|  |  |
| --- | --- |
| **Logo** | **UNIVERSIDAD**  **REY JUAN CARLOS** |

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA E   
INGENIERÍA DEL SOFTWARE

Curso Académico 2016/2017

Trabajo Fin de Grado

PREDICCIÓN DE LA DEMANDA DE ENERGÍA EN ESPAÑA A TRAVÉS DE ALGORITMOS METAHEURÍSTICOS Y REDES NEURONALES

**Autor:** César Valdés Martínez

**Directores/Tutor:** Jesús Sánchez-Oro Calvo  
Abraham Duarte Muñoz

**ÍNDICE**

[1. Introducción (<10 págs.) 1](#_Toc505013329)

[1.1. Modelo analítico 4](#_Toc505013330)

[1.2. Modelo no analítico 5](#_Toc505013331)

[2. Objetivos (<1 pág.) 7](#_Toc505013332)

[3. Descripción algorítmica (16 págs. 8 para cada cosa 2-2-4) 8](#_Toc505013333)

[3.1. Redes neuronales 8](#_Toc505013334)

[3.1.1. Estructura de un Sistema Neuronal Artificial 10](#_Toc505013335)

[3.1.2. Modelo neuronal 10](#_Toc505013336)

[3.1.3. Métodos de aprendizaje 12](#_Toc505013337)

[3.1.4. La elección: el perceptrón multicapa (MLP) 14](#_Toc505013338)

[3.2. Metaheurísticas 16](#_Toc505013339)

[3.2.1. Optimización 16](#_Toc505013340)

[3.2.2. Técnicas exactas 17](#_Toc505013341)

[3.2.3. Técnicas heurísticas 19](#_Toc505013342)

[3.2.4. Técnicas metaheurísticas 20](#_Toc505013343)

[3.2.5. La elección: GRASP (Greedy Randomized Adaptive Search Procedure) 22](#_Toc505013344)

[4. DESCRIPCIÓN INFORMÁTICA (<15 págs.) 24](#_Toc505013345)

[4.1. Lenguajes y programas. 24](#_Toc505013346)

[4.1.1. Python 24](#_Toc505013347)

[4.1.2. Java 25](#_Toc505013348)

[4.1.3. PyCharm 25](#_Toc505013349)

[4.1.4. NetBeans y Eclipse 26](#_Toc505013350)

[4.2. Librerías y herramientas 26](#_Toc505013351)

[4.2.1. PyBrain y Neuroph 26](#_Toc505013352)

[4.2.2. Maven 27](#_Toc505013353)

[4.2.3. Git (GitHub) 27](#_Toc505013354)

[4.2.4. ObjectAid 28](#_Toc505013355)

[4.2.5. SonarQube 28](#_Toc505013356)

[4.3. Metodología de desarrollo software 29](#_Toc505013357)

[4.3.1. La elección: El método incremental. 30](#_Toc505013358)

[4.4. Estructura del software 31](#_Toc505013359)

[4.4.1. Interfaz Gráfica 33](#_Toc505013360)

[4.4.2. Global 35](#_Toc505013361)

[4.4.3. Redes Neuronales 36](#_Toc505013362)

[4.4.4. Metaheurísticas 38](#_Toc505013363)

[4.4.5. Optimizadores de metaheurísticas 40](#_Toc505013364)

[5. EXPERIMENTOS (<10 págs.) 42](#_Toc505013365)

[5.1. Redes Neuronales 42](#_Toc505013366)

[5.1.1. Función de transferencia de las neuronas 42](#_Toc505013367)

[5.1.2. Número de iteraciones máximo en el proceso de entrenamiento 42](#_Toc505013368)

[5.2. Metaheurísticas 43](#_Toc505013369)

[6. CONCLUSIONES Y TRABAJOS FUTUROS (2 págs.) 44](#_Toc505013370)

[6.1. Especificación de requisitos 45](#_Toc505013371)

[6.1.1. Funcionales 45](#_Toc505013372)

[6.1.2. No funcionales 45](#_Toc505013373)

# Introducción (<10 págs.)

Hoy en día vivimos en un mundo globalizado, en el que la población aumenta de manera exponencial y sustentado en una industrialización feroz. Estos fenómenos inducen un aumento también exponencial del consumo energético mundial, que está en clara confrontación con la propia naturaleza de nuestro planeta y su funcionamiento, pudiendo comprometer seriamente los años venideros.

Como podemos ver en el gráfico, en el año 2014 el consumo energético era cubierto en un 80% por energías no renovables (principalmente carbón, petróleo o gas) y este indicador es aún mayor en países en vías de desarrollo.

Además, las actividades industriales están detrás del 50% de la demanda energética mundial, y, por ende, países con mayor crecimiento económico tienden a tener mayor demanda de energía que otros con economías basadas en sectores alternativos.

Por todo ello, la administración a medio y largo plazo de la demanda energética, así como el crecimiento de centrales basadas en renovables se ha convertido en un problema clave con un gran impacto en todas las economías y naciones en desarrollo.

De hecho, ya hay estudios que afirman que para el año 2050 el 90% de la energía provendrá de fuentes renovables:



Hace unos años ya se predijo un incremento de la demanda de energía mundial de más de un 50% en los siguientes 20 años, en lo que parecía un proceso imparable.

Sin embargo, todas esas predicciones fallaron cuando surgió la crisis mundial del 2008.

El problema principal en la estimación de la demanda energética a un nivel nacional es que dicha estimación depende directamente de una serie de variables macroeconómicas que en la mayoría de casos se calculan anualmente. Por este motivo, generalmente se disponen de muy pocos datos para construir un modelo predictivo consistente.

Para más inri, la naturaleza de un país va cambiando a lo largo del tiempo. Si observamos su economía hace 30 años y la comparamos con la que puede tener hoy día es muy probable que encontremos diferencias sustanciales, lo que restringe aún más la variedad de indicadores históricos macroeconómicos que se pueden considerar para la estimación.

Dicho esto, la primera aproximación para combatir el problema se propone en [3], donde un algoritmo genético se usó para obtener los parámetros de un modelo de predicción exponencial. Específicamente las entradas del modelo eran 4 variables macroeconómicas de Turquía (producto interior bruto, población, importaciones y exportaciones), con datos desde los 80 hasta los primeros años de los 2000.   
La predicción de la demanda de energía se hizo para el mismo año que las variables de entrada y se consideraron modelos tanto lineales como exponenciales.

La función objetivo era una medición del error medio cuadrático entre el dato real y el resultado del modelo, obtenido sobre los datos de entrenamiento (una fracción de todos los datos disponibles). Con los modelos obtenidos, se probó que la demanda de energía del futuro podía ser estimada mediante la proyección de variaciones en los parámetros de entrada. En este caso, estas proyecciones predijeron un incremento continuo de la demanda de energía en Turquía por los próximos 20 años.

La mayoría de los siguientes trabajos se han centrado en probar el rendimiento de los diferentes algoritmos evolutivos cuando son aplicados a este problema, tales como Particle Swarm Optimization (PSO) [4,5]) o algunas aproximaciones hibridas basadas en PSO y Ant Colony Optimizacion (ACO) [7]. Otro acercamiento hibrido fusionando PSO y GA ha sido reportado recientemente en [6,8,10] para la estimación de demanda energética en China. Otros acercamientos se han elaborado en modelos de predicción con un acercamiento distinto que las exponenciales usadas en [3]. Así, en [11], diversos nuevos modelos han estado basados en funciones alternativas exponenciales y logarítmicas, optimizados por un algoritmo genético en tiempo real.

En todas estas aproximaciones se considera un número reducido de factores (variables de entradas o características) a partir de los que las proyecciones muestran un incremento sostenido de la demanda energética en los próximos años. En todos los casos los años de entrenamiento no incluyen datos de más allá del año 2005, por lo que se están perdiendo eventos importantes que tienen un impacto directo en la calidad de la predicción calculada (por ejemplo, la crisis del año 2008).

Este estudio se aborda desde otra novedosa perspectiva, que combina solvers evolutivos y la computación neuronal para construir una metodología eficiente que nos ayude a resolver el problema.

En primer lugar, nuestra estimación se realiza con un año de antelación. Esto es una gran diferencia con respecto a los otros enfoques donde los parámetros de entrada y demanda de energía se tomaban del mismo año. Como añadido, se consideran un número más grande de variables predictivas que en estudios anteriores para dotar de más información al modelo predictivo, y además disponemos de los datos recogidos en los últimos 30 años, desde 1981 hasta el 2011.

**Definición del modelo predictivo**

Se considera el conjunto de los datos de demanda energética para un país dado, con n valores discretos correspondiente a distintos años; un conjunto de variables predictivas con . Un modelo proporciona una estimación para .

En este estudio se propone un acercamiento por dos vías, la primera a través de la creación de un modelo analítico llevado a cabo por una metaheurística con método Grasp y uno no analítico de la mano de una red neuronal perceptrón multicapa.

En los dos casos, la función objetivo a minimizar será el error medio cuadrático (*MSE*) calculado entre el consumo real y el predicho por cada uno de nuestros modelos:

Siendo n\* un subconjunto de n de entrenamiento.

## Modelo analítico

Tiene como objetivo encontrar el mejor conjunto de todas las m posibles características presentes en X, así como los valores de los parámetros del modelo para los cuáles la función objetivo *MSE* alcanza los valores mínimos.

Esto corresponde a un tipo de problemas llamados de Selección de características, *Feature Selection* (FS). Esta tarea es muy importante en problemas de clasificación y regresión supervisadas ya que al introducir características innecesarias en el proceso de entrenamiento se produce un aumento del coste y tiempo de procesamiento mientras que se degrada la propia predicción ya que se introduce en el cálculo un parámetro que no está relacionado con la salida buscada. [14].

Los problemas de selección de características se pueden plantear de dos maneras distintas:

* Independientemente del rendimiento del modelo, que preserva la mayoría de la información que proviene de los datos. Es conocido como *filter method* para selección de características
* En función del rendimiento del modelo, que selecciona directamente un subconjunto de características del total, de manera que el rendimiento del modelo se mejore o al menos no empeore. Conocido como *wrapper method*. Generalmente resultan más potentes que los de filtrado, aunque el coste computacional sea más elevado.

El filtrado de características se puede realizar mediante cualquier algoritmo de búsqueda como ascenso de colinas (*hill-climbing*), voraz (*greedy*) o solvers evolutivos (*evolutionary* *solvers*) entre otros.

En este caso, hemos usado un GRASP (Greedy Randomized Adaptive Search Procedure), y hemos elegido un modelo exponencial, como ya se sugirió y usó en [3]. La función que modelará y guiará la búsqueda será la siguiente:

En esta función, podemos observar que todas las son parámetros de nuestro problema que se encargan de dar un peso a cada una de las variables macroeconómicas de las que disponemos. A su vez, también vemos que es una variable global cuya acometida es realizar un último ajuste a la estimación del modelo.

Es importante subrayar, que el valor de todas las variables de entrada deben estar normalizadas para evitar posibles problemas de escala con el modelo de regresión.

A su vez, los pesos también están normalizados en el rango [-1, 1], mientras que estará en un rango mayor de [-5, 5]

Mediante el proceso Grasp se irán ajustando todos estos pesos para finalmente dar con la mejor configuración.

## Modelo no analítico

La parte no analítica del proyecto está constituida por una red neuronal. El objetivo será obtener la configuración óptima para confluir en una estimación de demanda de energía lo más acertada posible.

Los beneficios del uso de redes neuronales en la resolución de proyectos son bien conocidos, su capacidad de generalización, su tolerancia a errores, su alto nivel de computación, etc. Todo esto aporta un gran valor añadido a este estudio, ya que se cree que tras crear suficientes redes mediante un entrenamiento supervisado basado en el conjunto de datos del que se dispone se puede obtener una capaz de realizar estimaciones extremadamente precisas, e incluso, prever grandes depresiones o catástrofes económicas o sociales antes que cualquier empresa o humano.

La algoritmia interna de una red neuronal y sus procesos es muy compleja, tras estudiar la viabilidad de crear un modelo propio se decidió delegar toda la implementación a la librería Neuroph (Punto X.X). Así, nuestros esfuerzos se pueden centrar al cien por cien en el estudio de los parámetros que nos permitan obtener la red ideal, es decir, aquella que tenga mayor capacidad de previsión con mayor precisión.

En el caso de una red neuronal, la función que determinaría la estimación vendría dada por la salida de la única neurona de la última capa, definida por:

Para n igual al número de neuronas presente en la capa anterior, f la función de transferencia y W y X el peso y valor de las neuronas de la capa anterior. Esta función es la misma para calcular la salida de cada una de las neuronas presentes en la red.

Continuar con lo que me dijo Abraham, proceso analítico meta y no analítico la red, lo que queremos hacer, tal cual… Quien ha tratado y los intereses

Descripción del problema (real y matemática). Estimación de la energía y tal, y posteriormente como se modela con los alfas y betas, que te dan una estimación del modelo real. Sacar del artículo primero que me pasó Jesús. Poner un ejemplo de cómo se evalúa

Quien ha trabajado en este tema.

Repercusiones prácticas. por qué es interesante resolver este problema. Que ventaja tiene una persona que pueda tomar decisiones saber cuál va a ser el consumo energético que puede tener al año siguiente.

Propuesta muy resumida.

# Objetivos (<1 pág.)

El objetivo de este trabajo es único, desarrollar un **algoritmo** robusto y eficaz que ayude a realizar la estimación del gasto energético de un país a un año vista.

Este objetivo se descompone en dos tipos de sub-objetivos:

* Generales
  + Incrementar los conocimientos en java
  + Familiarizarme más con los entornos de desarrollo cono Eclipse, NetBeans o PyCharm
  + Utilizar el sistema de control de versiones Git
* Específicos
  + Adquirir conocimientos en el área de las metaheurísticas
  + Aprender el funcionamiento interno de una red neuronal y su uso

Eclipse, Java, aprender redes neuronales, git… MIRAR OTROS TFGS

Probar varios Redes, constructivos, búsquedas locales, metaheurísticas, revisar el estado del arte (los trabajos previos).

# Descripción algorítmica (16 págs. 8 para cada cosa 2-2-4)

En este apartado se explicará en detalle los algoritmos que se han usado.

## Redes neuronales

La estructura neuronal del sistema nervioso fue descubierta por Santiago Ramón y Cajal en 1888 gracias a la técnica de tinción de Golgi. Las estimaciones indican que un sistema nervioso contiene cerca de cien mil millones de neuronas.



Axón

Dendritas

Soma

A nivel funcional, no son más que procesadores sencillos de información, con un canal de entrada (las dendritas), uno de computación (el soma) y uno de salida (el axón). La transmisión de esa información entre las neuronas se realiza mediante impulsos eléctricos; para que esto ocurra ambas neuronas tienen que estar conectadas. A esta unión la llamamos sinapsis.

Cada neurona tiene de media unas 7000 conexiones sinápticas, y cada una de ellas puede tener un efecto positivo o negativo sobre la transmisión del impulso, de manera que una neurona procesará cada uno de esos efectos para transmitir una señal que será combinación de todas.

El sistema nervioso también tiene una organización interna, ya que observando el cortex podemos apreciar una organización tanto vertical como horizontal.

Aunque los sistemas electrónicos avanzan a pasos agigantados, aún hay ciertas tareas en las que el cerebro se presenta como la opción más eficiente. A grandes rasgos, las tareas de alto nivel como el cálculo o el razonamiento son resueltas con facilidad por ordenadores, pero las de bajo nivel, de percepción, control, reconocimiento de patrones, etc, son resueltas con mayor facilidad por el cerebro.

Dado que esos problemas son eficazmente resueltos por nuestro cerebro, surge la idea de copiar la estructura que presenta una red neuronal biológica, creando un sistema de neuronas artificiales conectadas entre sí, formando así un sistema auto organizado y posteriormente analizar si los resultados obtenidos se acercan a los biológicos.

Obviamente en ningún momento se intenta crear un sistema tan complejo como un cerebro, sino utilizar estos sistemas para tareas más sencillas como reconocimiento de caracteres en matrículas, formas, formular estimaciones, etc. Todas estas tareas se resuelven con resultados excelentes, lo que provoca que actualmente en este campo confluyan numerosas ramas de la ciencia como pueden ser la electrónica, física, matemáticas, ingenierías…

Hay tres conceptos clave a la hora de emular un sistema nervioso:

1. Paralelismo de cálculo: Mientras un procesamiento secuencial de una imagen de grandes dimensiones y resolución puede llegar a llevar minutos, hacerlo de manera paralela reduce el tiempo significativamente. Un sistema nervioso humano es capaz de capturar, analizar rasgos y características e interpretar la imagen que capturan nuestros conos y bastones de nuestra retina al instante (un ojo saludable tiene aproximadamente 7 millones de conos y 125 millones de bastones, lo que significa que la calidad de la imagen podría llegar a más de 100 megapíxeles).
2. Memoria distribuida: En un ordenador convencional la información ocupa posiciones de memoria perfectamente definidas. En una red neuronal esa información se encuentra distribuida en la red, (una biológica cuenta con redundancia de manera que varias neuronas pueden desempeñar prácticamente la misma función). Esta distribución hace que una corrupción en la red provoque solo una pequeña pérdida de toda esa información, es decir, genera un sistema tolerante a fallos.
3. Adaptabilidad al entorno: Las redes neuronales se adaptan al entorno modificando entre otras cosas su sinapsis, y además aprenden con la experiencia. En nuestro campo podemos explicarlo como generalización a partir de ejemplos.

### Estructura de un Sistema Neuronal Artificial

El elemento más simple es la neurona artificial. Esta se organizará en capas; varias capas constituirán una red neuronal; y una o varias redes junto con las interfaces de entrada y salida **más los módulos convencionales necesarios,** constituirían un sistema global de proceso:

Foto de wikipedia



Aparte de las neuronas debemos de contar con los siguientes elementos:

1. Patrón de conectividad (arquitectura)
2. Una dinámica de activaciones
3. Una dinámica de aprendizaje
4. Un entorno

### Modelo neuronal

La neurona es el elemento de cálculo mínimo de nuestra red. Su función es a partir de unos valores de entrada procedentes de otras neuronas o del exterior generar una única respuesta.

El modelo más genérico para una neurona i contiene:

* **Conjunto de entradas** - : Pueden ser binarias o continuas, o como en el caso del perceptrón multicapa, admitir ambos tipos en función de la naturaleza del problema.
* **Pesos sinápticos** - : Representa la intensidad con la que se transmite la entrada entre la neurona presináptica j y la postináptica i. Si el peso es positivo tenderá a excitar la entrada y si es negativo, a inhibirla.
* **Regla de propagación** - : Proporciona el valor del potencial postsináptico en funcion de las entradas y los pesos sinápticos. La función más habitual suele ser el sumatorio del producto de cada entrada con su peso sináptico.  
  Aunque en modelos basados en el cálculo de distancias como RBF (Radial Basis Function), mapas de Kohonen o LVQ viene definida por la distancia euclídea:
* **Función de activación o transferencia** : : Proporciona el estado de activación de la neurona i en funcion de su estado anterior y de su potencial postsináptico actual.

No obstante, en otros modelos se considera que el estado de la neurona actual no es afectado por el estado anterior, quedando: .

Las funciones de activación más usadas son: (gráficos de https://www.desmos.com)

Gaussian

**Gaussian**:

Log

**Log**



**Linear**

**RectifiedLinear**

RectifiedLinear

Linear

**Ramp**

Ramp

Sgn



**Sgn**

Sigmoid

**Sigmoid**



**Sin**

Sin

Tanh

**Tanh**

* **Función de salida** - : Proporciona la salida de la neurona en función de su estado de activación. En muchos modelos, la función de salida es la identidad, de tal manera que el resultado de la función de activación es la propia salida (como ocurre en MLP).

Aun así, en otros modelos, la salida puede tener una función de escalón para asegurar que la función de activación supere un umbral, o incluso funciones estocásticas para que la salida genere un comportamiento probabilístico.

Así, la operación de salida de una neurona genérica sería la siguiente:

**Neurona estándar**

Si bien todos los conceptos anteriores se aplican a una neurona genérica, de manera global se utiliza un modelo más simple, en concreto en el que la función de propagación es la suma ponderada de peso y entrada () y la función de salida correspondería a la identidad, quedando como variable la función de activación.

Hay otros modelos de neuronas, como pueden ser las todo-nada, la continua sigmoidea o la estocástica.

### Métodos de aprendizaje

Al construir una red neuronal se parte de una configuración determinada (tipo de neurona, arquitectura de red…), y se establecen todos los pesos iniciales como nulos o aleatorios. Evidentemente, esta primera red neuronal no tiene ninguna capacidad especial, y los resultados que produce son completamente desechables. Para que la red produzca los resultados que buscamos primero hay que realizar un proceso de entrenamiento o aprendizaje.

Definimos aprendizaje como el proceso (por lo general, iterativo) mediante el cual se produce el ajuste de los parámetros libres de la red. Usualmente solo consiste en determinar el conjunto de pesos sinápticos que generen una configuración de red que realice correctamente el cálculo o procesamiento deseado.

Hay dos métodos principales de aprendizaje:

* Entrenamiento supervisado: A la red se le presentan un conjunto de datos junto con las salidas u objetivos deseados. Ésta iterativamente, ajustará los pesos hasta que la salida tienda a ser la deseada.
* Entrenamiento no supervisado: Se presentan a la red un conjunto de patrones sin especificar la salida deseada. Así, la red estimará un conjunto de entradas con rasgos comunes y agrupar patrones según su similitud.

Aparte de estos métodos, hay otros como el aprendizaje hibrido, en el que coexisten ambos tipos dependiendo de la capa, o el aprendizaje reforzado, con información sobre el error que se comete, pero sin proporcionar la salida (también llamado aprendizaje por premio-castigo).

Podemos clasificar los tipos de redes neuronales en función del tipo de entrenamiento:



Hay varios tipos perceptrón, bla bla, nos centramos en la multilayer. ¿PONER COMO SE PRODUCE UN ENTRENAMIENTO? NO CABE

Después descripción particular de la red, con tantas neuronas de entrada, 1 de salida, tantas capas internas, como he llegado.

### La elección: el perceptrón multicapa (MLP)

Debida a la naturaleza de este problema, de todas las combinaciones se ha decidido optar por una configuración de perceptrón multicapa.

Cuenta con neuronas estándar (ver x.xx), y su naturaleza se basa en añadir un número indeterminado de capas intermedias a un perceptrón simple. Las únicas diferencias entre un perceptrón simple y uno multicapa reside en el número de capas, la complejidad del entrenamiento y la capacidad (y, por tanto, las aplicaciones) que presenta cada uno.

El entrenamiento se produce mediante retropropagación (*BackPropagation*) o en algunos casos *ResilientPropagation* (una mejora al *BackPropagation*). A este tipo de redes multicapa se les denomina redes de retropropagación.

Las MLP surgieron tras observar las limitadas capacidades computacionales de las redes perceptrón simple. Se comprobó experimentalmente que daban unos resultados realmente buenos al ser aplicadas a problemas complejos, pero se tardaron varios años en demostrar esas capacidades de manera científica. Tras varios estudios por parte de diversas personas (McCulloch y Pitts, Denker, Lippmann, Lapedes y Farber) fue Hecht-Nielsen el que demostró que una arquitectura similar al MLP con una capa oculta resultaba ser un aproximador de funciones y finalmente otros grupos pudieron demostrarlo para el caso concreto de la red MLP.

Tras esto, el campo no está cerrado, ya que aún quedan diversos campos de estudio, como por ejemplo el número mínimo de capas o neuronas que se necesitan para representar una función concreta. ¿**Sigue hoy en día esta cuestión? El libro es del 2002**

Una vez elegido el modelo de red que se pretende estudiar, hay varios parámetros que tendremos que ajustar para obtener una red cuyos cálculos sean lo más precisos posibles.

El número máximo de ***iteraciones*** como condición de parada para el entrenamiento.

La clase de ***retropropagación*** que se usará

El ***learningRate*** encargado de en el proceso de entrenamiento de la red dar un peso a la iteración anterior con respecto a la actual.

La ***función de transferencia*** que proporciona el estado de activación de cada neurona, generando un cambio en la transmisión del pulso.

El ***número de capas*** intermedias de electrones que va a haber en la red.

El ***número de electrones*** que va a haber en cada capa.

Para analizar todas las posibles redes a partir de las combinaciones de estos parámetros se ha creado un algoritmo que de manera automática las explora y va almacenando tanto la red obtenida como una imagen asociada con la evolución del error a lo largo del entrenamiento.

Posteriormente, otro algoritmo será el encargado de seleccionar cual de todas las redes presenta el menor error medio cuadrático, y será la red elegida.

Este experimento, estos datos, lo que he probado… Explicar todo lo que no sean números, de implementación nada. Funciones de activación, etc. etc.

COMO MUCHO LLEGAR A PSEUDOCÓDIGO, quizás mejor diagrama de flujo…

Después descripción particular de la red, con tantas neuronas de entrada, 1 de salida, tantas capas internas, como he llegado.

## Metaheurísticas

### Optimización

La búsqueda de las mejores soluciones mediante métodos de optimización dado un problema se ha convertido en un campo de importancia tanto en la vida real como en el campo de la ingeniería. De manera constante y subconsciente resolvemos problemas de optimización como encontrar el camino más corto entre dos puntos, organizar agendas o trayectos… La complejidad de estos problemas es sencilla, pero al enfrentarnos a otros más complejos y grandes, resulta imposible no utilizar un ordenador para resolverlos.

Cuando hablamos de un problema de optimización nos referimos a un problema que presenta varias soluciones y un método de comparación entre ellas.

Asumiendo el caso de una minimización, y dado S como el espacio de soluciones (o de búsqueda) del problema y f como la función objetivo , un problema de optimización consiste en encontrar una solución, tal que se satisfaga la siguiente desigualdad:

El caso de maximización tampoco presentaría ningún problema ya que

Además, en función del dominio al que pertenezca S, se pueden definir problemas de optimización binaria, entera, continua o heterogénea.

Para resolver problemas de optimización se han desarrollado diferentes métodos a lo largo de la historia. Podemos crear una clasificación en función de esas técnicas entre exactas o aproximadas.

### Técnicas exactas

Como su propio nombre indica, estas técnicas nos garantizan la obtención de la solución óptima de cualquier instancia de un problema en un tiempo acotado, pero crece exponencialmente junto con el tamaño del problema.

Hay problemas cuya naturaleza obliga al uso de estas técnicas, son aquellos para los cuales no se ha encontrado un método que no implique el estudio de todo el espacio de soluciones, o aquellos cuyo objetivo es la obtención del óptimo necesariamente. Un ejemplo sencillo puede ser la obtención de números primos, para lo cual es necesario iterar por todos los números, ya que no hay un método conocido que nos evite estudiar todo el espacio de soluciones. Un caso más concreto podemos encontrarlo en la investigación médica, donde se realizan estudios de gran complicación cuyo objetivo es encontrar un compuesto determinado para derrotar una enfermedad. Estudios tan complejos como éste, provocan la necesidad de recurrir a la computación colaborativa (más info en [www.worldcommunitygrid.org](http://www.worldcommunitygrid.org)).

Las principales técnicas exactas son las siguientes:

1. Divide y vencerás: El objetivo es descomponer el caso en subcasos más pequeños del mismo tipo, posteriormente resolver cada uno de esos subcasos (recursivamente) y terminar uniendo las soluciones de cada subcaso para obtener una solución.
2. Algoritmos voraces: Generalmente utilizados para problemas de maximización o minimización, construyen la solución por etapas, escogiendo en cada una al mejor candidato que verifique las restricciones hasta completar una solución válida.
3. Backtracking: Consiste en el recorrido en profundidad de un árbol de búsqueda, generando soluciones parciales a medida que se avanza, cada nodo hoja representa una solución. Cualquier solución no factible implica la no exploración del nodo y la continuación con un nodo vecino no visitado.
4. Ramifica y Poda: Similar al backtracking, primero se genera una posible solución mediante otro método, se recorre el árbol, generando los posibles nodos hijos, pero en este caso se calcula una cota para cada nodo hijo que representará el mejor valor posible de nuestra función objetivo por ese camino. Si es peor que la solución, se poda. Si no, se explora hasta llegar a una solución.

Dentro de todos los problemas de optimización que existen podemos encontrar una categoría concreta denominada “problemas de optimización combinatoria”, que consisten en encontrar un óptimo entre un conjunto finito de posibilidades.

Los ejemplos más típicos de problemas de optimización combinatoria son: el problema del viajante, el problema de la asignación cuadrática, problemas de planificación o de cortes sobre grafos.

Una particularidad de estos problemas es que está garantizado que existe un método mediante el cual obtener la solución óptima, la *enumeración* del conjunto de soluciones. Aun así, el hecho de que se pueda obtener teóricamente el óptimo no significa que sea viable, ya que la realidad es que todos los problemas con interés son demasiado complejos como para poder aplicar dicho método debido al incremento exponencial en los tiempos de resolución de los mismos.

Los problemas combinatorios se han prestado a diversos estudios, a partir de los cuales se dedujo la existencia de varios subconjuntos:

* Clase P: Su resolución es factible debido a que se conocen métodos de obtención del óptimo cuyo tiempo de ejecución crece de manera polinómica.
* Clase NP: No se conocen métodos para los cuales el tiempo crezca de manera polinómica y se resuelva de forma exacta. Para obtener el óptimo solo queda el método enumerativo, cuyo tiempo de ejecución es exponencial.

Debido a la gran cantidad de problemas de interés científico que se encuentran en el tipo NP, se plantea la posibilidad de crear un tipo de algoritmos que sacrifiquen la garantía de encontrar el óptimo (aunque técnicamente pueden seguir encontrándolo no es seguro), a cambio de asegurar un mejor rendimiento en cuanto a tiempo de resolución, los algoritmos aproximados.

### Técnicas heurísticas

Los métodos heurísticos se postulan como las mejores técnicas de resolución de problemas de gran complejidad, donde la calidad de la solución no deja de lado la importancia que tiene el disponer de un algoritmo cuyo tiempo de ejecución sea razonable. De manera general, utilizaremos un método heurístico cuando se nos presente alguna de las siguientes afirmaciones:

* Se desconoce de un método exacto para la solución del problema
* El método exacto de resolución del problema presenta un coste computacional demasiado costoso o su obtención tarda demasiado.
* El método heurístico nos permite modelizar o incorporar condiciones imposibles en el método exacto.
* El método exacto es demasiado complejo como para ser implementado
* Se pretende encontrar una solución inicial razonable para posteriormente continuar con el método exacto, por lo que dicha solución será obtenida mediante un heurístico.

El término “heurística” proviene del griego, en concreto de ευρισκειν, heuriskein cuyo significado puede ser traducido como encontrar, hallar o descubrir.

La definición varía en función del contexto de aplicación, en términos científicos una de las más sencillas es la dada por Zanakis (331 libro Abraham): *“Procedimientos simples a menudo basados en el sentido común, que se supone que obtendrán una buena solución (no necesariamente óptima) a problemas difíciles de un modo sencillo y rápido”*

En la Diccionario de la Real Academia Española encontramos dos acepciones relacionadas con nuestro ámbito:

* Técnica de la indagación y del descubrimiento
* En algunas ciencias, manera de buscar la solución de un problema mediante métodos no rigurosos, como por tanteo, reglas empíricas, etc.

El principal defecto de las técnicas heurísticas es su tendencia a quedar estancado en óptimos locales. Esto es debido a que sencillamente carecen de un mecanismo para escapar de ellos o reconocerlos, por lo que, al encontrar un óptimo local, el algoritmo creerá que es el óptimo global del problema y terminará con la búsqueda.

Debido a este defecto surgen las llamadas metaheurísticas, que son métodos más globales, que guían a algoritmos heurísticos ya conocidos aportando los mecanismos necesarios para evitar caer en óptimos locales y acercarse así al óptimo real.

### Técnicas metaheurísticas

En el año 1986 F. Glover uso el término metaheurística para referirse a “*un procedimiento maestro de alto nivel que guía y modifica otras heurísticas para explorar soluciones más allá de la simple optimalidad local*”

El objetivo de las metaheurísticas es hacer la exploración del espacio de soluciones más eficiente y efectiva. Entre los algoritmos metaheurísticos más conocidos están las colonias de hormigas, algoritmos evolutivos, GRASP, búsqueda tabú…

Las características que pueden definir a las metaheurísticas son las siguientes:

* Son estrategias que “guían” la búsqueda.
* Su objetivo es explorar el espacio eficientemente para encontrar soluciones lo más optimas posibles.
* Desconocen si llegan a una solución óptima, por lo que hay que implementar condiciones de parada
* No es seguro la obtención de la solución óptima.
* Durante la exploración, se pueden producir un deterioro de la solución, llegando incluso a aceptar soluciones no factibles para explorar otras regiones del espacio de soluciones.
* Necesitan un campo de soluciones adecuado, una solución inicial (o varias) y un mecanismo de exploración de ese campo de soluciones.
* Se pueden aplicar a la mayoría de los problemas

Cuando hablamos de metaheurísticas debemos de tener dos conceptos muy claros: **diversificación** e **intensificación**. Estos conceptos hacen referencia a dos características de la exploración de nuestro espacio de búsqueda. Diversificación se refiere a la evaluación de posibles soluciones en zonas distantes del espacio de búsqueda. Por otro lado, intensificación se refiere a la búsqueda de soluciones en zonas acotadas y pequeñas dada una solución concreta.

Ambos conceptos deben de tener un equilibrio entre ellos, ya que es importante explorar una gran parte de nuestro espacio de soluciones, pero a la vez, solo invirtiendo el tiempo en aquellas zonas que puedan contener soluciones de alta calidad.

En definitiva, son estrategias que sirven para abordar problemas con un espacio de soluciones de gran tamaño mediante diferentes métodos.

La clasificación de metaheurísticas se puede dar en función de diversas características: basadas en la naturaleza o no, con o sin memoria, con una o varias estructuras de vecindario… Pero la manera más usada es dividirlas en metaheurísticas basadas en trayectoria (en cada iteración solo se modifica un elemento del espacio de búsqueda) o basadas en población (en cada iteración se modifica un conjunto de elementos).

En nuestro caso, el algoritmo elegido ha sido un GRASP, es decir una metaheurística basada en trayectoria.

### La elección: GRASP (Greedy Randomized Adaptive Search Procedure)

Pag 127

En español, procedimiento de búsqueda “miope” aleatorizado y adaptativo.

Como ya sabemos, es una metaheurística avanzada, cuyo procedimiento de búsqueda es miope, avaricioso o greedy, además de aleatorio y adaptativo. Puede ser aplicado sobre un gran conjunto de problemas de optimización garantizando una buena solución (aunque no necesariamente óptima).

Combina heurísticos constructivos con búsqueda local y consta de dos fases, la primera genera una solución, y la segunda trata de mejorarla (optimización local).

La construcción de la solución se realiza de manera aleatoria, introduciendo los componentes necesarios a la solución, inicialmente vacía. A continuación, se aplica un método de búsqueda local para mejorar la solución, que dependiendo de la naturaleza del problema será una técnica de mejora u otra, como pueden ser Tabu Search, Simulated Annealing, Local Search FI o LI, o simplemente un método aleatorio. Estas funciones por lo general medirán la mejora que se produce al realizar un cambio y se quedará con la mejor. Esta mejora se calcula solo para la iteración actual, sin tener en cuenta lo prometedor que sea más adelante. Así, las fases de un algoritmo GRASP son las siguientes:

1. Construcción: Crear aleatoriamente una solución factible al problema. Si la solución consta de varios componentes construir cada componente en cada iteración o seleccionarlo de una lista predefinida.
2. Mejora: Utilizar un método de búsqueda local utilizando la solución anterior como partida hasta que no se pueda mejorar más, se llegue a un error mínimo, o un límite de iteraciones
3. Actualización: En caso de que la última solución sea mejor que la que haya almacenada, actualizar la mejor solución

O en pseudocódigo:



Como anteriormente se expuso en la introducción, la función que optimizaremos mediante Grasp es:

Para tener un resultado consistente, debemos de crear varias instancias Grasp de tal manera que nos aseguremos que estamos explorando la mayor parte del espacio de soluciones. Cada una de esas instancias Grasp tendrá como objetivo generar y optimizar un número determinado de soluciones, asociado a la variable ***leaves***. A su vez, la variable ***branches*** controlará el número de instancias que se crearán.

Los ***métodos de optimización*** implementados han sido *Randomized*, *Line Search First Improvement* y *Line Search Best Improvement*. Para los métodos de tipo Line Search es necesaria una variable más llamada ***parts***que definirá las porciones en las que subdividiremos las variables alfa, beta y épsilon. La inclusión de esta última variable está relacionada con la necesidad de reducir las variables de un conjunto infinito ( en caso de épsilon) a un conjunto finito que pueda ser explorado en tiempos razonables.

Es obvio que debido a la naturaleza de los parámetros branches, leaves y parts un número más elevado en cualquiera de ellos produce una exploración más exhaustiva del espacio de soluciones y por tanto estaremos obteniendo resultados mejores de manera teórica.

Las variables más importantes en este proceso son ***alpha***, ***beta*** y ***épsilon***. En su caso, el ajuste será generado automáticamente por los optimizadores desarrollados. Así, el buen ajuste de estos parámetros dependerá del optimizador elegido.

Más adelante se debatirá cual ha sido la configuración óptima, sus motivos y los experimentos llevados a cabo para llegar a ella.

# DESCRIPCIÓN INFORMÁTICA (<15 págs.)

Todos los experimentos se han realizado en un ordenador de sobremesa sobre Windows 10, con 16GB de RAM DDR3 y un AMD FX-8350 4.0GHz como unidad de procesamiento.

Estos datos son meramente informativos ya que el objetivo del proyecto es el desarrollo de un algoritmo que resuelva el problema planteado al inicio. Cualquier cambio en las características del dispositivo que ejecute el código simplemente repercutiría en el tiempo de ejecución y en ningún caso en la efectividad del algoritmo.

A continuación, se presentan los lenguajes, programas, librerías, herramientas, metodología y requisitos **a satisfacer**

## Lenguajes y programas.

En primer lugar, comenzaremos explicando los lenguajes que se han utilizado: Python y Java, aunque el primero fue usado solo en la parte inicial del proyecto.

La elección de estos dos lenguajes de alto nivel es debido a la naturaleza del proyecto ya que estos lenguajes presentaban librerías que nos resultaron atractivas y con potencial para ayudarnos a resolver el problema de las cuales hablaremos más adelante.

### Python



Es un lenguaje nacido a finales de los años 80 de la mano de Guido van Rossum, un holandés que desarrolló este lenguaje de programación para el Centro para las Matemáticas y la Informática de los Países Bajos con la intención de sustituir al lenguaje ABC, que a su vez surgió como alternativa a BASIC.

Python es un lenguaje interpretado, de alto nivel, de propósito general, multiplataforma, open-source, multitarea, con compatibilidad con una gran cantidad de paradigmas, tipado dinámico y sintaxis indentada, y también destaca por la gran cantidad de librerías tanto built-in como mediante descarga.

Este lenguaje se utilizó en los primeros meses, junto a la librería PyBrain (punto 4.2.1), pero debido a la poca documentación y al aparente estado de abandono en el que se encontraba la librería se decidió hacer un cambio de lenguaje y librerías a Java y Neuroph.

### Java



Se creó a principios de los 90 y fue desarrollado por James Gosling (Sun Microsystems). El objetivo era crear un lenguaje independiente de la plataforma y un entorno (JVM) ligero y gratuito para que las aplicaciones se pudieran ejecutar en la mayor parte de plataformas.

Java es un lenguaje de alto nivel, de propósito general, multiplataforma, open-source, multitarea, con compatibilidad con una gran cantidad de paradigmas, y al igual que Python cuenta con gran cantidad de librerías.

Las diferencias que presenta con respecto a Python son que tiene tipado estático, sintaxis no indentada, es un lenguaje compilado e interpretado (el código fuente se traduce a bytecode para posteriormente ser interpretado por la máquina virtual de Java) y es un lenguaje dinámico (las clases compiladas pueden ser cargadas bajo demanda en tiempo de ejecución).

Al igual que con los lenguajes, los IDE’s también han sido diversos. Al estar usando Python al comienzo del proyecto, el IDE que elegimos para este lenguaje fue PyCharm.

Con el cambio de lenguaje se produjo un cambio en los IDE’s. En el caso de Java primero se utilizó NetBeans y para finalmente acabar usando la plataforma Eclipse.

### https://www.fourdigits.nl/images/pycharm-logo.gifPyCharm

Es un entorno de desarrollo integrado (IDE) específico para el lenguaje Python desarrollado por la empresa JetBrains en 2010 (también creadora de otros famosos IDE’s como WebStorm, RubyMine…). Es multiplataforma con versiones para Linux, Windows y Mac.

Entre sus características podemos encontrar: Refactorizaciones automáticas, soporte para diferentes frameworks, debugger, unidad de testeo integrada, integración de control de versiones, navegador de proyecto con vistas y estructuras especializadas y análisis y asistencia de código con autocompletado, subrayado de errores y arreglos de diversos problemas.

### Resultado de imagen de eclipse software logoResultado de imagen de netbeans logoNetBeans y Eclipse

Dos de los IDE más conocidos hoy en día. NetBeans, desarrollado por Oracle Corporation a partir del año 2000.

Eclipse, desarrollado por IBM a partir del año 2004.

Ambos disponen de versiones para Windows, Linux y Mac, y son compatibles con desarrollos en Java, JavaScript, C, C++, desarrollo web y más.

Sus características son las mismas que cualquier otro IDE como PyCharm. Soporte para refactorizaciones automáticas, debugger, integración de control de versiones, autocompletado, etc...

## Librerías y herramientas

A continuación, se exponen las librerías de redes neuronales que se han usado en el proyecto, así como herramientas secundarias para la gestión del mismo:



### PyBrain y Neuroph

PyBrain es una librería para Python. Su objetivo es ofrecer algoritmos Machine Learning potentes, flexibles y fáciles de usar. Su nombre proviene de las siglas **Py**thon-**B**ased **R**einforcement Learning, **A**rtificial **I**ntelligence and **N**eural Network Library. Su código es abierto y de uso gratuito para cualquiera ([Licencia BSD](https://es.wikipedia.org/wiki/Licencia_BSD#Licencia_BSD_modificada_.28de_3_cl.C3.A1usulas.29)), pero por desgracia los últimos cambios en su [repositorio](https://github.com/pybrain/pybrain) datan de hace varios años, por lo que podemos considerar que ya se encuentra muy desactualizada y podemos encontrar otras opciones mejores en librerías como TensorFlow, Blocks, Deepy, Neupy, etc.

En el caso de Neuroph, hablamos de una librería Java que también cuenta con un IDE completo basado en NetBeans que añade una interfaz gráfica intuitiva a la hora de programar, la cual no se ha utilizado en este proyecto.  
Está orientado al desarrollo de redes neuronales comunes, con un reducido número de clases asociadas a los conceptos básicos.  
Al igual que PyBrain, es Open Source y gratuito, bajo licencia [Apache 2.0](http://www.apache.org/licenses/LICENSE-2.0.html). Su [repositorio](https://github.com/neuroph/neuroph) se encuentra activo desde mediados de 2015.

Ambas están orientadas al trabajo con redes neuronales, desde la creación de conjuntos de entrenamiento, configuración de redes, entrenamiento etc. Útiles para resolver cualquier tipo de problema que pueda ser planteado desde un enfoque de redes neuronales.

Otras herramientas que se han utilizado en el proyecto han sido Maven, Git, ObjectAid y SonarQube.

### https://maven.apache.org/images/maven-logo-black-on-white.pngMaven

Es una herramienta Open Source, creada por Jason van Zyl, de Sonatype en el año 2002. Es similar a las herramientas Apache Ant, PEAR (php) y CPAN (Perl)

Su objetivo es simplificar la gestión de un proyecto software de tal manera que un desarrollador pueda extraerse de ciertos procesos con la ganancia de tiempo que esto conlleva. Con Maven la build de un proyecto se basa en tener un fichero pom.xml donde tengamos definida la configuración de nuestro proyecto con sus módulos, dependencias, librerías, etc. De esta manera con ejecutar el comando mvn install, Maven se encargará de leer el pom y hacer las tareas y configuraciones que se hayan definido.

Aunque en realidad, Maven es capaz de gestionar completamente el ciclo de un software ya que puede gestionar Validación, Compilación, Tests unitarios, Empaquetado, Pruebas de Integración, Verificado, Instalación y Despliegue de nuestro proyecto.

Como añadido, cuenta con un repositorio en internet llamado Maven Central. En él se encuentra una colección de librerías asociadas a sus posibles dependencias, de tal manera que con definir en el pom.xml las librerías que necesita nuestro proyecto Maven accederá al almacén central y nos descargará automáticamente todo lo que necesitemos (incluyendo todas las librerías que se necesiten a niveles más bajos).

### https://upload.wikimedia.org/wikipedia/commons/thumb/e/e0/Git-logo.svg/1200px-Git-logo.svg.pngGit (GitHub)

En el año 2002, el proyecto del núcleo de Linux empezó a usar un software de control de versiones llamado “BitKeeper” de manera gratuita. Tres años más tarde (2005) la compañía propietaria decidió dejar de ofrecer los servicios de manera gratuita a la comunidad Linux. Esto propició que el propio Linus Torvalds comenzara a desarrollar esta herramienta gratuita de control de versiones: Git.

Este tipo de programas se centran en la gestión de los cambios que se producen en un proyecto a lo largo del tiempo, de tal manera que todos esos cambios se guarden y se organicen en versiones incrementales. Otros VCS (versión control system) son SVN. Mercurial, CVS…

Gracias a esto tenemos distintas snapshots de nuestro código, que siempre pueden ser revisadas, y pueden servir para revertir todos los cambios hasta ese punto si en algún momento se ha producido un fallo y se quiere volver al estado de un commit concreto.

### Resultado de imagen de objectaidObjectAid

Es un plugin de Eclipse encargado de generar los diagramas UML a partir de nuestro código fuente. La creación de las cajas y enlaces es casi automática, y si por algún motivo se realiza algún cambio en el código después de la creación, dichos cambios se verán reflejados automáticamente en el diagrama. Como contraprestación, la disposición automática del diagrama es realmente mala, y hay que perder bastante tiempo en recolocar cada una de nuestras clases para facilitar la comprensión del mismo.

### SonarQube

Como añadido, se ha utilizado la herramienta SonarQube para analizar la calidad del código, detectar posibles defectos en el mismo, y poder así subsanarlos.

Tras realizar varios escaneos de calidad a nuestro proyecto pasamos de tener 10 bugs, 15 vulnerabilidades 218 “*code smells*” y algún porcentaje de código duplicado. A solo 23 “*code smells*” que son derivados de la complejidad de algunas funciones o clases:



Primer análisis de código Último análisis

## Metodología de desarrollo software

El término metodología se puede definir como el conjunto de métodos que se emplean para lograr un objetivo.

A su vez, un método se compone de herramientas, técnicas y procedimientos.

Así, podemos decir que una metodología de desarrollo de software define el conjunto de herramientas, procedimientos y técnicas que se emplean con el objetivo de construir el software deseado, o, dicho de otra manera, es el conjunto de las estrategias de desarrollo que nos ayudan a organizar los procesos y actividades dentro del ciclo de vida del software definiendo así el marco de trabajo que se va a utilizar durante el desarrollo de un proyecto. Entre todos los modelos de desarrollo podemos destacar el modelo en Cascada, el modelo Incremental y los modelos Ágiles.

El modelo en cascada implica carecer de cualquier tipo de iteración, sus etapas son secuenciales y no repetibles, por lo que una etapa solo comienza cuando ha acabado la anterior, esto es muy útil para procesos muy cerrados.

El modelo incremental consiste en repetir las fases del modelo en cascada. Al tener varias iteraciones, permite añadir nuevas funcionalidades, requisitos o especificaciones, creando al final de cada ciclo una evolución del software.

Los modelos ágiles tienen como objetivo desarrollar software rápidamente, de calidad y con una gran capacidad de respuesta frente a cambios o imprevistos en el proyecto. Se enfocan principalmente en la gente y en sus resultados. En el año 2001 se creó el “Manifiesto Ágil” que consta de los valores de valorar a los individuos más que a las herramientas, valorar más el software funcional sobre la documentación exhaustiva, valorar más la colaboración con el cliente que los contratos contractuales y valorar más la respuesta a cambios que tener planes concretos e inmutables.

Estos son los modelos más relevantes a día de hoy, pero hay otras muy importantes como pueden ser las de prototipado, desarrollo en espiral o Rapid Application Development (RAD)

### La elección: El método incremental.

Si bien hoy en día el mundo de desarrollo software está siendo bombardeado por numerosos métodos de programación ágiles como Scrum o XP (eXtreme Programming), se ha considerado que la metodología que más encajaba con este proyecto era el modelo incremental.

Los principales argumentos que se han tenido en cuenta para su elección han sido los siguientes:

* Descarte total del modelo en cascada debido a la certeza de incertidumbre que iba a presentar el proyecto (previsibles cambios en requisitos o funcionalidades, tendencia al cambio en general)
* Preferencia sobre los modelos ágiles debido a:
  + Los modelos ágiles están pensados en la interacción con numerosas personas en un equipo de desarrollo, en este caso solo hay 1 persona.
  + Poca certeza de tener la capacidad de tener todas las reuniones necesarias para implementar estas metodologías.
  + Poco tiempo máximo entre un sprint y otro, que unido a la poca disponibilidad en ese momento del autor del proyecto hace que sea inviable este modelo
* Viabilidad del modelo incremental debido a:
  + Existencia de iteraciones en el desarrollo que van aumentando el valor del producto. Desarrollo modular.
  + Una vez finalizada una iteración se analiza un plan para el desarrollo de la siguiente iteración en contraposición con requisitos fijos del modelo en cascada. Esto favorece una mejor gestión de riesgos.
  + Flexibilidad del modelo incremental en cuanto a tiempos de desarrollo de cada iteración
  + Facilidad de implementar cambios en el proyecto en función del tamaño de los incrementos

El esquema general del modelo incremental es el siguiente:

Requisitos

Diseño

Implementación

Verificación

## Estructura del software

En cuanto a la organización interna del proyecto, cada clase se ha diseñado dentro de un paquete que engloba su objetivo. Así, los paquetes existentes son los siguientes:



* Global: Contiene lo relacionado con variables globales **Problem** y **YearInfo**
* GUI: Clases asociadas a la interfaz gráfica de la aplicación **MainWindow**, **DefaultTab**, **MetaGui** y **NeuralGui**
* Metaheuristic: Clases asociadas a la resolución de la parte de metaheurísticas **MetaSolver** y **MetaSearch**.
  + Models: Clases de los modelos asociados al paquete **MetaResults**, **MetaSolution** y **MetaVariable**.
* NeuralNetwork: Clases asociadas a la resolución de la parte de redes neuronales **NeurophSolver** y **NeurophSearch**.
  + Charts: Clases de los modelos asociados con los datos y visualización de gráficos **ChartData** y **LineChartSample**.
* Util: Clases generales con diversos usos **CSVTableWriter** y **Normalizer**
  + Optimizers: Al tener una estructura más compleja y tener varias clases disponemos de un paquete interno para las clases **Optimizer**, **EvaluationOptimizer**, **RandomEvaluationOptimizer**, **LSEvaluationOptimizer**, **LSFIEvaluationOptimizer** y **LSBIEvaluationOptimizer**.

Las siglas UML vienen del inglés Unified Modeling Language (lenguaje unificado de modelado). Es un lenguaje gráfico que sirve para visualizar sistemas, procesos, bases de datos…

En este caso lo utilizamos para representar las clases que tiene nuestro proyecto y todas sus relaciones:

En el diagrama hemos podido ver todas las clases de nuestro proyecto, pero para facilitar el entendimiento se explicará cada una de sus partes por separado:

### C:\Users\Cesar\AppData\Local\Microsoft\Windows\INetCache\Content.Word\1 Gui Resaltado.jpgInterfaz Gráfica

En cuanto a la interfaz gráfica, disponemos de una ventana principal, correspondiente a la clase **MainWindow**, que funciona como punto de entrada de nuestro programa, con la función main. Es un JPanel con pestañas. Cada una de esas pestañas dispondrá de una clase propia donde se encontrará su diseño. Al tener un esquema compartido se crea la clase **DefaultTab**, de la que heredarán ambas.

Las clases **MetaGui** y **NeuralGui** contienen el diseño interno que se mostrará al pulsar cada pestaña.

***En el caso de MetaGui, el diseño final es éste:***



***El diseño final de NeuralGui:***



Además, también están encargadas de lanzar el procedimiento que surge tras pulsar un botón, incluyendo todo el proceso de recogida de datos y validación de los mismos, seguido del posterior lanzamiento de la búsqueda por parte de la clase MetaSolver o NeurophSolver, que serán explicados detalladamente más adelante. Es importante que estos procesos sean lanzados en segundo plano para permitir que la interfaz gráfica continúe actualizándose

### Global

Estas clases están encargadas de facilitar el acceso a los datos globales del problema.

**YearInfo** es la encargada de guardar cada una de las 14 variables macroeconómicas para cada año.

**Normalizer** estará encargada de realizar todas las labores de normalización y desnormalización de los datos.

**Problem** mediante un *HashMap* contendrá tantos **YearInfo** como años de datos dispongamos, aparte de una instancia de **Normalizer**. Esta clase está diseñada en forma de *Singleton*, de esta forma cada vez que se quiera acceder a los datos o a alguna ruta concreta del problema siempre se accederá al mismo objeto y no estaremos creando objetos innecesarios.

### Redes Neuronales

Dentro de las clases relacionadas con redes neuronales podemos encontrar 3 grupos:

* **Clases relacionadas con la creación de gráficos**

**ChartData** contiene la información correspondiente a los errores que se han ido obteniendo a lo largo de un entrenamiento y la configuración del mismo.

**LineChartSample** genera un *Thread* que se encarga de crear una ventana y pintar tantas líneas como ChartData tenga de entrada, así como guardar dicho gráfico para el posterior análisis de resultados previo al cierre.

* **Clases externas**

**Problem** es utilizada para acceder a parámetros globales y estáticos, como pueden ser las rutas que contienen los datos y conjuntos de entrenamiento y test, o los propios datos accesibles a través de la propia clase.

**CSVTableWriter** es una clase que se utiliza de manera global en el proyecto cada vez que se quiere generar un archivo Excel con formato csv. En este caso, se utiliza para guardar numéricamente los datos que previamente se han mostrado gráficamente a través de LineChartSample.

* **Clases *“importantes”***:
  + **NeurophSolver:** Cuenta con diversos procedimientos en función de la tarea a desarrollar.
    - Método simple: Que solo admite una combinación y genera un gráfico a partir de ella
    - Método avanzado: Que admite parámetros en forma de listas y posteriormente entrena cada posibilidad, generando así **n** gráficos.
    - Método de elección de red: Visita todos los archivos de guardado de redes y calcula cual es el mejor de todos.
    - Método de testeo de red: Dado un archivo de red, genera un archivo csv en el cuál se visualizan los datos reales y calculados correspondientes a todos los años de los que disponemos.
    - Ir guardando el archivo csv
  + **NeurophSearch:** Es la clase que conecta directamente con la librería de Neuroph. Está encargada de:
    - Realizar las labores de creación y entrenamiento de redes
    - Creación de gráficos
    - Cálculo y guardado de MSE al finalizar cada epoch
    - Salvado de redes una vez han finalizado el entrenamiento.

### Metaheurísticas

En el caso de las metaheurística podemos encontrar 4 grupos:

* **Modelos**
  + MetaVariable: Modelo con los parámetros alfa y beta de cada parámetro.
  + MetaSolution: Guarda los valores de MetaVariable para cada uno de nuestros parámetros; el valor de épsilon, la evaluación y el tiempo de ejecución
  + MetaResults: Modelo con el que definimos el resultado de la Metaheurística: una MetaSolution, el tiempo total secuencial y concurrente, así como error y tiempo medios.
* **Clases externas**
  + **Problem y CSVTableWriter** tienen el mismo propósito que en la parte neuronal.
* **Clase Optimizador**

Para poder realizar las mejoras sobre nuestra solución mediante nuestro método Grasp, necesitamos un evolutivo. En este caso, la interfaz **Optimizer** será la encargada de proporcionarnos acceso a los métodos de evaluación y optimización. La estructura de optimizadores se verá a continuación

* **Clases “importantes”**
  + **MetaSolver**: Encargada de las siguientes tareas:
    - Generar todas las instancias de MetaSearch en un nuevo Thread.
    - Recoger todos los resultados que reporte cada MetaSearch
    - Recorrer todos los resultados para decidir cuál es el mejor
    - Generar y guardar el archivo csv
  + **MetaSearch**: Recordamos que un Grasp requiere un método constructivo y uno evolutivo. En esta clase tenemos ambos:
    - Construye tantas posibles soluciones como se requieran en base a un método aleatorio.
    - Posteriormente utilizando el **Optimizer** que le haya sido asignado optimiza cada una de las soluciones y las devuelve.

### Optimizadores de metaheurísticas

* **Interfaz Optimizer**

Cuenta con las funciones optimize y evaluate, que serán desarrolladas en las clases que la implementen.

* **Clase Abstracta EvaluationOptimizer**

Esta clase implementa la anterior, pero al ser abstracta no necesita definir todas las funciones de su interfaz, así, solo implementa la función evaluate, que será global para todas las subclases que vengan detrás, deja abstracta la función optimize que será específica y define los primeros atributos de clase y funciones auxiliares

* **Clase Concreta RandomEvaluationOptimizer**

Esta es la primera clase no abstracta, y es el optimizador más sencillo con el que contamos. Como su nombre indica, genera cambios al azar sobre las variables presentes en la instancia de MetaSolution. Simplemente extiende la clase EvaluationOptimizer y termina de implementar la función Optimize.

* **Clase Abstracta LSEvaluationOptimizer**

Esta clase se bifurca de la anterior, redefiniendo el comportamiento del constructor y otra función que será usada por sus subclases. De nuevo, es otra clase abstracta, por lo que no puede ser instanciada.

* **Clases Concretas LSBIEvaluationOptimizer y LSFIEvaluationOptimizer**

Ambas heredan directamente de LSEvaluationOptimizer e implementan la función optimize de acuerdo a la implementación de búsquedas locales first improvement y best improvement.

Diagramas de clases. Describir a nivel de clase relevante los métodos más importantes, (mirar en otros proyectos). Figura, explicar un poco y tal, que ilustre la cantidad de código que he elaborado, como se integran las librerías. A nivel de bloque el nivel de código que se ha desarrollado (El código se sube al aula, aquí no hay que explicar nada de código, solo ver que se ha hecho), quizás algo de pseudocódigo de la búsqueda o algo concreto, que sea específico y relevante.34:00

# EXPERIMENTOS (<10 págs.)

## Redes Neuronales

Para entrenar una red neuronal satisfactoriamente hay varios parámetros que hay que ajustar para que los resultados sean los deseados. Esos parámetros son los siguientes:

### Tipo de propagación

Disponemos de dos tipos de retropropagación, BackPropagation y ResilientPropagation.

En BackPropagation disponemos de un LearningRate global que afecta a todas las neuronas, que tendrá que ser ajustado antes de realizar el entrenamiento.

En Resilient cada neurona posee un LearningRate propio, que se irá ajustando de manera automática con las necesidades del entrenamiento. Estos learningRates no se pueden configurar, ya que se manejan de manera interna y completamente automática.

En principio se puede afirmar que ResilientPropagation es un algoritmo de retropropagación más complejo y debería dar mejores resultados que BackPropagation, aun así, en los experimentos debemos estudiar ambas configuraciones, ya que a priori desconocemos cuál de las dos encaja mejor con la naturaleza de nuestro problema.

### Learning Rate (LR)

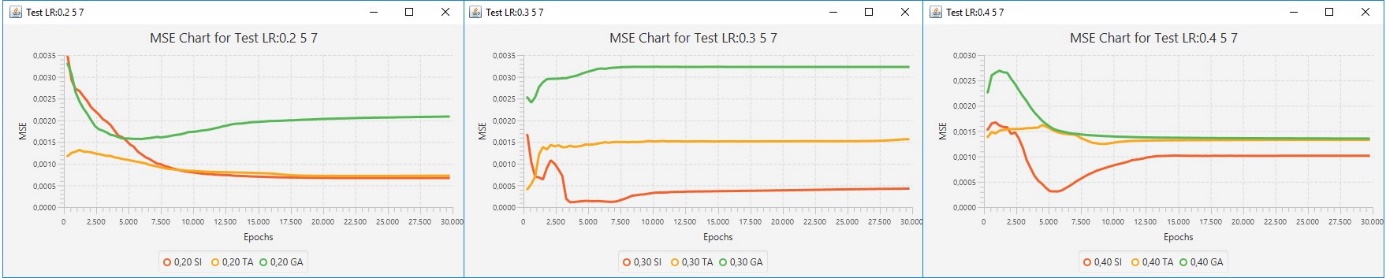
De manera arbitraria se han establecido 4 valores para las pruebas: 0.001, 0.01, 0.1 y 0.3. Así se pretende estudiar un abanico extenso de este parámetro, permitiendo así que la configuración de la red resultante sea lo más óptima posible, dando 4 opciones a cada disposición de parámetros. Estos valores solo se aplicarán a las redes con retropropagación de tipo BackPropagation.

En las redes con ResilientPropagation al no poder establecer un LearningRate, cada prueba será guiada por sí misma. Esto implica que cada uno de los LR de cada neurona serán generados aleatoriamente antes de iniciar el entrenamiento.

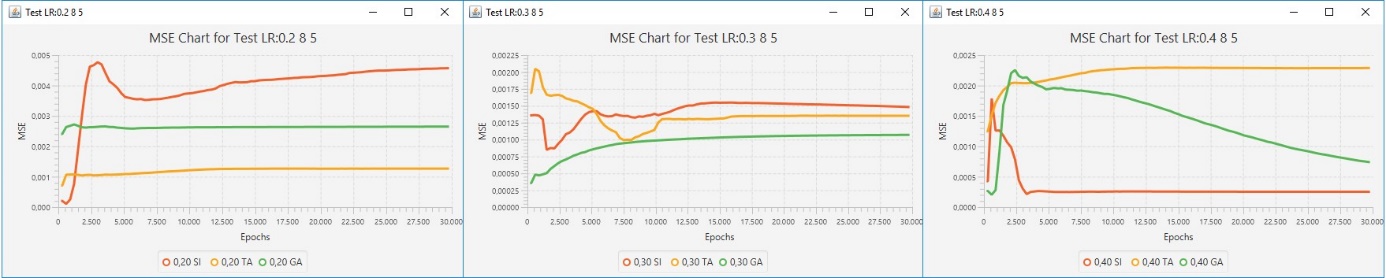
Debido a la gran cantidad de parámetros que se inician de manera aleatoria, se establece que para cada configuración de red con ResilientPropagation se crearán y entrenarán 3 redes distintas para dar opciones a que la red inicial sea suficientemente buena para que el entrenamiento de un resultado aceptable.

### Número de iteraciones máximo en el proceso de entrenamiento

Para este parámetro se realizaron varios experimentos con diferentes configuraciones. Se estableció de manera arbitraria un límite de 30.000 para observar la evolución de los entrenamientos y valorar la posibilidad de bajar aún más el valor.



Éstas solo son un conjunto reducido de todas las pruebas en las que en términos generales se pudo observar que la mayoría de las redes tardaban pocos miles de iteraciones en estabilizarse.

En la red verde podemos observar un ejemplo de entrenamiento prolongado a lo largo de un gran número de iteraciones. Estos casos pueden ocurrir, pero para no limitar la capacidad de los experimentos se decidió establecer el límite en 15.000 iteraciones, dado que en ese momento consideramos que la mayoría de las redes se habrán estabilizado, y debido a las limitadas capacidades computacionales de las que dispone el estudio se priorizará ampliar el análisis de un mayor número de redes sobre llegar al estado óptimo de cada una de ellas.

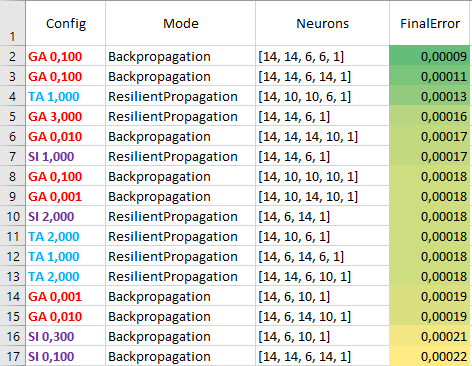
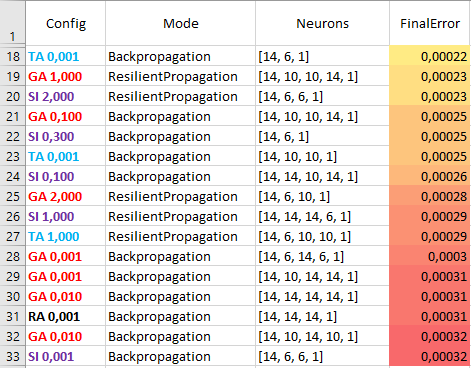
### Función de transferencia de las neuronas

En la librería Neuroph se encuentran implementadas las siguientes funciones de transferencia: Gaussian, Log, Linear, RectifiedLinear, Ramp, Sgn, Sigmoid, Sin y Tanh. El segundo experimento que se realiza consta de generar varias redes neuronales para cada una de las funciones de transferencia.

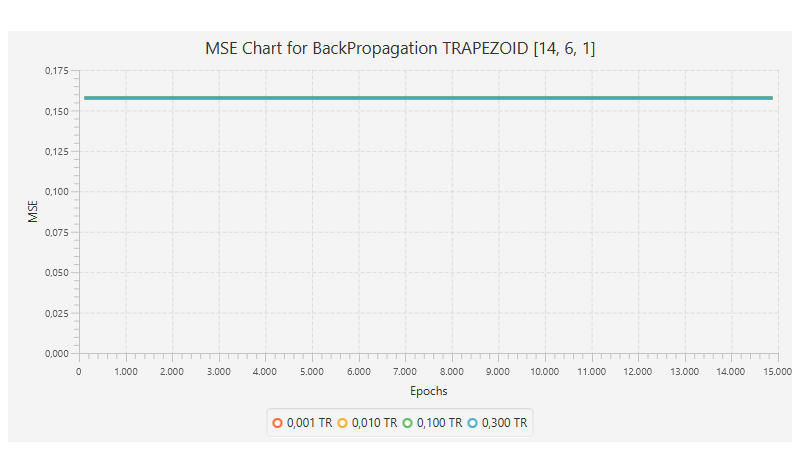
En estas pruebas se pretende analizar el impacto que tiene cada función en la red, y generar un conjunto de funciones prometedoras. La configuración de la prueba será la siguiente:

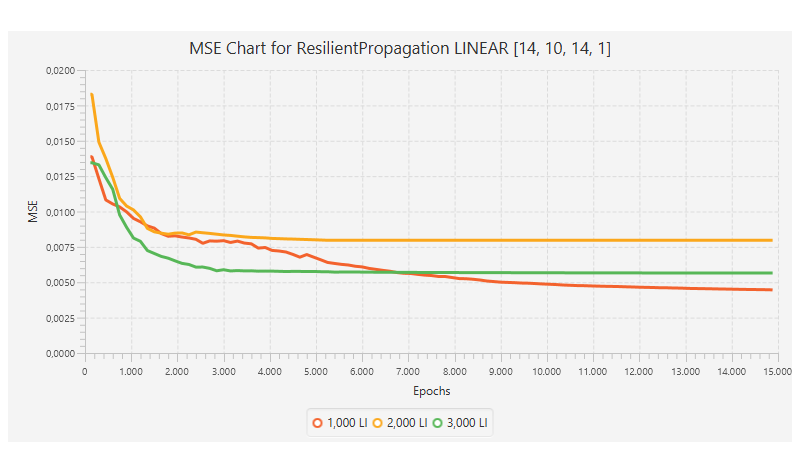
* Establecer las normas establecidas anteriormente
  + Analizar ambos tipos de retropropagación
  + Para BackPropagation analizar los 4 LR del apartado anterior
  + Para ResilientPropagation analizar 3 redes distintas para cada configuración
* Analizar redes de entre 1 y 3 capas ocultas
* En cada capa oculta analizar todas las posibles combinaciones de 6, 10 o 14 neuronas.

Al finalizar este experimento obtuvimos 2.653 redes. Las 32 redes más prometedoras fueron las siguientes:

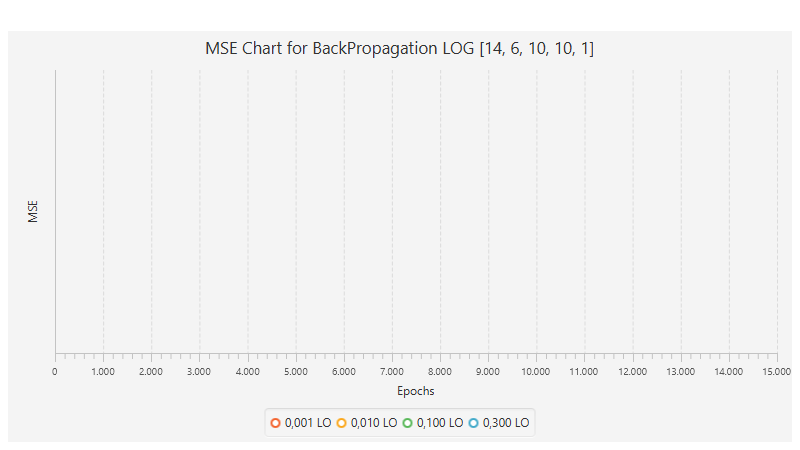
Como se puede observar en la tabla, las primeras posiciones están copadas por las funciones de transferencia Gaussian, Tanh y Sin. El resto de funciones se descartan por no presentar una precisión tan elevada o por ser incompatibles con nuestro experimento.

Para cada configuración de función de transferencia y, capas y número de neuronas se genera un gráfico con diversas líneas correspondientes a cada red. Los siguientes gráficos tienen como objetivo ejemplificar los resultados obtenidos:

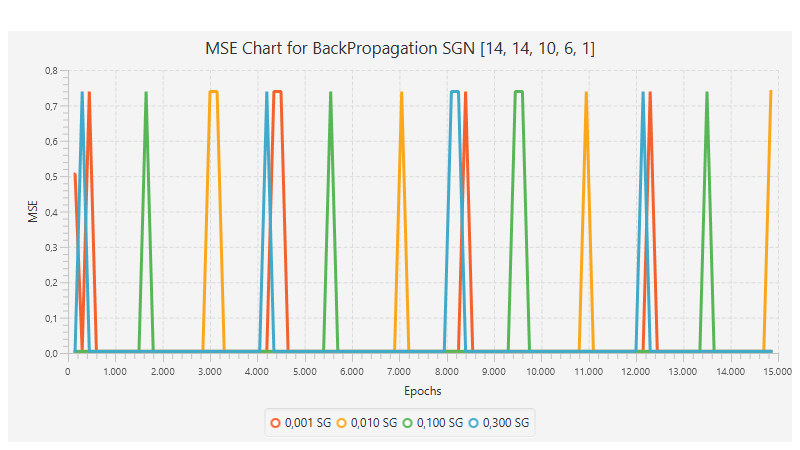
En el caso de la función Trapezoid todos los entrenamientos presentan la misma forma, y con el mismo error, el entrenamiento no presenta ninguna mejoría. Indiferente comportamiento entre Resilient y Backpropagation.

La función Linear presenta numerosos entrenamientos consistentes y se puede observar como reduce de manera continua el error, pero el resultado se queda lejos de otras funciones.

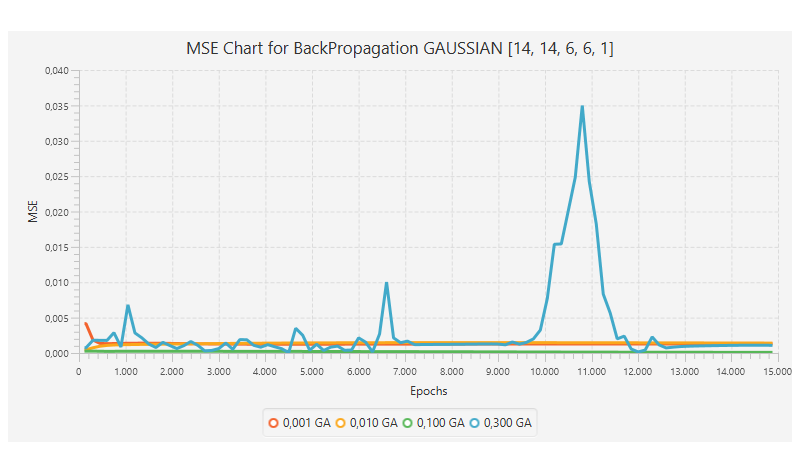
Similar comportamiento entre Resilient y Backpropagation.



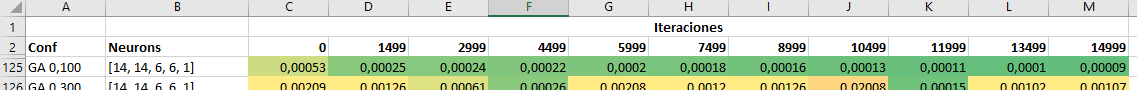
La función Logaritmo no genera ningún tipo de resultado, indiferentemente del tipo de retropropagación o de la configuración de número de capas o número de neuronas.



Las funciones Signal y Step presentan entrenamientos de características similares. Muy oscilantes, y aunque en el momento de finalizar el entrenamiento se encuentren en su valor óptimo los resultados quedan lejos de ser los mejores.

La función Gaussian es una de las mejores, en este gráfico se observa la red con un mejor resultado, correspondiente a la línea verde.

Debido al reescalado producido por la red azul se pierde el detalle, pero disponemos de los siguientes datos en los que podemos observar la progresión del entrenamiento:



### Número de capas ocultas y número de neuronas por capa

* Capas ocultas: en la siguiente tabla se muestra el comportamiento con **x, y, y z**. Como se puede ver en tal tabla lo mejor es tantas capaz ocultas
* Número de neuronas presentes en cada capa:

## Metaheurísticas

Las metaheurísticas también cuentan con parámetros que hay que ajustar, específicamente:

* Clase de optimizador: **Explicar**
* Número de hilos de búsqueda (ramas): **Explicar**
* Número de soluciones que va a optimizar cada hilo (hojas): **Explicar**
* Número de partes en las que vamos a dividir el rango de valores que pueden tomar las variables de nuestra MetaSolution: **Explicar**

Así, para realizar el estudio de todos estos parámetros **Explicar método de experimentación**

Ajuste y estudio de parámetros, como afectan sus valores…Tanto de la red (LR, Epochs), como de la meta (optimización elegida, iteraciones…) que cosas permiten elegir los mejores parámetros teniendo en cuenta los objetivos (si tarda más o menos o que)

TIRAR A PONER GRÁFICOS O FIGURAS, mucho mejor que una tabla.

Comparar con métodos previos (En la introducción, nombras gente que ha trabajado, es posible que sean comparables con esto de aquí)

# CONCLUSIONES Y TRABAJOS FUTUROS (2 págs.)

Lo de objetivos que iba en presente, ahora en pasado, que he logrado de todo.

En futuros, cosas mejorables o que probar. Interfaz gráfica, paralelización, redes neuronales Deep learning, otra metaheurística... O que habría hecho si hubiera tenido más tiempo.

## Especificación de requisitos

Nuestro software tiene que cumplir una serie de condiciones expresadas a modo de requisitos funcionales y no funcionales:

### Funcionales

Son una descripción de lo que debe o no debe hacer el sistema a bajo nivel, que servicios debe proporcional, cuestiones técnicas… “Qué” debe hacer

* Fasdf
* Asdf
* Asdfa
* Ssdf

### No funcionales

Especifica criterios para juzgar la operación de un sistema. “Cómo” debe hacerlo

* Fghj
* Fghj
* Fghj
* Fghj
* Fgh
* df