a continuación:

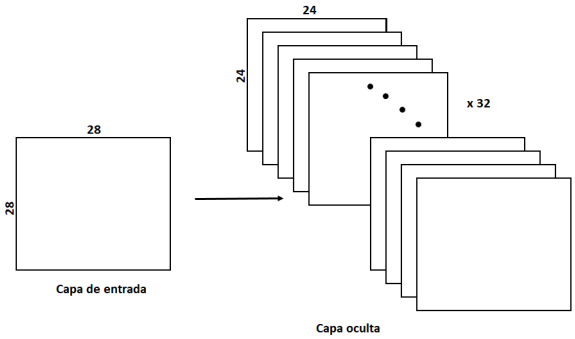


Imagen 2.5: Primera capa de la red convolucional con 32 filtros

Al resultado de la aplicación de los diferentes filtros se les suele aplicar la función de activación denominada RELU y que ya se comentó en la introducción. Una interesante fuente de información es la documentación del software gratuito GIMP donde expone diferentes efectos que se producen en las imágenes al aplicar diversas convoluciones. Un ejmplo claro y didáctico lo podemos obtener de la docuemntación del software libre de dibujo y tratamiento de imágenes denominado GIMP (https://docs.gimp.org/2.6/es/plug-in-convmatrix.html). Algunos de estos efectos nos ayudan a entender la operación de los filtros en las redes convolucionales y cómo afectan a las imágenes, en concreto, el ejemplo que presenta lo realiza sobre la figura del Taj Mahal El filtro enfocar lo que consigue es afinar los rasgos, los contornos lo que nos permite agudizar los objetos de la imagen. Toma el valor central de la matriz de cinco por cinco lo multiplica por cinco y le resta el valor de los cuatro vecinos. Al final hace una media, lo que mejora la resolución del pixel central porque elimina el ruido o perturbaciones que tiene de sus pixeles vecinos. El filtro enfocar (Sharpen)

Lo contario al filtro enfocar lo obtenemos a través de la matriz siguiente, difuminando la imagen al ser estos píxeles mezclados o combinados con los pixeles cercanos. Promedia todos los píxeles vecinos a un pixel dado lo que implica que se obtienen bordes borrosos. Filtro desenfocar

Filtro Detectar bordes (Edge Detect) Este efecto se consigue mejorando los límites o las aristas de la imagen. En cada píxel se elimina su vecino inmediatamente anterior en horizontal y en vertical. Se eliminan las similitudes vecinas y quedan los bordes resaltados. Al pixel central se le suman los cuatro píxeles vecinos y lo que queda al final es una medida de cómo de diferente es un píxel frente a sus vecinos. En el ejemplo, al hacer esto da un valor de cero de ahí que se observen tantas zonas oscuras.

Filtro Repujado (Emboss) En este filto se observa que la matriz es simétrica y lo que intenta a través del diseño del filtro es mejorar los píxeles centrales y de derecha abajo restándole los anteriores. Se obtiene lo que en fotografía se conoce como un claro oscuro. Trata de mejorar las partes que tienen mayor relevancia.

2.3.2 Pooling

Con la operación de pooling se trata de condensar la información de la capa convolucional. A este procedimiento también se le conoce como submuestreo. Es simplemente una operación en la que reducimos los parámetros de la red. Se aplica normalmente a través de dos operaciones: max-pooling y mean-pooling, que también es conocido como average-pooling. Tal y como se observa en la imagen siguiente, desde la capa de convolución se genera una nueva capa aplicando la operación a todas las agrupaciones, donde previamente hemos elegido el tamaño de la región; en la figura siguiente es de tamaño 2, con lo que pasamos de un espacio de 24 x 24 neuronas a la mitad, 12 x 12 en la capa de pooling.

Figura 71. etapa de pooling de tamaño 2 x 2

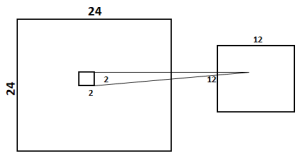


Imagen 2.6: etapa de pooling de tamaño 2 x 2

Vamos a estudiar el pooling suponiendo que tenemos una imagen de 5 x 5 píxeles y que queremos efectuar una agrupación max-pooling. Es la más utilizada, ya que obtiene buenos resultados. Observamos los valores de la matriz y se escoge el valor máximo de los cuatro bloques de matrices de dos por dos. Max Pooling

En la agrupación Average Pooling la operación que se realiza es sustituir los valores de cada grupo de entrada por su valor medio. Esta transformación es menos utilizada que el max-pooling.

La transformación max-pooling presenta un tipo de invarianza local: pequeños cambios en una región local no varían el resultado final realizado con el max – pooling: se mantiene la relación espacial. Para ilustrar este concepto hemos escogido la imagen que presenta Torres (2020) donde se ilustra como partiendo de una matriz de 12 x 12 que representa al número 7, al aplicar la operación de max-pooling con una ventana de 2 x 2 se conserva la relación espacial.

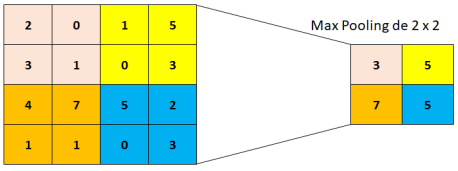


Imagen 2.7: Max Pooling

Fuente: Torres. J. (2020)

2.3.3 Padding

Para explicar el concepto del Padding vamos a suponer que tenemos una imagen de 5 x 5 píxeles, es decir 25 neuronas en la capa de entrada, y que elegimos, para realizar la

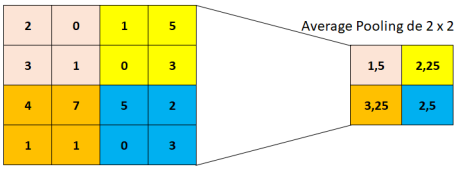


Imagen 2.8: Average Pooling

convolución, una ventana de 3 x 3. El número de neuronas de la capa oculta resultará ser de nueve. Enumeramos los píxeles de la imagen de forma natural del 1 al 25 para que resulte más sencillo de entender.

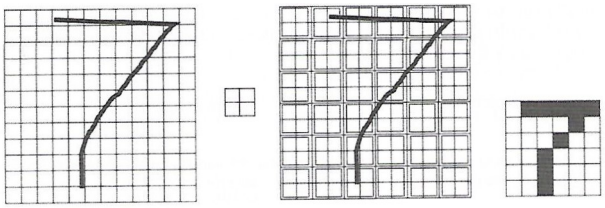


Imagen 2.9: mantenimiento del pooling con la transformación

Pero si queremos obtener un tensor de salida que tenga las mismas dimensiones que la entrada podemos rellenar la matriz de ceros antes de deslizar la ventana por ella. Vemos la figura siguiente donde ya se ha rellenado de valores cero y obtenemos, después de deslizar la ventana de 3 x3 de izquierda a derecha y de arriba abajo, las veinticinco matrices de la figura nº 71

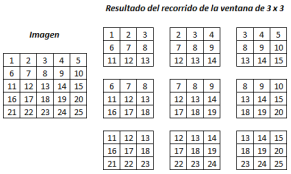


Imagen 2.10: Operación de convolución con una ventana de 3 x 3

Figura nº 74. imagen con relleno de ceros

Cuando utilizamos el programa keras disponemos de dos opciones para llevar a cabo esta operación de padding: “same” y “valid”. Si utilizamos “valid” implica no hacer padding y el método “same” obliga a que la salida tenga la misma dimensión que la entrada.

2.3.4 Stride

Hasta ahora, la forma de recorrer la matriz a través de la ventana se realiza desplazándola de un solo paso, pero podemos cambiar este hiperparámetro conocido como stride. Al aumentar el paso se decrementa la información que pasará a la capa posterior. A continuación, se muestra el resultado de las cuatro matrices que obtenemos con un stride de valor 3.

Finalmente, para resumir, una red convolucional contiene los siguientes elementos: • Entrada: Son el número de pixeles de la imagen. Serán alto, ancho y profundidad.

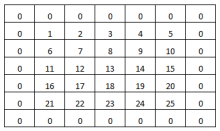


Imagen 2.11: Imagen con relleno de ceros

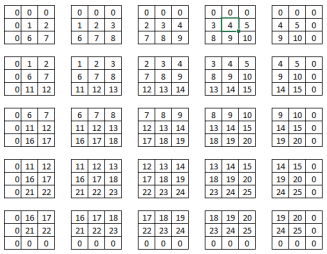
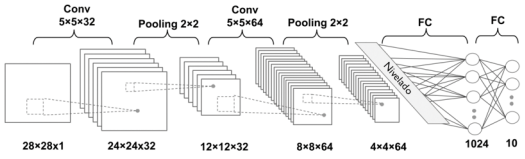


Imagen 2.12: Operación de convolución con ventana 3 x 3 y padding

Tenemos un solo color (escala de grises) o tres: rojo, verde y azul. • Capa de convolución: procesará la salida de neuronas que están conectadas en «regiones locales» de entrada (es decir pixeles cercanos), calculando el producto escalar entre sus pesos (valor de pixel) y una pequeña región a la que están conectados. En este epígrafe se presentan las imágenes con 32 filtros, pero puede realizarse con la cantidad que deseemos. • «Capa RELU» Se aplicará la función de activación en los elementos de la matriz. • POOL (agrupar) o Submuestreo: Se procede normalmente a una reducción en las dimensiones alto y ancho, pero se mantiene la profundidad. • CAPA tradicional. Se finalizará con la red de neuronas feedforward (Perceptrón multicapa que se denomina normalmente como red densamente conectada) que vinculará con la última capa de subsampling y finalizará con la cantidad de neuronas que queremos clasificar. En el gráfico siguiente se muestran todas las fases de una red neuronal convolucional.



Fuente: Raschka y Mirjalili (2019)

2.3.5 Redes convolucionales con nombre propio

Existen en la actualidad muchas arquitecturas de redes neuronales convolucionales que ya están preparadas, probadas, disponibles e incorporadas en el software de muchos programas como Keras y Tensorflow. Vamos a comentar algunos de estos modelos, bien por ser los primeros, o por sus excelentes resultados en concursos como el ILSVRC (Large

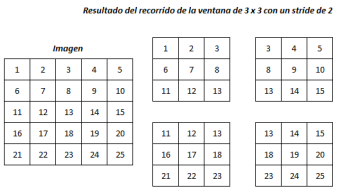


Imagen 2.13: Operación de convolución con una ventana de 3 x 3 y stride 2

Scale Visual Recognition Challenge). Estas estructuras merecen atención dado que son excelentes para estudiarlas e incorporarlas por su notable éxito. El ILSVRC fue un concurso celebrado de 2011 a 2016 de donde nacieron las principales aportaciones efectuadas en las redes convolucionales. Este concurso fue diseñado para estimular la innovación en el campo de la visión computacional. Actualmente se desarrollan este tipo de concursos a través de la plataforma web: https://www.kaggle.com/ Para ver más prototipos de redes convolucionales y los últimos avances y consejos sobre las redes convolucionales se puede consultar el siguiente artículo "Recent Advances in Convolutional Neural Networks" de Jiuxiang. G. et al. (2019) Los cinco modelos más destacados hasta el año 2017 son los siguientes: LeNet-5, Alexnet, GoogLeNet, VGG y Restnet. 1. LeNet-5. Este modelo de Yann LeCun de los años 90 consiguió excelentes resultados en la lectura de códigos postales consta de imágenes de entrada de 32 x 32 píxeles seguida de dos etapas de convolución – pooling, una capa densamente conectada y una capa softmax final que nos permite conocer los números o las imágenes. 2. AlexNet. Fue la arquitectura estrella a partir del año 2010 en el ILSVRC y popularizada en el documento de 2012 de Alex Krizhevsky, et al. titulado"Clasificación de ImageNet con redes neuronales convolucionales profundas". Podemos resumir los aspectos clave de la arquitectura relevantes en los modelos modernos de la siguiente manera: • Empleo de la función de activación ReLU después de capas convolucionales y softmax para la capa de salida. • Uso de la agrupación máxima en lugar de la agrupación media. • Utilización de la regularización de Dropout entre las capas totalmente conectadas. • Patrón de capa convolucional alimentada directamente a otra capa convolucional. • Uso del aumento de datos (Data Aumentation,) 3. VGG. Este prototipo fue desarrollado por un grupo de investigación de Geometría Visual en Oxford. Obtuvo el segundo puesto en la competición del año 2014 del ILSVRC. Las aportaciones principales de la investigación se pueden encontrar en el documento titulado " Redes convolucionales muy profundas para el reconocimiento de imágenes a gran escala " desarrollado por Karen Simonyan y Andrew Zisserman. Este modelo contribuyó a demostrar que la profundidad de la red es una componente crítica para alcanzar unos buenos resultados. Otra diferencia importante con los modelos anteriores y que actualmente es muy utilizada es el uso de un gran número de filtros y de tamaño reducido. Estas redes emplean ejemplos de dos, tres e incluso cuatro capas convolucionales apiladas antes de usar una capa de agrupación máxima. En esta arquitectura el número de filtros aumenta con la profundidad del modelo. El modelo comienza con 64 y aumenta a través de los filtros de 128, 256 y 512 al final de la parte de extracción de características del modelo. Los investigadores evaluaron varias variantes de la arquitectura si bien en los programas sólo se hace referencia a dos de ellas que son las que aportan un mayor rendimiento y que son nombradas por las capas que tienen: VGG-16 y VGG-19. 4. GoogLeNet. GoogLeNet fue desarrolado por investigadores de Google Research. de Google, que con su módulo denominado de inception reduce drásticamente los parámetros de la red (10 veces menos que AlexNet) y de ella han derivado varias versiones como la Inception-v4. Esta arquitectura ganó la competición en el año 2014 y su éxito se debió a que la red era mucho más profunda (muchas más capas) y como ya se ha indicado introdujeron en el modelo las subredes llamadas inception. Las aportaciones principales en el uso de capas convolucionales fueron propuestos en el documento de 2015 por Christian Szegedy, et al. titulado " Profundizando con las convoluciones ". Estos autores introducen una arquitectura llamada "inicio" y un modelo específico denominado GoogLenet. El módulo inicio es un bloque de capas convolucionales paralelas con filtros de diferentes tamaños y una capa de agrupación máxima de 3×3, cuyos resultados se concatenan. Otra decisión de diseño fundamental en el modelo inicial fue la conexión de la salida en diferentes puntos del modelo que lograron realizar con la

creación de pequeñas redes de salida desde la red principal y que fueron entrenadas para hacer una predicción. La intención era proporcionar una señal de error adicional de la tarea de clasificación en diferentes puntos del modelo profundo para abordar el problema de los gradientes de fuga. 5. Red Residual o ResNet. Esta arquitectura gano la competición de 2015 y fue creada por el grupo de investigación de Microsoft. Se puede ampliar la información en He, et al. en su documento de 2016 titulado " Aprendizaje profundo residual para el reconocimiento de la imagen ". Esta red es extremadamente profunda con 152 capas, confirmando al pasar los años que las redes son cada vez más profundas, más capas, pero con menos parámetros que estimar. La cuestión clave del diseño de esta red es la incorporación de la idea de bloques residuales que hacen uso de conexiones directa. Un bloque residual, según los autores, "es un patrón de dos capas convolucionales con activación ReLU donde la salida del bloque se combina con la entrada al bloque, por ejemplo, la conexión de acceso directo" Otra clave, en este caso para el entrenamiento de la red tan profunda es lo que llamaron skip connections que implica que la señal con la que se alimenta una capa también se agregue a una capa que se encuentre más adelante. Resumiendo, las tres principales aportaciones de este modelo son: • Empleo de conexiones de acceso directo. • Desarrollo y repetición de los bloques residuales. • Modelos muy profundos (152 capas) Aunque se encuentran otros modelos que también son muy populares con 34, 50 y 101 capas. Una buena parte de los modelos comentados se incluyen en la librería de Keras y se pueden encontrar en la siguiente dirección de internet: https://keras.io/api/applications/ Según los autores del programa Keras: "Las aplicaciones Keras son modelos de aprendizaje profundo que están disponibles junto con pesos preentrenados. Estos modelos se pueden usar para predicción, extracción de características y ajustes. Los pesos se descargan automáticamente cuando se crea una instancia de un modelo. Se almacenan en ~ / .keras / models /. Tras la creación de instancias, los modelos se construirán de acuerdo con el formato de datos de imagen establecido en su archivo de configuración de Keras en ~ / .keras / keras.json. Por ejemplo, si ha configurado image_data_format = channel_last, cualquier modelo cargado desde este repositorio se construirá de acuerdo con la convención de formato de datos TensorFlow,"Altura-Ancho-Profundidad".

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	46.0	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	186	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	186	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	210	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	258	308.3	15.1
EfficientNetB5	118	83.6%	96.7%	30.6M	312	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	360	958.1	40.4
EfficientNetB7	256	84.3%	97.0%	66.7M	438	1578.9	61.6

Fuente : https://keras.io/api/applications/

2.4 Redes Nueronales Recurrentes

2.5 Autoencoders

Bases del Autoencoder Los Autoencoders (AE) son uno de los tipos de redes neuronales que caen dentro del ámbito del Deep Learning, en la que nos encontramos con un modelo de aprendizaje no supervisado. Ya se empezó a hablar de AE en la década de los 80 (Bourlard and Kamp 1988), aunque es en estos últimos años donde más se está trabajando con ellos. La arquitectura de un AE es una Red Neuronal Artificial (ANN por sus siglas en inglés) que se encuentra dividida en dos partes, encoder y decoder (Charte et al. 2018), (Goodfellow, Bengio, and Courville 2016). El encoder va a ser la parte de la ANN que va codificar o comprimir los datos de entrada, y el decoder será el encargado de regenerar de nuevo los datos en la salida. Esta estructura de codificación y decodificación le llevará a tener una estructura simétrica. El AE es entrenado para ser capaz de reconstruir los datos de entrada en la capa de salida de la ANN, implementando una serie de restricciones (la reducción de elementos en las capas ocultas del encoder) que van a evitar que simplemente se copie la entrada en la salida. Si recordamos la estructura de una ANN clásica o también llamada Red Neuronal Densamente Conectada (ya que cada neurona conecta con todas las de la siguiente capa) nos encontramos en que en esta arquitectura, generalmente, el número de neuronas por capa se va reduciendo hasta llegar a la capa de salida que debería ser normalmente un número (si estamos en un problema regresión), un dato binario (si es un problema de clasificación). Figura nº 86: Red Neuronal Clasica

Fuente: Elaboración propia Si pensamos en una estructura básica de AE en la que tenemos una capa de entrada, una capa oculta y una capa de salida, ésta sería su representación:

Figura nº 87: Autoencoder básico

Fuente: Elaboración propia Donde los valores de son los datos de entrada y los datos son la reconstrucción de los mismos después de pasar por la capa oculta que tiene sólo dos dimensiones. El objetivo del entrenamiento de un AE será que estos valores de sean lo más parecidos posibles a los . Según (Charte et al. 2018) los AE se puden clasificar según el tipo de arquitectura de red en: • Incompleto simple • Incompleto profundo • Extra dimensionado simple • Extra dimensionado profundo

Figura nº 88: Tipos de Autoencoders por arquitectura

Fuente: Elaboración propia Cuando hablamos de Incompleto nos referimos a que tenemos una reducción de dimensiones que permite llegar a conseguir una “compresión” de los datos iniciales como técnica para que aprenda los patrones internos. En el caso de Extra dimensionado es cuando subimos de dimensión para conseguir que aprenda esos patrones. En este último caso sería necesario aplicar técnicas de regularización para evitar que haya un sobreajuste en el aprendizaje. Cuando hablamos de Simple estamos haciendo referencia a que hay una única capa oculta, y en el caso de Profundo es que contamos con más de una capa oculta. Normalmente se trabaja con las arquitecturas de tipo Incompleto profundo, sobre todo cuando se está trabajando con tipos de datos que son imágenes. Aunque también podríamos encontrar una combinación de Incompleto con Extra dimensionado profundo cuando trabajamos con tipos de datos que no son imágenes y así crecer en la primera o segunda capa oculta, para luego reducir. Esto

nos permitiría por ejemplo adaptarnos a estructuras de AE en las que trabajemos con número de neuronas en una capa que sean potencia de 2, y poder construir arquitecturas dinámicas en función del tamaño de los datos, adaptándolos a un tamaño prefijado. A continuación, vemos un gráfico de una estructura mixta Extra dimensionado - Incompleto profundo. Figura nº 89: Autoencoder Mixto (Incompleto y Extra dimensionado)

Fuente: Elaboración propia Idea intuitiva del uso de Autoencoders Si un AE trata de reproducir los datos de entrada mediante un encoder y decoder, ¿que nos puede aportar si ya tenemos los datos de entrada? Ya hemos comentado que la red neuronal de un AE es simétrica y está formada por un encoder y un decoder, además cuando trabajamos con los AE que son incompletos, se está produciendo una reducción del tamaño de los datos en la fase de codificación y de nuevo una regeneración a partir de esos datos más pequeños al original. Ya tenemos uno de los conceptos más importantes de los AE que es la reducción de dimensiones de los datos de entrada. Estas nuevas variables que se generan una vez pasado el encoder se les suele llamar el espacio lantente. Este concepto de reducción de dimensiones en el mundo de la minería de datos lo podemos asimiliar rápidamente a técnicas como el Análisis de Componentes Principales (PCA), que nos permite trabajar con un número más reducido de dimensiones que las originales. Igualmente, esa reducción de los datos y la capacidad de poder reconstruir el original podemos asociarlo al concepto de compresión de datos, de forma que con el encoder podemos comprimir los datos y con el decoder los podemos descomprimir. En este caso habría que tener en cuenta que sería una técnica de compresión de datos con pérdida de información (JPG también es un formato de compresión con pérdida de compresión). Es decir, con los datos codificados y el AE (pesos de la red neuronal), seríamos capaces de volver a regenerar los datos originales. Otra de las ideas alrededor de los AE es que, si nosotros tenemos un conjunto de datos de la misma naturaleza y los entrenamos con nuestro AE, somos capaces de construir una red neuronal (pesos en la red neuronal) que es capaz de reproducir esos datos a través del AE. Que ocurre si nosotros metemos un dato que no era de la misma naturaleza que los que entrenaron el AE, lo que tendremos entonces es que al recrear los datos originales no va a ser posible que se parezca a los datos de entrada. De forma que el error que vamos a tener va a ser mucho mayor por no ser datos de la misma naturaleza. Esto nos puede llevar a construir un AE que permita detectar anomalías, es decir, que seamos capaces de detectar cuando un dato es una anomalía porque realmente el AE no consigue tener un error lo bastante pequeño. Según lo visto de forma intuitiva vamos a tener el encoder que será el encargado de codificar los datos de entrada y luego tendremos el decoder que será el encargado de realizar la decodificación y conseguir acercarnos al dato original . Es decir intentamos conseguir . Si suponemos un Simple Autoencoder en el que tenemos una única capa oculta, con una función de activación intermedia y una función de activación de salida y los parámetros y represetan los parámetros de la red neuronal en cada capa, tendríamos la siguiente expresión: [190] [191]

Así tendremos que donde será la reconstrucción de Una vez tenemos la idea intuitiva de para qué nos puede ayudar un AE, recopilamos algunos de los principales usos sobre los que actualmente se está trabajando. Más adelante ,comentaremos algunos de ellos con más detalle. Principales Usos de los Autoencoders A continuación, veamos la explicación de cuales son algunos de los principales usos de los autoencoders: Reducción de dimensiones / Compresión de datos En la idea intuitiva de los AE ya hemos visto claro que se pueden usar para la reducción de dimensiones de los datos de entrada. Si estamos ante unos datos de entrada de tipo estructurado estamos en un caso de reducción de dimensiones clásico, en el que queremos disminuir el número de variables con

las que trabajar. Muchas veces este tipo de trabajo se hace mediante el PCA (Análisis de Componente Principales, por sus siglas en inglés), sabiendo que lo que se realiza es una transformación lineal de los datos, ya que conseguimos unas nuevas variables que son una combinación lineal de las mismas. En el caso de los AE conseguimos mediante las funciones de activación no lineales (simgmoide, ReLu, tanh, etc) combinaciones no lineales de las variables originales para reducir las dimensiones. También existen versiones de PCA no lineales llamadas Kernel PCA que mediante las técnicas de kernel son capaces de construir relaciones no lineales. En esta línea estamos viendo que cuando el encoder ha actuado, tenemos unos nuevos datos más reducidos y que somos capaces de practicamente volver a reproducir teniendo el decoder. Podríamos pensar en este tipo de técnica para simplemente comprimir información. Hay que tener en cuenta que este tipo de técnicas no se pueden aplicar a cualquier dato que queramos comprimir, ya que debemos haber entrenado al AE con unos datos de entrenamiento que ha sido capaz de obtener ciertos patrones de ellos, y por eso es capaz luego de reproducirlos. Búsqueda de imágenes Cuando pensamos en un buscador de imágenes nos podemos hacer a la idea que el buscar al igual que con el texto nos va a mostrar entradas que seán imágenes parecidas a la que estamos buscando. Si construimos un autoencoder, el encoder nos va a dar unas variables con información para poder recrear de nuevo la imagen. Lo que parece claro es que si hay muy poca distancia entre estas variables y otras la reconstrucción de la imagen será muy parecida. Así nosotros podemos entrenar el AE con nuestro conjunto de imágenes, una vez tenemos el AE pasamos el encoder a todas las imágenes y las tenemos todas en ese nuevo espacio de variables. Cuando queremos buscar una imagen, le pasamos el autoencoder, y ya buscamos las más cercanas a nuestra imagen en el espacio de variables generado por el encoder. Detección de Anomalías Cuando estamos ante un problema de clasificación y tenemos un conjunto de datos que está muy desbalanceado, es decir, tenemos una clase mayoritaria que es mucho más grande que la minoritaria (posiblemente del orden de más del 95%), muchas veces es complicado conseguir un conjunto de datos balanceado que sea realmente bueno para hacer las predicciones. Cuando estamos en estos entornos tan desbalanceados muchas veces se dice que estamos ante un sistema para detectar anomalías. Un AE nos puede ayudar a detectar estas anomalías de la siguiente forma: • Tomamos todos los datos de entrenamiento de la clase mayoritaria (o normales) y construimos un AE para ser capaces de reproducirlos. Al ser todos estos datos de la misma naturaleza conseguiremos entrenar el AE con un error muy pequeño. • Ahora tomamos los datos de la clase minoritaria (o a nomalías) y los pasamos a través del AE obteniendo unos errores de reconstrucción. • Definimos el umbral de error que nos separará los datos normales de las anomalías, ya que el AE sólo está entrenado con los normales y conseguirá un error más alto con las anomalías al reconstruirlas. • Cogemos los datos de test y los vamos pasando por el AE, si el error es menor del umbral, entonces será de la clase mayoritaria. Si el error es mayor que el umbral, entonces estaremos ante una anomalía. Eliminación de ruido Otra de las formas de uso de los autoencoders en tratamiento de imágenes es para eliminar ruido de las mismas, es decir poder quitar manchas de las imágenes. La forma de hacer esto es la siguiente: • Partimos de un conjunto de datos de entrenamiento (imágenes) a las que le metemos ruido, por ejemplo, modificando los valores de cada pixel usando una distribución normal, de forma que obtenemos unos datos de entrenamiento con ruido. • Construimos el AE de forma que los datos de entrada son los que tienen ruido, pero los de salida vamos a forzar que sean los originales. De forma que intentamos que aprendan a reconstruirse como los que no tienen ruido. • Una vez que tenemos el AE y le pasamos datos de test con ruido, seremos capaces de reconstruirlos sin el ruido. Modelos generativos Cuando hablamos de modelos generativos, nos referimos a AE que son capaces de generar cosas

nuevas a las que existían. De forma que mediante técnicas como los Variational Autoencoders, los Adversarial Autoencoders seremos capaces de generar nuevas imágenes que no teníamos inicialmente. Es decir, podríamos pensar en poder tener un AE que sea capaz de reconstruir imágenes de caras, pero que además con toda la información aprendida fuera capaz de generar nuevas caras que realmente no existen. Diseño del modelo de AE Transformación de datos Cuando se trabaja con redes neuronales y en particular con AEs, necesitamos representar los valores de las variables de entrada en forma numérica. En una red neuronal todos los datos son siempre numéricos. Esto significa que todas aquellas variables que sean categóricas necesitamos convertirlas en numéricas. Además es muy conveniente normalizar los datos para poder trabajar con valores entre 0 y 1, que van a ayudar a que sea más fácil que se pueda converger a la solución. Como ya sabemos normalmente nos encontramos que en una red neuronal las variables de salida son: • un número (regresión) • una serie de números (regresión múltiple) • un dato binario (clasificación binaria) • un número que representa una categoría (clasifiación múltiple) En el caso de los AE puede que tengamos una gran parte de las veces valores de series de números, ya que necesitamos volver a representar los datos de entrada. Esto significa que tendremos que conseguir en la capa de salida esos datos numéricos que teníamos inicialmente, como si se tuviera una regresión múltiple. Arquitectura de red Como ya se ha comentado en las redes neuronales, algunos de los hiperparámetros más importantes en un AE son los relacionados con la arquitectura de la red neuronal. Para la construcción de un AE vamos a elegir una topología simétrica del encoder y el decoder. Durante el diseño del AE necesitaremos ir probando y adaptando todos estos hiperparámetros de la ANN para conseguir que sea lo más eficiente posible: • Número de capas ocultas y neuronas en cada una • Función de coste y pérdida • Optimizador • Función de activación en capas ocultas • Función de activación en salida Número de capas ocultas y neuronas en cada una La selección del número de capas ocultas y la cantidad de neuronas en cada una va a ser un procedimiento de prueba y error en el que se pueden probar muchas combinaciones. Es cierto que en el caso de trabajar con imágenes y CNN ya hay muchas arquitecturas definidas y probadas que consiguen muy buenos resultados. Por otro lado para tipos de datos estructurados será muy dependiente de esos datos, de forma que será necesario realizar diferentes pruebas para conseguir un buen resultado. Función de coste y pérdida En este caso no hay ninguna recomendación especial para las funciones de costes/pérdida y dependerá al igual que en las redes neuronales de la naturaleza de los datos de salida con los que vamos a trabajar. Optimizador Se recomienda usar el optimizador ADAM (Diederik P. Kingma 2017) que es el que mejores resultados ha dado en las pruebas según (Walia 2017), consiguiendo una convergencia más rápida que con el resto de optimizadores.

Función de activación en capas ocultas En un AE las funciones de activación en las capas ocultas van a conseguir establecer las restricciones no lineales al pasar de una capa a la siguiente, normalmente se evita usar la función de activación lineal en las capas intermedias ya que queremos conseguir transformaciones no lineales. Se recomienda usar la función de activación ReLu en las capas ocultas, ya que parece ser que es la que mejores resultados da en la convergencia de la solución y además menor coste computacional tiene a la hora de realizar los cálculos. Función de activación en salida En la capa de salida tenemos que tener en cuenta cual es el tipo de datos final que queremos obtener, que en el caso de un AE es el mismo que el tipo de dato de entrada. Normalmente las funciones de activación que se usarán en la última capa seran: • Lineal con multiples unidades, para regresión de varios datos numéricos • Sigmoid para valores entre 0 y 1 Tipos de Autoencoders Una vez entendido el funcionamiento de los AE, veamos algunos

de los AE que se pueden construir para diversas tareas. • Simple • Multicapa o Profundo • Sparse • Convolucional • Denoising • Variational En la descripción de los tipos de AE vamos usar código en R y en python y el framework keras con el backend Tensorflow. Todo el código se proporciona aparte. Usaremos como dataset a MINIST, que contiene 60.000/10.000 (entrenamiento/validación) imágenes de los números del 0 al 9, escritos a mano. Cada imagen tiene un tamaño de 28x28 = 784 pixels, en escala de grises, con lo que para cada pixel tendremos un valor entre 0 y 255 para definir cuál es su intensidad de gris. Autoencoder Simple Vamos a describir como construir un autoencoder Simple usando una red neuronal densamente conectada en lugar de usar una red neuronal convolucional, para que sea más sencillo comprender el ejemplo. Es decir, vamos a tratar los datos de entrada como si fueran unos datos numéricos que queremos reproducir y no vamos a utilizar ninguna de las técnicas asociadas a las redes convolucionales. Hay que recordar que las redes convolucionales permiten mediante un tratamiento de las imágenes (convolución, pooling, etc) conseguir mejores resultados que si lo hiciéramos directamente con redes densamente conectadas. En este caso tendremos una capa de entrada con 784 neuronas (correspondientes a los pixels de cada imagen), una capa intermedia de 32 neuronas, y una capa de salida de nuevo de las 784 neuronas para poder volver a obtener de nuevo los datos originales. En nuestro ejemplo vamos a tener los siguientes elementos. • 1 capa de entrada (784 datos), 1 capa oculta (32 datos) y una capa de salida (784 datos) • La función de coste/pérdida va a ser la Entropía • Usaremos el optimizador Adam • Como función activación intermedia usaremos ReLu • Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1) Autoencoder Sparse Ya hemos comentado que una forma de conseguir que un autoencoder aprenda estructuras o correlaciones es la reducción del número de neuronas, pero parte de este trabajo también se puede conseguir mediante técnicas de sparsing (escasez). Este tipo de técnicas se usan normalmente en las ANN para evitar el sobreajuste de nuestro modelo, de forma que en cada actualización de los pesos de la red no se tienen en cuenta todas las neuronas de la capa. Es decir, vamos a conseguir que en las capas que decidamos no todas las neuronas van a estar activadas, de esta manera además de ayudar a evitar el sobreajuste, también conseguiremos crear esas correlaciones que ayudan a construir el autoencoder. Existen dos metodos básicos para generar el sparse que son: • Regularización L1 • Regularización L2 Básicamente los dos métodos tratan de hacer que los pesos de las neuronas tengan valores muy pequeños consiguiendo una distribución de pesos más regular. Esto lo consiguen al añadir a la función de pérdida un coste asociado a tener pesos grandes en las neuronas. Este peso se puede construir o bien con la norma L1 (proporcional al valor absoluto) o con la norma L2 (proporcional al cuadrado de los coeficientes de los pesos). Básicamente trabajaremos con un AE Simple, con una red densamente conectada, al que le aplicaremos la regularización en su capa oculta. En nuestro ejemplo vamos a tener los siguientes elementos. - 1 capa de entrada, 1 capaa oculta y una capa de salida - Las capas ocultas tendrán aplicada la Regularización L2 - La función de coste/pérdida va a ser la Entropía - Usaremos el optimizador Adam - Como función activación intermedia usaremos ReLu - Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1) Autoencoder Multicapa o profundo Vamos a pasar ahora a una versión del autoencoder donde habilitamos más capas ocultas y hacemos que el descenso del número de neuronas sea más gradual hasta llegar a nuestro valor deseado, para luego volver a reconstruirlo. En este caso seguimos con redes densamente conectadas y aplicamos varias capas intermedias reduciendo el número de neuronas en cada una hasta llegar a la capa donde acaba el encoder para volver a ir creciendo en las sucesivas capas hasta llegar a la de salida. En nuestro ejemplo vamos a tener los siguientes elementos. - 1 capa de entrada (784 datos), 5 capa ocultas (32

datos intermedia) y una capa de salida (784 datos) - La función de coste/pérdida va a ser la Entropía - Usaremos el optimizador Adam - Como función activación intermedia usaremos ReLu - Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1) Autoencoder Convolucional En nuestro ejemplo al estar trabajando con imágenes podemos pasar a trabajar con Redes Convolucionales (CNN) de forma que en lugar de usar las capas densamente conectadas que hemos usado hasta ahora, vamos a pasar a usar las capacidades de las redes convolucionales. Al trabajar con redes convolucionales necesitaremos trabajar con capas de convolución o pooling para llegar a la capa donde acaba el encoder para volver a ir creciendo aplicando operaciones de convolución y upsampling (contrario al pooling). En nuestro ejemplo vamos a tener los siguientes elementos. - 1 capa de entrada, 5 capas ocultas y una capa de salida - La función de coste/pérdida va a ser la Entropía - Usaremos el optimizador Adam - Como función activación intermedia usaremos ReLu - Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1) Autoencoder Denoising Vamos a usar ahora un autoencoder para hacer limpieza en imagen, es decir, conseguir a partir de una imagen que tiene ruido otra imagen sin ese ruido. Entrenaremos al autoencoder para que limpie "ruido" que hay en la imagen y lo reconstruya sin ello. El ruido lo vamos a generar mediante una distribución normal y modificaremos el valor de los pixels de las imágenes. Usaremos estas imágenes con ruido para que sea capaz de reconstruir la imagen original sin ruido con el AE. Para realizar este proceso lo que haremos será_ • Crear nuevas imágenes con ruido • Entrenar el autoencoder con estas nuevas imágenes • Calcular el error de reconstrucción respecto a las imágenes originales Al estar trabajando con imágenes vamos a partir del Autoencoder de Convolución para poder aplicar el denosing. En nuestro ejemplo vamos a tener los siguientes elementos. - 1 capa de entrada, 5 capas ocultas y una capa de salida - La función de coste/pérdida va a ser la Entropía - Usaremos el optimizador Adam - Como función activación intermedia usaremos ReLu - Como función activación de salida sigmoid (ya que queremos un valor entre 0 y 1) Autoencoder Variational Los Variational Autoencoder son un tipo de modelo que se denomina generativo, ya que va a permitir construir nuevas imágenes que no existían a partir de otras imagenes con las que se ha entrenado a la red neuronal. En realidad, es un autoencoder que durante el entrenamiento se le regulariza para evitar un sobreajuste y asegurar que en el espacio latente (intermedio) tenga buenas propiedades que permitan un buen proceso generativo. El proceso de construir este tipo de AE es muy parecido a los que ya hemos visto, con una pequeña diferencia en el paso entre el proceso de encoder y el posterior decoder. Hasta ahora lo que teníamos era que lo que obteníamos del encoder se lo pasábamos directamente al decoder, en este caso, el resultado del encoder no va a ser realmente un dato, sino una distribución de datos. De forma que al decoder no se le pasa directamente lo que ha salido del encoder, si no otro elemento cogido de la distribución generada. En este caso el proceso sería: • Se codifica la entrada con el encoder no como un dato concreto, sino como una distribución normal (media y desviación) • Se toma una muestra de un punto del espacio latente a partir de la distribución • Se decodifica el punto de muestra con el decoder • Se calcula la función de pérdida con el error de reconstrucción y la parte de regularización • Se usa el backpropagation a través de la red neuronal para ajustar los pesos Figura nº 90: Autoencoder Mixto (Incompleto y Extra dimensionado)

Fuente: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73 Una vez que tenemos entrenado nuestro autoencoder seremos capaces de construir nuevas imágenes partiendo de puntos que estén en la distribución del espacio latente, de forma que esas pequeñas variaciones van a dar lugar a imágenes finales

diferentes.

2.6 Arquiecturas Preentranadas

2.6.1 Detección de objetos

2.6.2 Tratamiento de audios

2.6.3 Tareas sobre textos

2.7 Aprendizaje por Refuerzo

2.7.1 Introducción

Hasta ahora hemos visto como el Deep Learning se usa para el **aprendizaje supervisado** y el **aprendizaje no supervisado**, pero vamos a dar un paso más, en el que veremos como usar Deep Learning en otro tipo de aprendizaje llamado **aprendizaje por refuerzo**.

El **Aprendizaje por Refuerzo** (**RL** por sus siglas en ingles, Reinforcement Learning) trata de conseguir que el sistema aprenda mediante recompensa/castigo, en función de si los pasos que da son buenos o malos. De esta manera, cuanta mayor recompensa se tenga es que nuestro sistema se ha acercado a la solución buena. Se trata de aprender mediante la interacción y la retroalimentación de lo que ocurra.

Partiremos de dos elementos clave **agente** (es el que aprende y toma decisiones), y el **entorno** (donde el agente aprende y decide que acciones tomar). Tendremos que el agente podrá realizar **acciones** que normalmente provocarán un cambio de **estado** y a la vez se tendrá una **recompensa** (positiva o negativa) en función de la acción tomada en el entorno en ese momento.

Es decir, nos encontraremos un agente que realizará una acción a_t en el tiempo t, esta acción afectará al entorno que estará en un estado S_t y mediante esta acción cambiará a un estado S_{t+1} y además dará una recompensa r_{t+1} en función de los malo o bueno que haya sido este paso.

El agente volverá a examinar el nuevo estado del entorno S_{t+1} y la nueva recompensa recibida r_{t+1} y volverá a tomar la decisión de realizar una nueva acción a_{t+1}.

2.7.2 Historia del Aprendizaje por Refuerzo

2.7.3 Formalismo Matemático

El formalismo matemático para el Aprendizaje por Refuerzo está basado en los **Procesos de Decisión de Markov** (MDP por sus siglas en ingles). (CS229 Lecture notes).

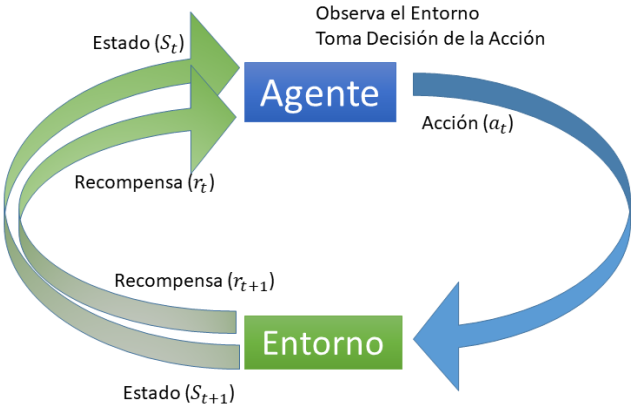


Imagen 2.14: Esquema Aprendizaje por Refuerzo - Fuente: Propia

2.7.3.1 Propiedad de Markov

Si tenemos una secuencia de estados $s_1, s_2, ..., s_t$ y tenemos la probabilidad de pasar a otro estado s_{t+1}, diremos que se cumple la **Propiedad de Markov** si el **futuro** es independiente del **pasado** y sólo se ve afectado por el **presente**, es decir:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_t, s_{t-1}, ...S_2, S_1]$$

Tendremos una **Matriz de Probabilidades de Transición** a una matriz con las probabilidades de todos los posibles cambios de estado que se puedan producir,

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$$

2.7.3.2 Proceso de Markov

Así llamaremos **Proceso de Markov** a un proceso aleatorio sin memmoria, es decir, una secuencia de estados $S_1, S_2, ...$ con la propiedad de Markov.

Un Proceso de Markov está formado por una dupla $< \mathcal{S}, \mathcal{P} >$,:

- $\mathcal{S}$ conjunto finito de Estados
- $\mathcal{P}$ matriz de probabilidades de transición

2.7.3.3 Proceso de Recompensa de Markov

LLamaremos **Proceso de Recompensa de Markov** (**MRP,** por sus siglas en ingles) a una cuádrupla $< \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma >$, formada por:

- $\mathcal{S}$ conjunto finito de Estados

- $\mathcal{P}$ matriz de probabilidades de transición
- $\mathcal{R}$ Función de recompensa definida como: $\mathcal{R}_s = E[R_{t+1}|S_t = s]$, donde R_{t+1}es la recompensa obtenida de pasar al estado S_{t+1} desde el estado S_t
- γ Factor de descuento, con $\gamma \in [0, 1]$

En este contexto llamaremos **Saldo (G_t)** a la suma de todas las recompensas conseguidas a partir del estado s_t con el factor de descuento aplicado.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

El hecho de usar γ (**factor descuento**), nos permite dar grandes recompensas lo antes posible, y no dar tanto valor a futuras recompensas lejanas. También puede haber otras interpretaciones por ejemplo a nivel económico, si la recompensa está basado en un dato monetario real, tendría sentido que el dinero a futuro tendría menos valor. También nos permite asegurar que este valor de G_t es finito ya que produce que la serie sea convergente.

Cuando los valores del factor descuento se acercan a **0** podríamos decir que nos fijamos sólo en los valores más cercanos de la recompensa. En cambio cuando los valores se acercan a **1** entonces les daremos más peso a los valores más lejanos de la recompensa.

Una vez definido el Saldo podemos definir la **Función Valor de Estado** como la función que nos da el **Saldo Esperado** comenzando por el estado s. Es decir:

$$V(s) = \mathbb{E}[G_t|S_t = s]$$

Esta función nos dice cómo de bueno es partir de este estado y continuar.

2.7.3.4 Proceso de Decisión de Markov

Un **Proceso de Decisión de Markov (MDP por sus siglas en inglés)** es un tupla $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ donde:

- $\mathcal{S}$ es el conjunto de posibles **estados**.
- $\mathcal{A}$ es el conjunto de posibles **acciones**.
- $\mathcal{P}$ son las **probabilidades de transición** de un estado a otro en función de la acción realizada. Por cada estado y acción hay una distribución de probabilidad para pasar a otro estado.

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$

- γ es el conocido como **factor de descuento** y tendrá un valor entre $[0, 1)]$ y nos proporciona cuanto descontamos en las recompensas a futuro.
- $\mathcal{R}$ es la **Función de recompensa** definida como: $\mathcal{R}^a_s = E[R\_t + 1|S_t = s, A_t = a]$, donde $R_t + 1$es la recompensa obtenida de pasar al estado S_{t+1} desde el estado

Además tenemos que este **proceso estocástico** cumple la **propiedad de Markov** que dice que el futuro es independiente del pasado dado el presente. En términos de nuestro problema, podría decir que pasar de un estado s_t al siguiente s_{t+1} sólo depende de s_t y no de los anteriores estados

$$\mathbb{P}(s_{t+1}|s_t) = \mathbb{P}(s_{t+1}|s_1, s_2, ..., s_t)$$

Veamos cual es la **dinámica** de un MDP:

- Empezamos con un estado $s_0 \in \mathcal{S}$
- Elegimos una acción $a_0 \in \mathcal{A}$ (la política será la que la elija)
- Obtenemos una recompensa $R_1 = R(s_0) = R(s_0, a_0)$
- Elegimos una acción $a_1 \in \mathcal{A}$(la política será la que la elija)
- Se transiciona aleatoriamente a un estado s_1 en un función del valor de $P^{a_1}_{s_0 s_1}$
- Obtenemos una recompensa $R_2 = R(s_1) = R(s_1, a_1)$
- Se transiciona a aleatoriamente a un estado $s_1 2$ en un función del valor de $P^{a_2}_{s_1 s_2}$
- ...
- Repetimos de forma iterativa este proceso

La meta en RL es elegir las **acciones** adecuadas en el tiempo para **maximizar**:

$$\mathbb{E}[G_t] = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + ...]$$

Que es conocido como la **hipotesis de la recompensa**.

Vamos a introducir el término de **política** como una función $\pi : \mathcal{S} \to \mathcal{A}$ que mapea los estados a las acciones. Es decir, es la que decide que **acción** hay que **ejecutar** en función de **cual** es el **estado** en el que estamos. Una política podría ser determinística o estocástica.

$$a = \pi(s) a = \pi(a|s) = \mathbb{P}[A = a|S = s]$$

Una política define cual va a ser el comportamiento de un **agente**. En un MDP las políticas dependen del estado actual, y no de la historia de los estados pasados.

Diremos que estamos **ejecutando una política** π si cuando estamos en un estado s aplicamos la acción $a = \pi(s)$

Definiremos:

$$\mathcal{P}^{\pi}_{s,s`} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^{a}_{s,s`} \mathcal{R}^{\pi}_{s} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^{a}_{s}$$

También definiremos la **Función Valor de Estado** para una **política** π a la función que nos predice la recompensa a futuro (el **saldo esperado**):

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t|S_t = s] = \mathbb{E}_{\pi}[R(s_t) + \gamma R(s_{t+1}) + \gamma^2 R(s_{t+2}) + \gamma^3 R(s_{t+3}) + ...|s_t = s]$$

Es decir, la esperanza de la suma de las recompensas con factor descuento suponiendo el comienzo en $s_t = s$ y tomando las acciones bajo la política π. Nos permite decir cómo de buenos o malos son los estados.

Añadiremos el concepto de la **Función Valor de Acción,** también llamada **Función de Calidad** (por eso se usa la Q (Quality)**,** para una **política** π a la función que nos

predice la recompensa a futuro (el saldo esperado), suponiendo que se se **parte de una acción** a.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|S_t = s, A_t = a] = \mathbb{E}_{\pi}[R(s_t)+\gamma R(s_{t+1})+\gamma^2 R(s_{t+2})+\gamma^3 R(s_{t+3})+...|s_t = s, A_t = a]$$

La función de **Valor de Estado** puede ser descompuesta en la **recompensa inmediata** y el resto de la recompensa:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_{t+1} + \gamma V^{\pi}(S_{t+1)}|S_t = s]$$

y del mismo modo se puede descomponer la función **Valor de Acción**:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_{t+1} + \gamma Q^{\pi}(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

Luego tenemos

$$V^{\pi}(s) = \sum_{a\in A} \pi(a|s) Q^{\pi}(s,a)$$

y

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a V^{\pi}(s')$$

Llegando a

$$V^{\pi}(s) = \sum_{a\in A} \pi(a|s)(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a V^{\pi}(s'))$$

y

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \sum_{a\in \mathcal{A}} \pi(a|s) Q^{\pi}(s',a)$$

Dada una política π su **función valor de estado** asociada $V^{\pi}(s)$ cumple la **Ecuación de Bellman**:

$$V^{\pi}(s) = R_s + +\gamma \sum_{s'\in S} P_{s,\pi(s)}(s') V^{\pi}(s')$$

Lo que nos dice que la **función valor** está separada en **dos términos**:

- La recompensa inmediata $R(s)$
- La suma de recompensas a futuro con el factor de descuento.

Igualmente su **función valor de acción asociada** $Q^{\pi}(s,a)$cumple la **Ecuación de Bellman**:

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \sum_{a\in \mathcal{A}} \pi(a|s) Q^{\pi}(s',a)$$

Las **Ecuaciones de Bellman** permiten garantizar una **solución óptima** del problema de forma que dada una **política óptima** (π^*), además se cumple:

$$V^{\pi^*}(s) = V^*(s) = max_{\pi} V^{\pi}(s) Q^{\pi^*}(s,a) = Q^*(s,a) = max_{\pi} Q^{\pi}(s,a)$$

Es decir, que las funciones de valor de estado y de acción óptimas son las mismas que se general con la **política óptima**.

Como la meta del RL es encontrar una **política óptima** π^* la cual maximize el valor del **saldo esperado total (desde el inicio)** $G_0 = \sum_{t=0}^{\infty}$, es decir, podríamos definir la política óptima como:

$$\pi^*(a|s) = \begin{cases} 1 \text{ si } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^*(s,a) \\ 0 \text{ si cualqier otro caso} \end{cases} \tag{2.1}$$

Luego si conocemos $Q^*(s,a)$ inmediatamente tenemos una **política óptima.**

2.7.3.5 Resolución de las Ecuaciones de Bellman

Las ecuaciones de Bellman pueden ser usadas para resolver de forma eficiente V^π, especialmente en un **MDP** de un número finito de estado, escribiendo una ecuación $V^\pi(s)$ por cada estado.

La mayoría de los algoritmos de RL usan las Ecuaciones de Bellman para resolver el problema. La forma básica de resolverlo es usando **progración dinámica** (PD por sus siglas en inglés), aunque nos encontramos con muchos problemas para resolverla cuando el número de acciones/estados aumenta. También se usan otras técnicas como los **métodos de montecarlo** (MMC, por sus siglas en inglés) o los métodos de **diferencia temporal** (TD, por sus siglas en ingles).

Pasemos a ver una clasificación de los tipos de algoritmos para resolver los problemas de RL.

2.7.4 Taxonomía de Algoritmos

Desde OpenAI (https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below) obtenemos la siguiente taxonomía de algoritmos de RL que nos servirá como guía para entender como clasificar los algoritmos:

La primera gran separación se hace sobre si los algoritmos siguen un modelo definido (model-baed) o no (modelo-free).

Model-free

Por otro lado los **model-free** usan la experiencia para aprender o una o ambas de dos cantidades más simples (valores estado/acción o políticas).

Las aproximaciones de estos algorimtos son de tres tipos:

- **Policy Optimization**

El agente aprende directamente la función política que mapea el estado a una acción. Nos podemos encontrar con dos tipos de políticas, las **políticas deterministicas** (no hay incertidumbre en el mapeo) y las **políticas estocásticas** (tenemos una distribución de probabilidad en las acciones)**.** En este último caso diremos que tenemos un Proceso de Decisión de Markov Parcialmente Observable (POMDP, por sus siglas en ingles).

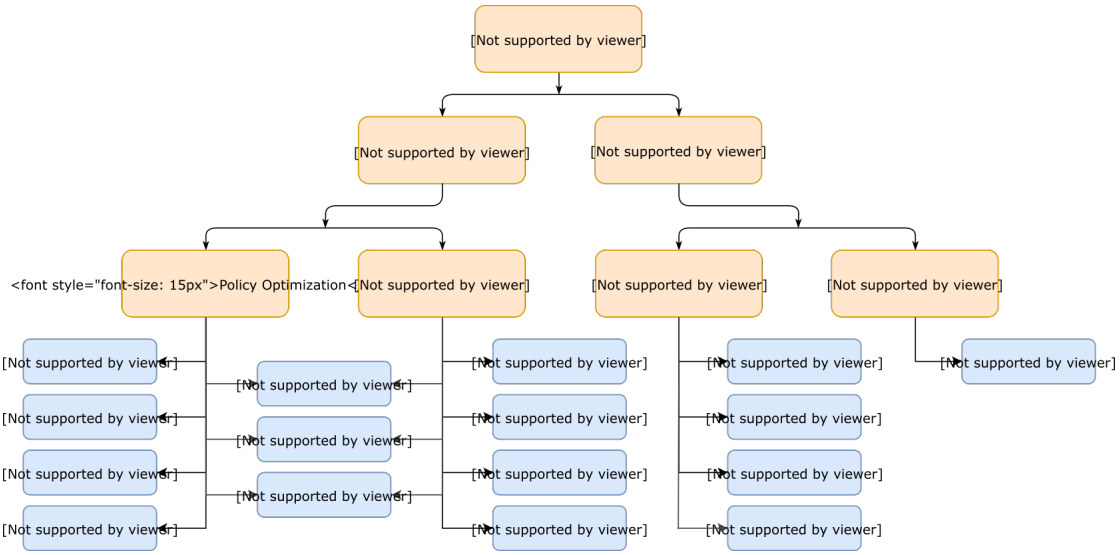


Imagen 2.15: Esquema Aprendizaje por Refuerzo - Fuente: Propia

- **Q-Learning**

En este caso el agente aprende una función valor de acción $Q(s, a)$ que nos dirá cómo de bueno es tomar una acción dependiendo del estado.

- **Híbridos**

Estos métodos combinan la fortaleza de los dos métodos anteriores, aprendiendo tanto la función política como la función valor de acción.

Model-based

Los algoritmos **model-based** usan la experiencia para construir un modelo interno de transiciones y resultados inmediatos en el entorno. Las acciones son elegidas mediante búsqueda o planificación en este modelo construido.

Las aproximaciones de estos algorimtos son de dos tipos:

- Aprender el Modelo

Para aprender el modelo se ejecuta una política base,

- Aprender dado el Modelo

Nos centraremos en los algoritmos de tipo **Model-Free** que son los más utilizados ya que no requieren del modelo. Si se quieren profundizar en los diferentes algoritmos, se puede consultar las documentación en: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#links-to-algorithms-in-taxonomy.

Vamos a ver 2 de los algoritmos de tipo **Model-free** que nos van a permitir el ver el paso de un algoritmo sin **Deep Learning** y otro en el que se aplica **Deep Learning** para obtener el objetivo final de tener un **agente** capaz de aprender por sí solo a realizar las tareas específicas que se tengan que realizar.

2.7.5 Q-Learning (value)

Q-Learning es un método basado en valor y que usa el **sistema TD** (actualización su función valor en cada paso) para el entrenamiento y su función de valor de estado.

En nombre de **Q** viende de **Quality** (calidad), por que nos da la calidad de la acción en un determinado estado. Lo que tenemos es que vamos a tener una **función de valor de acción (Q-función)** que nos da un valor numérico de cómo de buena es a partir de un estado **s** y una acción **a**.

En este caso tenemos que internamente nuestra **Q-función** ($Q(s, a)$) es una **Q-tabla**, de forma que cada fila corresponde a un estado, y cada columna a una de las posibles acciones.

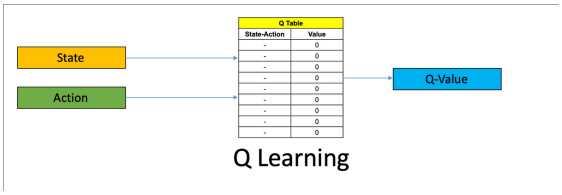


Imagen 2.16: Q-Learning - Fuente: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Es decir, esta tabla va a contener la información de **recompensa total esperada** para cada valor de estado y acción. Cuando nosotros realizamos el **entrenamiento** de la Q-función, nosotros conseguimos una función que **optimice** esta **Q-tabla**.

Si nosotros tenemos una Q-función óptima ($Q^*(s, a)$), entonces podremos obtener la **política óptima** a partir de ella:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^*(s, a)$$

Veamos cuales serían los pasos que deberíamos dar:

Inicializamos nuestra **Q-Tabla** con valores a 0. Conforme avanece nuestro entrenamiento estos valores irán cambiando en función de los datos que se obtengan al **porbar** a realizar **acciones** y obtener las **recompensas** correspondientes.

El siguiente elemento que necesitamos es una **política de entrenamiento** (función que nos permita elegir que acción tomar en función del estado en el que estemos), en este caso nuestra política estará basada en los valores de la **Q-tabla**, es lo que llamaremos **explotación** (explotamos la información que tenemos cogiendo la acción con mejor valor Q) o elegiremos otra acción, es lo que llamaremos **exploración** (exploramos nuevos caminos cogiendo una acción de forma aleatoria).

Esto es lo que se llama una política ϵ-greedy, ya que se usa un parámetro ϵ, valor entre 0 y 1, que nos permite decidir si elegimos **explorar** o si queremos **explotar** los datos que ya tenemos.

XXXXX Imagen del gráfico epsilon (epsilon respecto al número de epsisodios)

Tendremos que:

- con probabilidad 1-ϵ nosotros haremos **explotación** y

- con probabilidad ϵ nosotros haremos **exploración**.

Es decir, inicialmente le damos valor 1 a ϵ de forma que empezaremos haciendo **exploración** e iremos bajando este valor de epsilon conforme avance el entrenamiento para que cada vez usemos más la **explotación**.

La idea base es que al principio del entrenamiento, lo prioritario es **explorar**, es decir, seleccionar una acción al azar y obtener su recompensa, ya que nuestra **Q-Tabla** está inicializada a 0. Conforme avance el entrenamiento nos tendremos que ir fiando más de los datos que ya tenemos y tendrá que primar la **explotación** de nuestros datos de la **Q-Tabla**. Para hacer ésto de una forma efectiva, usaremos un parámetro **decay_epsion** que conforme avancemos en entrenamiento se encargará de ir reduciendo el valor de ϵ para conseguir este efecto.

Una vez que tenemos nuestros elementos base, pasaremos al **entrenamiento**, de forma que para todos los **episodios** (iteraciones de partidas) que definamos haremos lo siguiente:

- Partimos de un **estado inicial**, y obtenemos una **acción** a partir de nuestra **política de entrenamiento**
- Actualizmos ϵ con el nuevo valor en este episodio
- Iteramos para un número máximo de pasos dentro de este episodio
- Obtenemos el nuevo estado, así como la recompensa obtenida
- Actulizamos el valor de la **Q-Tabla** correspondiente según la fórmula basada en los métodos de **TD** (Diferencias temporales)

  $Q(s,a) = Q(s,a) + \alpha(R(s,a) + \gamma argmax_a Q(s',a) - Q(s,a))$ donde s es el estado actual y s' es el nuevo

- Verificamos si se ha llegado al final del juego para salir de este espisodio si es el caso
- Cambiamos el **estado** como el **nuevo estado**

Una vez acabemos nuestro entrenamiento, obtendremos nuestra **política óptima** como:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q^*(s,a)$$

Pseudo código Q-Learning

Algoritmo 1: Q-Learning

Input: Politica entrenamiento, *numero_episodios*, *numero_pasos*, ϵ, *decay_epsilon*
Output: Valores de la función Q optimizada

$Q \leftarrow 0$;
for $i \leftarrow 1$ ***to*** *numero_episodios* **do**
  $\epsilon \leftarrow$ *nuevo_epsilon*(*calculadocondecay_epsilon*);
  s_0;
  $t \leftarrow 0$;
  for $t \leftarrow 1$ ***to*** *numero_pasos & Not finished* **do**
    Elegimos una acción A_t con política entrenamiento;
    Aplicamos la acción A_t obteniendo R_{t+1}, S_{t+1};
    Actualizamos $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma argmax_a Q(S_{t+1}, a) - Q(S_t, A_t))$;
return Q;

Imagen 2.17: Algoritmo Q-Learning - Fuente: Propia

2.7.6 DQN (Deep Q-Learning)

https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Hemos visto el algoritmo de **Q-Learning** en el que usábamos una **Q-Tabla**, es decir una tabla donde guardábamos todos los valores de la función $Q(s, a)$ y que entrenando el agente, éramos capaces de conseguir aproximar a la función **Q óptima**, con lo cual teníamos una **Política Óptima**.

Este tipo de algoritmos son válidos cuando nos encontramos con un número "limitado" de estados y acciones, de forma que la tabla es relativamente manejable y somos capaces de entrenarla. Si nos encontramos ante un problema en el que tenemos miles o cientos de miles de estados no va a ser efectivo construir una tabla y entrenarla para todas las posibles combinaciones **etado-acción**. Para abordar este tipo de problemas, la mejor solución es buscar un **aproximador** de la función $Q(s, a)$, que nos permita obtener la mejor solución sin necesidad de entrenar todas las posibles combinaciones.

Para realizar este trabajo una de las posibles opciones es usar **redes neuronales** como función aproximadora y que nos abrirá la posibilidad de trabajar con problemas en los que existan grandes cantidades de estados/acciones.

Fue el equipo de **Deepmind** en 2013 (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf), en su artítulo **"Human-level control through deep reinforcementlearning"**, los primeros que decidieron atacar los problemas de alta dimensionalidad de **estados/acciones** mediante el uso de **Redes Neuronales Profundas**. La forma de probar su código fue mediante la implementación de **agentes** que fueran capaces de aprender a jugar a los clásicos **juegos de Atari 2600**. De forma que el agente, recibiendo la información de entrada de los pixels que hay en cada momento en pantalla y el marcador del juego, eran capaces de sobrepasar el rendimiento de algoritmos actuales que hacían ese trabajo. En estte caso usaron la misma red neuronal, con la misma arquitectura e hiperparámetros para los 49 juegos con los que se probaron.

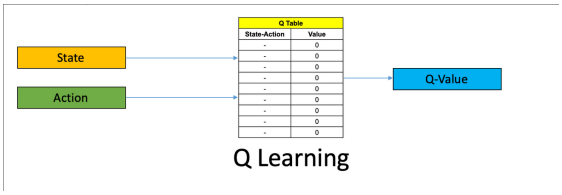


Imagen 2.18: Deep Q-Learning - Fuente: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Con nuestro algoritmo de **Q-Learning** teníamos una función $Q(s, a)$ que implementábamos con un tabla y nos daba para cada **estado** y cada **acción** cual era el valor de **Q** (Quality) de la recompensa esperada. Ahora, con **Deep Q-Learning** nos encontramos que vamos a tener una red neuronal que será la encargada de para cada **estado** obtener el valor de **Q** para cada posible **acción**.

DQN (Deep Q-Network) Arquitectura

Para poder implementar nuestro trabajo con redes neuronales nos vamos a encontrar con el problema de entrenar la red neuronal (obtener los pesos) que permitan alcancar

nuestra función **Q-Óptima** que nos daría la **Política Òptima** que es lo que realmente buscamos.

Básicamente para realizar el trabajo usaremos 2 redes neuronales que tendrán la misma arquitectura de forma que el entrenamiento sea estable.

- **DQN** que será la red de predicción, y que será la que entrenaremos para minimizar el valor del error $(R + \gamma argmax_{a'} Q(s', a', w') - Q(s, a, w))^2$
- **DQN_Target** que será la red que calculará $R + \gamma argmax'_a Q(s', a', w')$

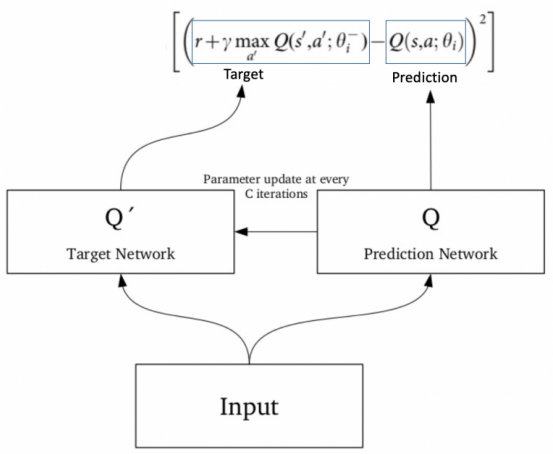


Imagen 2.19: Arquitectura Redes DQN - Fuente: https://www.analyticsvidhya.com/blog/2019/04/introduc deep-q-learning-python/

Experience Replay

El mecanismo del **Experience Replay** nos va a permitir entrenar nuestra red **DQN** con minibatchs que vamos a extraer de forma aleatoria de la memoria en la que vamos a ir guardando los resultados que vamos obteniendo **<s,a,r,s'>**.

Ésto nos va a permitir por un lado **entrenar** nuestra **red de predicción** y además va a servirnos para evitar **correlaciones** de secuencias consecutivas que pudieran producir un sesgo en nuestros resultados. De esta manera, al elegir al azar los elementos que vamos a usar para entrenar la red, no tendrán ninguna relación con los datos consecutivos que se van produciendo en los pasos de los episodios.

Algoritmo Deep Q-Learning

- Obtenemos los datos de entrada, que es el estado.
- Seleccionamos la acción usando nuestra política de entrenamiento epsilon-greedy
- Ejecutamos la acción y obtenemos el siguiente estado así como la recompensa obtenida
- Almacenamos en memoria <s,a,r,s'>
- Si tenemos bastantes elementos en la memoria
  - Hacemos un minibatch aleatorio y enteramos la red siendo $R + \gamma argmax_{a'} Q(s', a', w')$ el **target de la red** y $Q(s, a, w)$ el valor predicho.
  - La función de pérdida será la de Diferencia de Cuadrados $L = (R + \gamma argmax'_a Q(s', a', w') - Q(s, a, w))^2$

- Después de cada C iteraciones, copiaremos los pesos de la red DQN a la DQN_Target
- Repetiremos estos pasos durante M episodios

Pseudo-código Deep Q-Learning

Algoritmo 2: Deep Q-Learning (DQN)

Input: Politica entrenamiento epsilon-greedy, *numero_episodios*, *numero_pasos*, ϵ, *decay_epsilon*, *size_replay_memory*, *size_minibatch*, C

Output: Valores de la función Q optimizada

$DQNetwork \leftarrow Initializer$;
$DQNetwork\_target \leftarrow DQNetwork$;
Replay_Memory Initializer con *size_replay_memory* elementos;
$C \leftarrow 0$;
for $i \leftarrow 1$ ***to*** *numero_episodios* **do**
  $\epsilon \leftarrow nuevo\_epsilon$ (calculado con *decay_epsilon*);
  s_0;
  $t \leftarrow 0$;
  for $t \leftarrow 1$ ***to*** *numero_pasos & Not finished* **do**
    Elegimos una acción A_t con política entrenamiento epsilon-greedy;
    Aplicamos la acción A_t obteniendo R_{t+1}, S_{t+1};
    Almacenamos $< S_t, A_t, R_{t+1}, S_{t+1} >$ en *Replay_Memory*;
    Obtenemos de forma aleatoria un minibatch desde *Replay_Memory*;
    for $t \leftarrow 1$ ***to*** *size_minibatch & Not finished* **do**
      Entrenamos la red *DQNetwork* con función de pérdida $L = (R + \gamma argmax'_a Q(s', a', w') - Q(s, a, w))^2$;
      Donde $R + \gamma argmax'_a Q(s', a', w')$ lo calculamos con $DQNerwork\_Target$ y $Q(s, a, w)$ con *DQNetwork*
    if $i \%C = 0$ **then**
      Actualizamos la red *DQNetwork_Target* con *DQNewtork*
      $DQNetwork\_target \leftarrow DQNetwork$;
return *DQNetwork*;

Imagen 2.20: Algoritmo Deep Q-Learning - Fuente: Propia

Variantes de Deep Q-Learning

- Double Deep Q Network (DDQN) – 2015
- Deep Recurrent Q Network (DRQN) – 2015
- Dueling Q Network – 2015
- Persistent Advantage Learning (PAL) – 2015
- Bootstrapped Deep Q Network – 2016
- Normalized Advantage Functions (NAF) = Continuous DQN – 2016
- N-Step Q Learning – 2016
- Noisy Deep Q Network (NoisyNet DQN) – 2017
- Deep Q Learning for Demonstration (DqfD) – 2017
- Categorical Deep Q Network = Distributed Deep Q Network = C51 – 2017
  - Rainbow – 2017
- Quantile Regression Deep Q Network (QR-DQN) – 2017
- Implicit Quantile Network – 2018

2.7.7 Listado Algoritmos

1. Model-Free

Value-based

Q-learning = SARSA max – 1992

State Action Reward State-Action (SARSA)– 1994

Deep Q Network (DQN) – 2013

Double Deep Q Network (DDQN) – 2015

Deep Recurrent Q Network (DRQN) – 2015

Dueling Q Network – 2015

Persistent Advantage Learning (PAL) – 2015

Bootstrapped Deep Q Network – 2016

Normalized Advantage Functions (NAF) = Continuous DQN – 2016

N-Step Q Learning – 2016

Noisy Deep Q Network (NoisyNet DQN) – 2017

Deep Q Learning for Demonstration (DqfD) – 2017

Categorical Deep Q Network = Distributed Deep Q Network = C51 – 2017

- Rainbow – 2017

Quantile Regression Deep Q Network (QR-DQN) – 2017

Implicit Quantile Network– 2018

Mixed Monte Carlo (MMC) – 2017

Neural Episodic Control (NEC) – 2017

Policy-based

Cross-Entropy Method (CEM)– 1999

Policy Gradient

- REINFORCE = Vanilla Policy Gradient(VPG)- 1992
- Policy gradient softmax
- Natural Policy Gradient (Optimisation) (NPG) / (NPO) – 2002
- Truncated Natural Policy Gradient (TNPG) – 2016

Actor-Critic

Advantage Actor Critic (A2C) – 2016

Asynchronous Advantage Actor-Critic (A3C) – 2016

Generalized Advantage Estimation (GAE) – 2015

Trust Region Policy Optimization (TRPO) – 2015

Deterministic Policy Gradient (DPG) – 2014

Deep Deterministic Policy Gradients (DDPG) – 2015

- Distributed Distributional Deterministic Policy Gradients (D4PG) – 2018

- Twin Delayed Deep Deterministic Policy Gradient (TD3) – 2018

Actor-Critic with Experience Replay (ACER) – 2016

Actor Critic using Kronecker-Factored Trust Region (ACKTR) – 2017

Proximal Policy Optimization (PPO) – 2017

- Distributed PPO (DPPO) – 2017
- Clipped PPO (CPPO) – 2017
- Decentralized Distributed PPO (DD-PPO)– 2019

Soft Actor-Critic (SAC) – 2018

General Agents

- Covariance Matrix Adaptation Evolution Strategy (CMA-ES)– 1996
- Episodic Reward-Weighted Regression (ERWR) – 2009
- Relative Entropy Policy Search (REPS)– 2010
- Direct Future Prediction (DFP) – 2016

Imitation Learning Agents

Behavioral Cloning (BC)

Dataset Aggregation (Dagger) (i.e. query the expert) – 2011

Adversarial Reinforcement Learning

- Generative Adversarial Imitation Learning (GAIL) – 2016
- Adverserial Inverse Reinforcement Learning (AIRL)– 2017

Conditional Imitation Learning – 2017

Soft Q-Imitation Learning (SQIL) – 2019

Hierarchical Reinforcement Learning Agents

- Hierarchical Actor Critic (HAC) – 2017

Memory Types

- Prioritized Experience Replay (PER) – 2015
- Hindsight Experience Replay (HER) – 2017

Exploration Techniques

- E-Greedy
- Boltzmann
- Ornstein–Uhlenbeck process
- Normal Noise
- Truncated Normal Noise
- Bootstrapped Deep Q Network
- UCB Exploration via Q-Ensembles (UCB)
- Noisy Networks for Exploration
- Intrinsic Curiosity Module (ICM) – 2017

Meta Learning

- Model-agnostic meta-learning (MAML)– 2017
- Improving Generalization in Meta Reinforcement Learning using Learned Objectives (MetaGenRLis) – 2020

2. Model-Based

Dyna-Style Algorithms / Model-based data generation

- Dynamic Programming (DP) = DYNA-Q – 1990
- Embed to Control (E2C)– 2015
- Model-Ensemble Trust-Region Policy Optimization (ME-TRPO) – 2018
- Stochastic Lower Bound Optimization (SLBO) – 2018
- Model-Based Meta-Policy-Optimzation (MB-MPO) (meta learning) – 2018
- Stochastic Ensemble Value Expansion (STEVE) – 2018
- Model-based Value Expansion (MVE) – 2018
- Simulated Policy Learning (SimPLe) – 2019
- Model Based Policy Optimization (MBPO) – 2019

Policy Search with Backpropagation through Time / Analytic gradient computation

- Differential Dynamic Programming (DDP) – 1970
- Linear Dynamical Systems and Quadratic Cost (LQR) – 1989
- Iterative Linear Quadratic Regulator (ILQR) – 2004
- Probabilistic Inference for Learning Control (PILCO) – 2011
- Iterative Linear Quadratic-Gaussian (iLQG) – 2012
- Approximate iterative LQR with Gaussian Processes (AGP-iLQR) – 2014
- Guided Policy Search (GPS) – 2013
- Stochastic Value Gradients (SVG) – 2015
- Policy search with Gaussian Process – 2019

Shooting Algorithms / sampling-based planning

Random Shooting (RS) – 2017

Cross-Entropy Method (CEM)– 2013

- Deep Planning Network (DPN)-2018
- Probabilistic Ensembles with Trajectory Sampling (PETS-RS and PETS-CEM) – 2018
- Visual Foresight – 2016

Model Predictive Path Integral (MPPI) – 2015

- Planning with Deep Dynamics Models (PDDM) – 2019

Monte-Carlo Tree Search (MCTS) – 2006

- AlphaZero – 2017

2.8 Redes Generativas Adversarias

2.9 Actualidad y algunos conceptos relacionados con el Deep Learning

2.10 Software para aplicar Deep Learning

3 Análisis Causal y Modelos Gráficos Probabilísticos

3.1 Relaciones y Modelos Causales

3.2 Redes bayesianas: aprendizaje y clasificadores

3.2.1 Hipótesis MAP y Naive-Bayes

3.2.2 Modelos bayesianos

3.3 Modelos Ocultos de Markov

3.3.1 Tipos de cadenas de Markov

3.3.2 Aplicaciones de MOMs

3.4 Causal Machine Learning

3.4.1 Efectos causales. ATE y CATE

3.4.2 Uplift

4 Algoritmos Genéticos

Buenas referencias https://repository.urosario.edu.co/server/api/core/bitstreams/7ae959ec-81de-435b-aca4-e9bb365e4894/content

Buena documentación (graficos)

https://www.cs.us.es/~fsancho/Blog/posts/Algoritmos_Geneticos.md.html

http://www.robolabo.etsit.upm.es/asignaturas/irin/transparencias/AG.pdf

http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf

https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Bioinformatica/Tema%2006%20-%20AGs%20I.pdf

Individuo o Cromosoma Compuesto de Genes Codificación de genes (Binaria, Entera, Real)

4.1 Introducción

Los algoritmos genéticos es un método bastante común en minería de datos. Se inspiran en el **proceso natural de selección y evolución** tal y como se describe por la teoría evolucionista de la selección natural postulada por **Darwin**.

Los **principios** sobre los que se asientan los algoritmos genéticos son:

- Los individuos **mejor adaptados** al entorno son aquellos que tienen una probabilidad mayor de **sobrevivir** y, por ende, de **reproducirse**.
- Los descendientes **heredan** características de sus progenitores.
- De forma esporádica y natural se producen **mutaciones** en el material genético de algunos individuos, provocando cambios permanentes.

Los algoritmos genéticos son adecuados para obtener buenas aproximaciones en **problemas de búsqueda, aprendizaje y optimización** [Marczyk. 2004].

De forma esquemática un algoritmo genético es una **función matemática** que tomando como entrada unos individuos iniciales (**población origen**) selecciona aquellos **ejemplares** (también llamados individuos o cromosomas) que **recombinándose** por algún método generarán como resultado la **siguiente generación**. Esta función se aplicará de forma **iterativa** hasta verificar alguna condición de parada, bien pueda ser un número máximo de iteraciones o bien la obtención de un individuo que cumpla unas restricciones iniciales.

Condiciones para la aplicación de los Algoritmos Genéticos

No es posible la aplicación en toda clase de problemas de los algoritmos genéticos. Para que estos puedan aplicarse, los problemas deben cumplir las siguientes condiciones:

- El **espacio de búsqueda** [1] debe estar acotado, por tanto ser **finito**.
- Es necesario poseer una **función** de aptitud, que denominaremos **fitness**, que evalúe cada solución (individuo) indicándonos de forma cuantitativa cuán buena o mala es una solución concreta.
- Las **soluciones** deben ser **codificables** en un lenguaje comprensible para un **ordenador**, y si es posible de la forma más **compacta** y abreviada posible.

Habitualmente, la segunda condición es la más complicada de conseguir, para ciertos problemas es trivial la función de fitness (por ejemplo, en el caso de la búsqueda del máximo de una función) no obstante, en la vida real a veces es muy complicada de obtener y, habitualmente, se realizan conjeturas evaluándose los algoritmos con varias funciones de fitness.

Ventajas e inconvenientes

Ventajas

- No necesitan ningún conocimiento particular del problema sobre el que trabajan, únicamente cada ejemplar debe representar una posible solución al problema.
- Es un algoritmo admisible, es decir, con un número de iteraciones suficiente son capaces de obtener la solución óptima en problemas de optimización.
- Los algoritmos genéticos son bastante robustos frente a falsas soluciones ya que al realizar una inspección del espacio solución de forma no lineal (por ejemplo, si quisiéramos obtener el máximo absoluto de una función) el algoritmo no recorre la función de forma consecutiva por lo que no se ve afectada por máximos locales.
- Altamente paralelizable, es decir, ya que el cálculo no es lineal podemos utilizar varias máquinas para ejecutar el programa y evaluar así un mayor número de casos.
- Pueden ser incrustrables en muchos algoritmos de data mining para formar modelos híbridos. Por ejemplo para seleccionar el número óptimo de neuronas en un modelo de Perceptrón Multicapa.

Inconvenientes

- Su coste computacional puede llegar a ser muy elevado, si el espacio de trabajo es muy grande.
- En el caso de que no se haga un correcto ajuste de los parámetros pueden llegar a caer en una situación de dominación en la que se produce un bucle infinito ya que unos individuos dominan sobre los demás impidiendo la evolución de la población y por tanto inhiben la diversidad biológica.
- Puede llegar a ser muy complicado encontrar una función de evaluación de cada uno de los individuos para seleccionar los mejores de los peores.

[1] **Recordemos que cualquier método de Data Mining se puede asimilar como una búsqueda en el espacio solución, es decir, el espacio formado por todas las posibles soluciones de un problema**

4.2 Fundamentos teóricos (conceptos)

A continuación, se explican, someramente, los conceptos básicos de los algoritmos genéticos.

4.2.1 Codificación de los datos

Cada **individuo o cromosoma** está formado por unos cuantos **genes**. Estos individuos con sus genes los tenemos que representar de forma que podamos codificar esa información.

Los principales métodos de representación son: - **Binaria:** Los individuos/cromosomas están representados por una serie de genes que son bits (valores 0 ó 1).

- **Entera:** Los individuos/cromosomas están representados por una serie de genes que son números enteros.
- **Real:** Los individuos/cromosomas están representados por una serie de genes que son números reales en coma flotante.
- **Permutacional:** Los individuos/cromosomas están representados por una serie de genes que son permutaciones de un conjunto de elementos. Se usan en aquellos problemas en los que la secuencia u orden es importante.
- **Basada en árboles:** Los individuos/cromosomas están representados por una serie de genes que son estructuras jerárquicas.

El primer paso para conseguir que un ordenador procese unos **datos** es conseguir **representarlos** de una forma apropiada. En primer término, para codificar los datos, es necesario separar las posibles configuraciones posibles del dominio del problema en un **conjunto** de **estados finito**.

Una vez obtenida esta clasificación el objetivo es representar cada **estado** de **forma unívoca** con una cadena de caracteres (compuesta en la mayoría de casos por unos y ceros).

A pesar de que cada estado puede codificarse con alfabetos de diferente cardinalidad[2], uno de los resultados fundamentales de la teoría de algoritmos genéticos es el **teorema del esquema**, que afirma que la codificación **óptima** es aquella en la que los algoritmos tienen un alfabeto de cardinalidad, es decir el uso del **alfabeto binario**.

El enunciado del **teorema** del **esquema** es el siguiente: «*Esquemas cortos, de bajo orden y aptitud superior al promedio reciben un incremento exponencial de representantes en generaciones subsecuentes de un Algoritmo Genético.*»

Una de las ventajas de usar un alfabeto binario para la construcción de configuraciones de estados es la sencillez de los operadores utilizados para la modificación de estas. En el caso de que el alfabeto sea binario, los operadores se denominan, lógicamente, **operadores binarios**. Es importante destacar que variables que estén próximas en el espacio del problema deben preferiblemente estarlo en la codificación ya que la proximidad entre ellas condiciona un elemento determinante en la mutación y reproducibilidad de éstas.

[2]La longitud de las cadenas que representen los posibles estados no es necesario que sea fija, representaciones como la de Kitano para representar operaciones matemáticas son un ejemplo de esto

Es decir, dos estados que en nuestro espacio de estados del universo del problema que están consecutivos deberían estarlo en la representación de los datos, esto es útil para que cuando haya mutaciones los saltos se den entre estados consecutivos. En términos generales cumplir esta premisa mejora experimentalmente los resultados obtenidos con algoritmos genéticos.

En la práctica el factor que condiciona en mayor grado el fracaso o el **éxito** de la aplicación de algoritmos genéticos a un problema dado es una **codificación acorde** con los **datos**.

Otra opción muy común es establecer a cada uno de los posibles casos un **número natural** y luego codificar ese número en binario natural, de esta forma minimizamos el problema que surge al concatenar múltiples variables independientes en el que su representación binaria diera lugar a numerosos huecos que produjeran soluciones no válidas. Por ejemplo, tenemos 3 variables, las dos primeras tienen 3 posibles estados y la última dos, el número posible de estados es 3+3+2 = 8, combinando las 3 variables podemos codificar todo con 3 bits en comparación con los 2+2+1 = 5 bits necesarios que utilizaríamos en el caso de realizar el procedimiento anterior. En este ejemplo no sólo ahorraríamos espacio sino que además evitaríamos que se produjeran individuos cuya solución no es factible.

4.2.2 Algoritmo

Un algoritmo genético implementado en **pseudo código** podría ser el siguiente:

```
Generar de forma aleatoria una población inicial aleatoria. Cada individuo/cromosoma
tiene sus propios genes.

Mientras (condición de terminación es falsa).

Evaluar el fitness de cada uno de los individuos.

Selección de los individuos con mejor fitness.

Recombinación de los individuos.

Mutación de los indiviudos.
```

Un posible esquema que puede representar una posible implementación de algoritmos genéticos se muestra en la figura Figure 4.1 .

A continuación, en los siguientes apartados, se hará una descripción de las fases anteriormente expuestas:

Inicializar Población

Como ya se ha explicado antes el primer paso es inicializar la población origen. Habitualmente la inicialización se hace de forma **aleatoria** procurando una **distribución homogénea** en los casos iniciales de prueba. No obstante, si se tiene un conocimiento más profundo del problema es posible obtener mejores resultados inicializando la población de una forma apropiada a la clase de soluciones que se esperan obtener.

Evaluar Población

Durante cada **iteración** (generación) cada gen se decodifica convirtiéndose en un grupo de parámetros del problema y se evalúa el problema con esos datos.

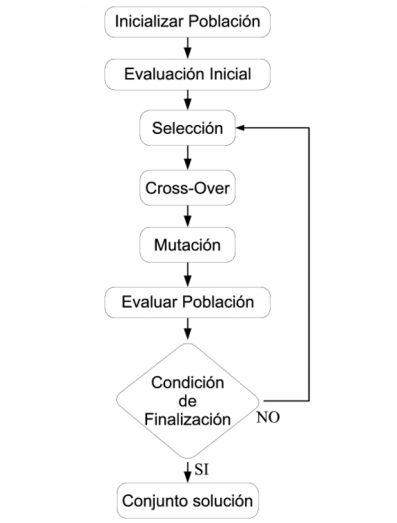


Imagen 4.1: Esquema de implementación de un algoritmo genético

Pongamos por ejemplo que queremos evaluar el máximo de la función $f(x) = x^2$ en el intervalo $[0, 1]$ y supongamos que construimos cada gen con **6 dígitos** $(2^6 = 64)$, por lo que interpretando el número obtenido en binario natural y dividiéndolo entre 64 obtendremos el punto de la función que corresponde al gen (individuo). Evaluando dicho punto en la función que queremos evaluar $(f(x) = x^2)$ obtenemos lo que en nuestro caso sería el **fitness**, en este caso cuanto mayor fitness tenga un gen mejor valorado está y más probable es que prospere su descendencia en el futuro. No en todas las implementaciones de algoritmos genéticos se realiza una fase de evaluación de la población tal y como aquí está descrita, en ciertas ocasiones se omite y no se genera ningún fitness asociado a cada estado evaluado. Selección La fase de selección elije los individuos a reproducirse en la próxima generación, esta selección puede realizarse por muy distintos métodos.

En el algoritmo mostrado en pseudo código anteriormente el **método** de **selección** usado depende del fitness de cada individuo. A continuación, se describen los más comunes:

Selección elitista: Se seleccionan los individuos con mayor fitness de cada generación. La mayoría de los algoritmos genéticos no aplican un elitismo puro, sino que en cada generación evalúan el fitness de cada uno de los individuos, en el caso de que los mejores de la anterior generación sean mejores que los de la actual éstos se copian sin recombinación a la siguiente generación.

Selección proporcional a la aptitud: los individuos más aptos tienen más probabilidad de ser seleccionados, asignándoles una probabilidad de selección más alta. Una vez seleccionadas las probabilidades de selección a cada uno de los individuos se genera una nueva población teniendo en cuenta éstas.

Selección por rueda de ruleta: Es un método conceptualmente similar al anterior. Se le asigna una probabilidad absoluta de aparición de cada individuo de acuerdo al fitness de forma que ocupe un tramo del intervalo total de probabilidad (de 0 a 1) de forma acorde a su fitness. Una vez completado el tramo total se generan números aleatorios de 0 a 1 de forma que se seleccionen los individuos que serán el caldo de cultivo de la siguiente generación.

Selección por torneo: se eligen subgrupos de individuos de la población, y los miembros de cada subgrupo compiten entre ellos. Sólo se elige a un individuo de cada subgrupo

para la reproducción.

Selección por rango: a cada individuo de la población se le asigna un rango numérico basado en su fitness, y la selección se basa en este ranking, en lugar de las diferencias absolutas en el fitness. La ventaja de este método es que puede evitar que individuos muy aptos ganen dominancia al principio a expensas de los menos aptos, lo que reduciría la diversidad genética de la población y podría obstaculizar la búsqueda de una solución aceptable.

Un ejemplo de esto podría ser que al intentar maximizar una función el algoritmo genético convergiera hacía un máximo local que posee un fitness mucho mejor que el de sus congéneres de población lo que haría que hubiera una dominancia clara con la consecuente desaparición de los individuos menos aptos (con peor fitness).

Selección generacional: la descendencia de los individuos seleccionados en cada generación se convierte en la siguiente generación. No se conservan individuos entre las generaciones.

Selección por estado estacionario: la descendencia de los individuos seleccionados en cada generación vuelve al acervo genético preexistente, reemplazando a algunos de los miembros menos aptos de la siguiente generación. Se conservan algunos individuos entre generaciones.

Búsqueda del estado estacionario: Ordenamos todos los genes por su fitness en orden decreciente y eliminamos los últimos m genes, que se sustituyen por otros m descendientes de los demás. Este método tiende a estabilizarse y converger.

Selección jerárquica: los individuos atraviesan múltiples rondas de selección en cada generación. Las evaluaciones de los primeros niveles son más rápidas y menos discriminatorias, mientras que los que sobreviven hasta niveles más altos son evaluados más rigurosamente. La ventaja de este método es que reduce el tiempo total de cálculo al utilizar una evaluación más rápida y menos selectiva para eliminar a la mayoría de los individuos que se muestran poco o nada prometedores, y sometiendo a una evaluación de aptitud más rigurosa y computacionalmente más costosa sólo a los que sobreviven a esta prueba inicial.

Recombinación.

Recombinación también llamada **Cross-over o reproducción**. La recombinación es el operador genético más utilizado y consiste en el **intercambio** de **material genético** entre **dos individuos** al azar (pueden ser incluso entre el mismo elemento). El material genético se intercambia entre **bloques**. Gracias a la presión selectiva[3] irán predominando los mejores bloques génicos.

Existen diversos **tipos** de **cross-over:**

Cross-over de 1 punto. Los cromosomas se cortan por 1 punto y se intercambian los dos bloques de genes.

Cross-over de n-puntos. Los cromosomas se cortan por n puntos y el resultado se intercambia.

[3]Presión Selectiva es la fuerza a la que se ven sometido naturalmente los genes con el paso del tiempo. Con el sucesivo paso de las generaciones los genes menos útiles estarán sometidos a una mayor presión selectiva produciéndose la paulatina desaparición de estos

Cross-over uniforme. Se genera un patrón aleatorio en binario, y en los elementos que haya un 1 se realiza intercambio genético.

Cross-over especializados. En ocasiones, el espacio de soluciones no es continuo y hay soluciones que a pesar de que sean factibles de producirse en el gen no lo son en la realidad, por lo que hay que incluir restricciones al realizar la recombinación que impidan la aparición de algunas combinaciones.

Mutación.

Este fenómeno, generalmente muy raro en la naturaleza, se modela de la siguiente forma: cuando se genera un gen hijo se examinan uno a uno los bits del mismo y se genera un coeficiente aleatorio para cada uno. En el caso de que algún coeficiente supere un cierto umbral se modifica dicho bit. Modificando el umbral podemos variar la probabilidad de la mutación. Las mutaciones son un mecanismo muy interesante por el cual es posible generar nuevos individuos con rasgos distintos a sus predecesores.

Los **tipos** de **mutación** más conocidos son:

```
En la imagen @fig-mutacion mostramos gráficamente este tipo.

![Esquema de mutación multibit de un algoritmo genético](imagenes/capitulo3/mutacion.png)
```

- **Mutación de gen**[4]: existe una única probabilidad de que se produzca una mutación de algún bit. De producirse, el algoritmo toma aleatoriamente un bit, y lo invierte.
- **Mutación multigen:** cada bit tiene una probabilidad de mutarse o no, que es calculada en cada pasada del operador de mutación multibit.
- **Mutación de intercambio:** Se intercambia el contenido de dos genes aleatoriamente.
- **Mutación de barajado:** existe una probabilidad de que se produzca una mutación. De producirse, toma dos genes aleatoriamente y baraja de forma aleatoria los genes, según hubiéramos escogido, comprendidos entre los dos.

Condición de finalización

Una vez que se ha generado la nueva población se evalúa la misma y se selecciona a aquel individuo o aquellos que por su fitness se consideran los más aptos.

4.2.3 Otros Operadores

Los operadores descritos anteriormente suelen ser operadores **generalistas** (aplicables y de hecho aplicados a todo tipo de problemas), sin embargo, para ciertos contextos suele ser más recomendable el uso de operadores específicos para realizar un recorrido por el espacio de solución más acorde a la solución buscada.

Modificadores de la longitud de los individuos. En ocasiones las soluciones no son una combinación de todas las variables de entrada, en estas ocasiones los individuos

[4] **Gen e Individuo en este contexto es lo mismo**

deberán tener una longitud variable[5]. Lógicamente, en este tipo de casos, es necesario modificar la longitud de los individuos, para ello haremos uso de los operadores añadir y quitar, que añadirán o quitarán a un individuo un trozo de su carga génica (es decir, un trozo de información).

4.2.4 Parámetros necesarios al aplicar Algoritmos Genéticos

Cualquier algoritmo genético necesita ciertos parámetros que deben fijarse antes de cada ejecución, como:

Tamaño de la población: Determina el tamaño máximo de la población a obtener. En la práctica debe ser de un valor lo suficientemente grande para permitir diversidad de soluciones e intentar llegar a una buena solución, pero siendo un número que sea computable en un tiempo razonable.

Condición de terminación: Es la condición de parada del algoritmo. Habitualmente es la convergencia de la solución (si es que la hay), un número prefijado de generaciones o una aproximación a la solución con un cierto margen de error.

Individuos que intervienen en la reproducción de cada generación: se especifica el porcentaje de individuos de la población total que formarán parte del acervo de padres de la siguiente generación. Esta proporción es denominada proporción de cruces.

Probabilidad de ocurrencia de una mutación: En toda ejecución de un algoritmo genético hay que decidir con qué frecuencia se va a aplicar la mutación. Se debe de añadir algún parámetro adicional que indique con qué frecuencia se va a aplicar dentro de cada gen del cromosoma. La frecuencia de aplicación de cada operador estará en función del problema; teniendo en cuenta los efectos de cada operador, tendrá que aplicarse con cierta frecuencia o no. Generalmente, la mutación y otros operadores que generen diversidad se suelen aplicar con poca frecuencia; la recombinación se suele aplicar con frecuencia alta.

4.3 Conclusiones

Los algoritmos genéticos es uno de los enfoques más originales en data mining. Su sencillez, combinada con su flexibilidad les proporciona una robustez que les hace adecuados a infinidad de problemas. No obstante, su simplicidad y sobre todo independencia del problema hace que sean algoritmos poco específicos. Recorriendo este capítulo hemos visto los numerosos parámetros y métodos aplicables a los algoritmos genéticos que nos ayudan a realizar una adaptación de los algoritmos genéticos más concreta a un problema. En definitiva, la implementación de esquemas evolutivos tal y como se describen en biología podemos afirmar que funciona.

[5]En muchas ocasiones, se realizan estudios de minería de datos sobre todos los datos existentes, encontrándose en ellos variables espúreas, es decir, variables que no aportan nada de información para el problema evaluado

4.4 Ejemplos prácticos

4.4.1 Algoritmos genéticos con R

Selección de variables

El objetivo del ejemplo es ver cómo podemos usar un algoritmo genético para hacer una selección de variables, quedándonos sólo con unas pocas.

Para ver que estamos acertados en la selección de variables vamos a tomar el ejemplo del **dataset** de **Iris**, que es un problema de **clasificación** con **3 clases**, cuenta con **150 muestras** y **4 variables** explicativas. Como queremos usarlo para selección de variables lo que vamos a realizar es meter de forma **sintética 10 variables más**, que siguen una **distribución normal** (0,1) y veremos el comportamiento del algoritmo a ver si realmente aparecen las variables originales como parte de las más importantes.

Tendremos que, para el algoritmo genético, nuestro cromosoma o individuo será un **vector** de tamaño 14 (**14 genes**), que representa las **14 variables** del dataset que hemos preparado.

En la imagen Figure 4.2 mostramosla población en una iteración N.

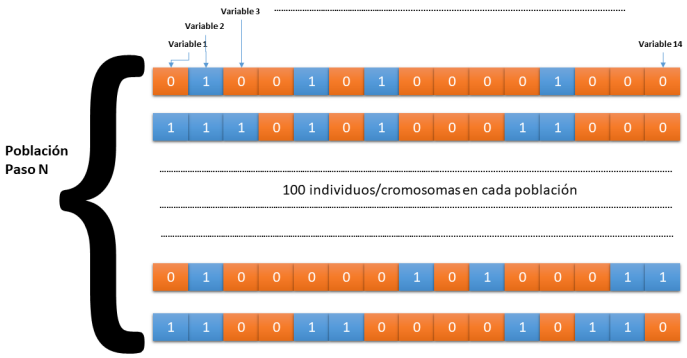


Imagen 4.2: Población en iteración N

Función Fitness (Evaluación)

Cuando estamos trabajando con selección de variables, el objetivo es conseguir el conjunto de variables que mejor modelo construyan según nuestro dataset. En este caso, al ser un problema de **clasificación**, veremos cual es la combinación de variables que nos da **menos errores al clasificar**.

Nuestra función fitness deberá seguir estos **pasos**:

- Recibe una **variable** llamada **indices** que tiene el tamaño del numero de variables (el tamaño del cromosoma) que hay en el dataframe (en nuestro caso 14) de datos.

  - Los valores son **1** si esa variable se va a **usar** y **0** si **no** se va a **usar**.

- Se construye un **modelo**, en este caso usamos **LDA** (Análisis Discriminante Lineal) con las **variables** que tienen **valor 1**.

- Calculamos el **error** que queremos minimizar (número de fallos)

- Para este caso del LDA cogemos los valores **$posterior** que nos dan la probabilidad de cada clase para cada entrada de la muestra

- Calculamos cual es el **máximo** y así le asignamos esa **clase** como su solución. También podríamos coger directamente el valor de $class con la clase dada como predicción.

- Verificamos cuantos hemos fallado y lo dividimos por el número de muestras para ver el **porcentaje** de **fallos**

- Devolvemos el **porcentaje de fallos**. El resultado de la ejecución del algoritmo evolutivo (rbga.bin())) nos dará un objeto del que tendremos que obtener que variables son las que queremos usar.

Figure 4.3

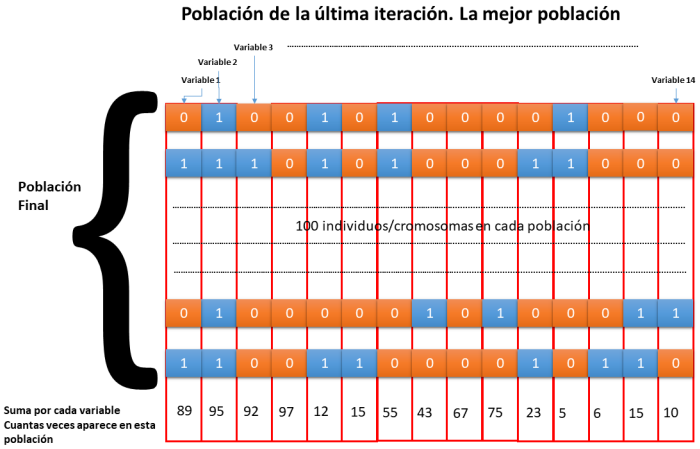


Imagen 4.3: Población Final

Una vez que nuestro algoritmo pare, deberíamos tener la población que mejor se ha adaptado según el fitness que habíamos definido.

En nuestro caso estarán las 100 mejores combinaciones de variables, que dan el menor error al clasificar. De esta manera si para cada variable contamos cuantas veces ha salido en cada elemento de la población, sabremos cuantas veces se ha usado en las combinaciones de variables de esta última iteración (que es la mejor hasta ese momento). Con lo cual podremos saber cuales han sido las **variables más usadas** en la población final.

El objeto **modelo_evolutivo**, que nos devuelve rbga.bin, tiene una variable **population** de dimensiones tamaño_población x numero_variables , en nuestro caso de 100x14, que tiene la información de la población de la **última iteración del algoritmo**, que en un principio debería ser la mejor. Este population contiene para cada fila (elemento en la población), 0 o 1 en la posición que corresponde a cada variable.

Para ver cuales son las variables que más se han usado tenemos que sumar por columnas y ese dato nos dará para cada columna (corresponde con una variable) la cantidad de veces que se ha usado en esta población. Una vez tenemos estos datos ya podemos

quedarnos con el número de variables que deseemos cogiendo las que más alto valor tienen.

Algunos de los **parámetros** que tenemos que definir y que algunos pueden afectar al rendimiento:

- **size:** Número de genes, en nuestro caso, número de variables
- **popsize:** Tamaño de la población con la que se trabaja en cada iteración. Cuanto más alto sea este valor, más tiempo tardará la ejecución. En este caso vamos a usar 100 elementos en la población, es decir, en cada iteración habrá 100 combinaciones de variables.
- **iters:** Número de iteraciones del algoritmo. Va a depender del problema pero seguramente entre 50 y 100 tendríamos una buena solución. Cuanto mayor sea esta variable, mayor será el tiempo de ejecución.
- **zeroToOneRatio:** Con esta variable le indicamos al algoritmo cual es el ratio de ceros frente a unos cuando se generan elementos de la población, es decir, más o menos nos tiene que dar una idea de cuantas variables se van a usar a la vez (las que tienen unos). Podríamos estimar que si tenemos 14 variables y queremos reducir a tener alrededor de 4, deberíamos intentar en cada iteración tener un rango parecido a ese. La forma de conseguirlo sería poner un valor a esta variable de 1, es decir, cada 1 cero hay 1 uno. Lo que más o menos nos dejaría alrededor de 7 variables con 1. Como esto se genera de forma aleatoria, más o menos estaremos en un rango de variables útil parecido a lo que queremos.
- **verbose:** Esta variable si la ponemos a TRUE se encargará de sacarnos información de la evolución del algoritmo. La recomendación es dejarla a FALSE ya que incrementa mucho el tiempo de ejecución.

—

“‘r

“‘

—

```
library(genalg) library(MASS)
```

```
data(iris) set.seed(999) # Alas variables reales del dataset, les añadimos 10 variables
```

```
iris.evaluate <- function(indices) {
```

```
result = 1 # Tiene que haber al menos 2 variables if (sum(indices) > 2) {
```

```
# Creamos un modelo de clasificación con LDA usando sólo las variables que vienen
# marcadas en la variablindices con valor 1
# El LDA tiene el valor $posterior que devuelve la probabilidad de cada clase (tenemos t
# Podríamos usar el valor de $class y así tendríamos directamente la clase.
modelo_lda <- lda(X[,indices==1], Y, CV=TRUE)$posterior

# Cogemos la probabilidad más alta apply(modelo_lda, 1,function(x)which(x == max(x))), p
```

```
# Comprobamos cuantos hemos fallado y lo dividimos por el tamaño de Y (150 muestras)
# El objetivo es que sea mínimo este número de fallos
result = sum(Y != dimnames(modelo_lda)[[2]][apply(modelo_lda, 1,function(x)which(x == max
```

```
result }
```

Ejecutamos el algoritmo evolutivo

```
system.time({ modelo_evolutivo <- rbga.bin(size=14, mutationChance=0.05, zeroToOneRatio=
```

Veamos como ha quedado el resultado de la población segúnla gráfica que nos da cuantas ve

Mostramos cual es el que ha aparecido más veces en la última iteración

```
uso_variables <- colSums(modelo_evolutivo$population)
```

Visualizamos cuanto se ha usado cada variable en esta última población

Construimos un dataframe con los datos

```
datos <- data.frame(variables=c(1:14),uso=uso_variables)
```

```
ggplot(datos,aes(x=variables,y=uso, fill=uso)) + geom_bar(stat="identity") + theme_minima
```

Obtenemos ahora cuales son las variables con los valores más altos. Primero vemos las sum

Figure 4.4

Imagen 4.4

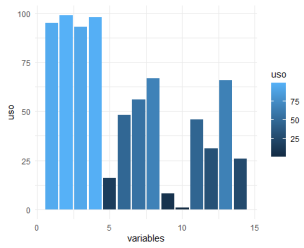


Imagen 4.4: Frecuencia de las Variables

```
posicion_maximos <- function(datos, cuantos) { variables <- NULL if( cuantos>0) { for( i
```

```
} variables
```

```
}
```

```
Obtenemos ahora cuales son las 6 variables más usados
```

```
posicion_maximos(uso_variables, 6) ## [1] 2 4 1 3 8 13
```

4.4.2 Algoritmos genéticos en Python

```
import os import pandas as pd import numpy as np import matplotlib.pyplot as plt import s

from genetic_selection import GeneticSelectionCV

from sklearn.preprocessing import StandardScaler from sklearn.model_selection import trai
```

Cargar datos de trabajo

```

os.chdir('C:/Users/p_san/Desktop/Máster_2020/Módulo_5') #directorio datos=pd.read_csv('g
```

Todas las variables son categóricas salvo:

duration

credit_amount

residence_since

age

existing_credits

num_dependents

Conversión a variables categóricas

```
datos\['checking_status'\]=datos\['checking_status'\].astype('category') datos\['credit_h
```

La variable class es una variable reservada en diferentes módulos de Python -> reemplazar por por target

```
datos.rename(columns={'class': 'target'}, inplace=True) datos\['target'\]=np.where(datos\
```

Definición de la muestra de trabajo

```
datos_entrada=datos.drop('target', axis=1) \# Datos de entrada datos_entrada= pd.get_dumm
```

datos de salida

```
respuesta=datos.loc\[:, 'target'\]
```

Escalado de las variables, partición de la muestra y Cross Validation

```
seed=123 \# Escalado de los datos de entrada x_esc=StandardScaler().fit_transform(datos_
```

Partición de la muestra

test_size=0.3 #muestra para el test x_train, x_test, y_train, y_test = train_test_split(x_esc,respuesta, test_size=test_size, random_state=seed, stratify=respuesta) Usando un modelo Cart from sklearn.tree import DecisionTreeClassifier cart=DecisionTreeClassifier(max_depth=5, random_state=seed) cart_algoritmo_gen=GeneticSelectionCV(cart, cv=5, # 5 particiones verbose=0, # no se muestran los resultados en la pantalla scoring="roc_auc", # ejemplo métrica para evaluar max_features=15, # número de variables máximas en la selección de
# características n_population=50, # tamaño de la población crossover_proba=0.5, # probabilidad de cruce entre parejas de genes mutation_proba=0.2, #probabilidad de mutación n_generations=40, #número de generaciones crossover_independent_proba=0.5, # prob. cruce para genes # independientes mutation_independent_proba=0.05, # prob. mutación de genes # independientes tournament_size=3, #tamaño de los grupos n_gen_no_change=10, # genes que se mantienen -> no pasan a # la segunda generación caching=True, n_jobs=1)

cart_algoritmo_gen=cart_algoritmo_gen.fit(x_train, y_train)
ajuste del modelo usando algoritmo genéticos para la selección de variables # Variables Seleccionadas print('Num. Var:', cart_algoritmo_gen.n_features_) # print(cart_algoritmo_gen.support_)
Resultados matriz numpy -> mala visualización. Los resultados se # convierten a df de pandas df=pd.DataFrame(cart_algoritmo_gen.support_, columns=['Variables'], index=x_train.columns) df=df.loc[~df['Variables'].isin([False])] #se elimina del df las variables no seleccionadas por el algoritmo list(df.index) # variables seleccionadas por el algoritmo

Num. Var: 9 ['checking_status_0<=X<200', 'checking_status_<0', "credit_history_'critical/other existing credit' ", "purpose_'used car' ", 'savings_status_>=1000', "personal_status_'male single' ", 'other_parties_guarantor', 'other_payment_plans_stores', 'job_skilled']

Resultados test - predicción & Matriz de confusión (modelo CART con selección de variables a través de Algoritmos Genéticos)

pred=cart_algoritmo_gen.predict(x_test)

confusion_matrix(y_test, pred) # Matriz de confusión

array([[173, 37], [46, 44]], dtype=int64)

5 Lógica Difusa

5.1 Conceptos clave.

5.1.1 Conjuntos difusos y funciones de membresía

5.1.2 Inferencia y Modelamiento difuso

5.2 Ejemplos prácticos

5.2.1 Clustering Difuso

5.2.2 Herramientas de Diagnóstico

Bibliografia

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

A Anexo 1

Anexo 1