UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Sistemas de comunicaciones e interfaces para aplicaciones en la Industria 4.0

ANEXO

TRABAJO FINAL DE MASTER

Nombre: Ángel Fernández Sobrino
Titulación: Màster Universitari en Enginyeria Industrial
Director: Jose Luis Romeral Martínez
Codirector: Miguel Delgado Prieto
Convocatoria: Octubre 2020
Escola: Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Tabla de Contenidos

1 Software Simulador

1.1 Clases auxiliares simulador

```
import copy
import Utilities as Ut

class Bandeja:
    def __init__(self, identificador=-1, tipoProducto=-1):
        self.id = identificador
        self.tipo = tipoProducto
        self.finalizado = False
        self.materiaPrima = False

    def nuevoProducto(self, tipo):
        self.tipo = tipo

    def cargarMateriaPrima(self):
        self.materiaPrima = True

    def vaciar(self):
        self.tipo = -1
        self.finalizado = False
        self.materiaPrima = False

    def finalizar(self):
        self.finalizado = True
        self.materiaPrima = False


class Output:
    def __init__(self, tiposDiferentes, cantidadRetenedores):
        self.produccion = []
        self.tiempoReposo = []
        self.tiempoPeticion = []
        self.tiempoAvance = []
        self.tiempoProceso = []

        for i in range(0, tiposDiferentes):
            self.produccion += [0]

        for i in range(0, cantidadRetenedores):
            self.tiempoReposo += [0]
            self.tiempoPeticion += [0]
            self.tiempoAvance += [0]
            self.tiempoProceso += [0]

    def producir(self, tipo):
        self.produccion[tipo] += 1

    def incrementarTiempo(self, retenedor, elemento):
        if elemento == 'reposo':
            self.tiempoReposo[retenedor] += 1
```

```
            if elemento == 'peticion':
                self.tiempoPeticion[retenedor] += 1
            if elemento == 'avance':
                self.tiempoAvance[retenedor] += 1
            if elemento == 'proceso':
                self.tiempoProceso[retenedor] += 1

    def print(self):
        print('')
        Ut.pNoEs('Produccion: ')
        print(self.produccion)
        Ut.pNoEs('Reposo: ')
        print(self.tiempoReposo)
        Ut.pNoEs('Peticion: ')
        print(self.tiempoPeticion)
        Ut.pNoEs('Avance: ')
        print(self.tiempoAvance)
        Ut.pNoEs('Proceso: ')
        print(self.tiempoProceso)

    def exportar(self):
        return [self.produccion, self.tiempoReposo, self.tiempoProceso,
self.tiempoPeticion, self.tiempoAvance]


class Retenedor:

    def __init__(self, identificador, topologia, tiempoAvance,
cantidadSalidas):
        self.identificador = identificador
        self.tiempoAvance = tiempoAvance
        self.timerAvance = []
        self.reposo = True
        self.peticion = False
        self.avance = []
        self.cantidadSalidas = cantidadSalidas
        self.stop = []
        self.salidas = topologia[identificador]
        self.entradas = []
        self.sensor = False
        self.bandejaEntrada = Bandeja()
        self.bandejasSalida = []
        self.temporizador = 0
        self.tiempoEntrada = 0
        self.procesando = False

        for i in topologia[identificador]:
            self.avance += [False]
            self.stop += [False]
            self.timerAvance += [-1]
            self.bandejasSalida += [Bandeja()]

        for i in range(1, len(topologia)):
            if identificador in topologia[i]:
                self.entradas += [i]

    # print("Nodo: " + str(self.identificador) + " Salidas: " +
```

```
str(self.salidas) + " Entradas: " + str(self.entradas))

    def __ne__(self, retenedor):
        return retenedor.reposo != self.reposo or self.peticion !=
retenedor.peticion or self.avance != retenedor.avance

    def bloquear(self, salida):
        self.stop[self.salidas.index(salida)] = True
        self.temporizador = -1

    def desbloquear(self, salida):
        self.stop[self.salidas.index(salida)] = False
        self.temporizador = -1

    def bloquearTiempo(self, salida, tiempo):
        self.stop[self.salidas.index(salida)] = True
        self.temporizador = tiempo
        self.procesando = True

    def llegada(self, bandeja):
        if self.reposo:
            self.sensor = True
            self.bandejaEntrada = copy.deepcopy(bandeja)

    def estaAvanceA(self, destino):
        j = 0
        for i in self.salidas:
            if i == destino:
                if self.avance[j]: return True
            j += 1
        return False

    def estaOcupado(self, sistema):
        for i in self.entradas:
            if sistema[i].estaAvanceA(self.identificador): return True

        if self.reposo:
            return False
        else:
            return True

    def update(self, sistema, tiempo, log, resultado, actualizaEsperas):

        # if log: Ut.pNoEs(" R" + str(self.identificador))
        # if log: Ut.pNoEs("")
        # if log: Ut.pNoEs(int(self.reposo))
        # if log: Ut.pNoEs(int(self.peticion))
        # if log:
        #    for i in range(0, len(self.salidas)):
Ut.pNoEs(int(self.avance[i]))

        # if log: Ut.pNoEs(self.bandejaEntrada.id)
        # if log: Ut.pNoEs(self.bandejaEntrada.tipo)
        # if log: Ut.pNoEs(int(self.bandejaEntrada.materiaPrima))
        # if log: Ut.pNoEs(int(self.bandejaEntrada.finalizado))

        for i in range(0, len(self.salidas)):
```

```
            if self.stop[i]:
                if self.temporizador > 0:
                    self.temporizador -= 1
                elif self.temporizador == 0:
                    self.stop[i] = False
                    self.temporizador = -1
                    self.procesando = False

            if self.avance[i]:
                resultado.incrementarTiempo(self.identificador, 'avance')
                if not self.reposo and not self.peticion:
                    # if (tiempo - self.timerAvance[i]) >= 1:     # Hay que
poner delay para los retenedores lentos
                    self.sensor = False
                    self.reposo = True

                if (tiempo - self.timerAvance[i]) > self.tiempoAvance:

sistema[self.salidas[i]].llegada(copy.deepcopy(self.bandejasSalida[i]))
                    self.bandejasSalida[i] = Bandeja()
                    self.avance[i] = False

            if self.peticion:
                if not self.stop[i] and not self.avance[i] and not
sistema[self.salidas[i]].estaOcupado(sistema):
                    self.peticion = False
                    self.avance[i] = True
                    self.timerAvance[i] = tiempo
                    self.bandejasSalida[i] =
copy.deepcopy(self.bandejaEntrada)
                    self.bandejaEntrada = Bandeja()

        if self.peticion:
            if actualizaEsperas:
                if not self.procesando:
                    resultado.incrementarTiempo(self.identificador,
'peticion')
                else:
                    resultado.incrementarTiempo(self.identificador,
'proceso')

        if self.reposo:
            if actualizaEsperas:
resultado.incrementarTiempo(self.identificador, 'reposo')
            if self.sensor:
                self.tiempoEntrada = tiempo
                self.reposo = False
                self.peticion = True

    def influxData(self, measurement, time, logs):
        datos = {
            "measurement": measurement,
            "fields": {
                "estado": self.reposo * 2 ** 0 + self.peticion * 2 ** 1,
                "bandejaEntrada": str(self.bandejaEntrada.id) + ";" +
str(self.bandejaEntrada.tipo)
            },
```

```
            "tags": {
                "idRetenedor": self.identificador
            },
            "time": time
        }
        for i in range(0, len(self.salidas)):
            datos["fields"]["estado"] += int(self.avance[i]) * 2 ** (2 + i)
            datos["fields"]["bandejaSalida" + str(i)] =
str(self.bandejasSalida[i].id) + ";" + str(
                self.bandejasSalida[i].tipo)

        if logs: print(datos)
        return datos
```

1.2 Clase Simulador

```
import time
from influxdb import InfluxDBClient
import Clases
import copy
import tkinter
import Utilities as Ut

class SimuladorInteractivo:  # Integrado en InfluxDB

    # Funciones de la app
    def entradaProductoTipo1(self):  # De momento se ignora el id de la
bandeja
        self.modeloPlanta[0].llegada(Clases.Bandeja(1, 0))
    def entradaProductoTipo2(self):  # De momento se ignora el id de la
bandeja
        self.modeloPlanta[0].llegada(Clases.Bandeja(1, 1))
    def paradaSimulacion(self):
        self.paroSimulacion = True

    def __init__(self, logs):
        ######## Instancion de elementos ########

        # Info base de datos a utilizar
        self.baseDeDatos = 'Pruebas'
        self.measurement = 'Sim'

        # Variables de simulacion
        self.tiempo = 0
        self.paso = 0

        # Parametros de simulacion
        self.desplazamientoTiempoInicio = 240 * 10 ** 9
        self.intervaloTiempo = 1 * 10 ** 9
        self.tiempoReal = False
        self.almacenajeDatos = False
        self.paroSimulacion = False
        self.actualizarResultados = False
        self.logsActivos = logs
        self.logsNN = True
```

```
        # Objetos de la simulacion
        self.indice = [*range(34)]
        self.topologia = [[1], [2], [3], [4], [5, 9], [6], [7], [8], [13],
[10], [11], [12], [8], [14], [15], [16, 25],
                          [17], [18], [19, 29], [20], [21], [22], [23], [6,
24], [2], [26], [27], [28], [30], [30],
                          [31], [32], [33], [20]]
        self.tiempomodeloPlanta = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                                   2, 2, 2, 2, 2, 2]
        self.modeloPlanta = []
        self.modeloPlantaAnterior = []

        # Inicializacion objetos simulacion
        for i in self.indice:
            self.modeloPlanta += [Clases.Retenedor(self.indice[i],
self.topologia, self.tiempomodeloPlanta[i], len(self.topologia))]

        # Iniciacion almacenamiento datos a guardar
        self.dataSend = []

        # Inicializacion almacenamiento resultados
        self.resultado = Clases.Output(2, 34)

        # Instancia cliente influx y borrado de datos anteriores
        if self.almacenajeDatos: self.client = InfluxDBClient('127.0.0.1',
8086, '', '', self.baseDeDatos)
        if self.almacenajeDatos:
self.client.drop_measurement(self.measurement)

        ######## Aplicacion de control ########
        self.app = tkinter.Tk()
        self.app.title("Sim Controller")
        self.etiqueta1 = tkinter.Label(self.app, text="Entrada -> ")
        self.etiqueta1.grid(column=0, row=0)
        self.app.geometry('350x250')
        self.boton1 = tkinter.Button(self.app, text="Producto 1",
command=self.entradaProductoTipo1)
        self.boton1.grid(column=1, row=0)
        self.boton2 = tkinter.Button(self.app, text="Producto 2",
command=self.entradaProductoTipo2)
        self.boton2.grid(column=2, row=0)
        self.boton3 = tkinter.Button(self.app, text="Stop",
command=self.paradaSimulacion)
        self.boton3.grid(column=3, row=0)

    ######## Funciones ########
    # Funciones de control del sistema
    def controlR3(self):
        # Comprobamos el tipo de producto actual
        if self.modeloPlanta[3].peticion:
            if self.modeloPlanta[3].bandejaEntrada.finalizado and not
self.modeloPlanta[3].stop[0]:
                self.modeloPlanta[3].bloquearTiempo(4, 10)
                if self.modeloPlanta[3].bandejaEntrada.tipo == 0:
                    self.resultado.producir(0)
                else:
```

```
                    self.resultado.producir(1)
                self.modeloPlanta[3].bandejaEntrada.vaciar()
    def controlR4(self):
        if self.modeloPlanta[4].peticion:
            self.modeloPlanta[4].bloquear(5)
            self.modeloPlanta[4].bloquear(9)
            if self.modeloPlanta[4].bandejaEntrada.materiaPrima:
                self.modeloPlanta[4].desbloquear(5)
            else:
                self.modeloPlanta[4].desbloquear(9)
    def controlR10(self, NN):
        if self.modeloPlanta[10].peticion:
            if not self.modeloPlanta[10].bandejaEntrada.materiaPrima and not
self.modeloPlanta[10].stop[0]:
                NN.actualizarEntrada(self.modeloPlanta)
                NN.calculaSalida()
                self.modeloPlanta[10].bloquearTiempo(11, 10)
                if self.logsNN: print(NN.salidaModelo)

self.modeloPlanta[10].bandejaEntrada.nuevoProducto(NN.salidaModelo)
                self.modeloPlanta[10].bandejaEntrada.cargarMateriaPrima()
    def controlR15(self):
        if self.modeloPlanta[15].peticion:
            self.modeloPlanta[15].bloquear(16)
            self.modeloPlanta[15].bloquear(25)
            if self.modeloPlanta[15].bandejaEntrada.tipo == 0:
                self.modeloPlanta[15].desbloquear(25)
            else:
                self.modeloPlanta[15].desbloquear(16)
    def controlR18(self):
        if self.modeloPlanta[18].peticion:
            self.modeloPlanta[18].bloquear(19)
            self.modeloPlanta[18].bloquear(29)
            if self.modeloPlanta[18].bandejaEntrada.tipo == 1 and not
self.modeloPlanta[18].bandejaEntrada.finalizado and
self.modeloPlanta[18].bandejaEntrada.materiaPrima:
                self.modeloPlanta[18].desbloquear(29)
            else:
                self.modeloPlanta[18].desbloquear(19)
    def controlR23(self):
        if self.modeloPlanta[23].peticion:
            self.modeloPlanta[23].bloquear(24)
            self.modeloPlanta[23].bloquear(6)
            if self.modeloPlanta[23].bandejaEntrada.finalizado or not
self.modeloPlanta[23].bandejaEntrada.materiaPrima:
                self.modeloPlanta[23].desbloquear(24)
            else:
                self.modeloPlanta[23].desbloquear(6)
    def controlR27(self):
        if self.modeloPlanta[27].peticion:
            if not self.modeloPlanta[27].stop[0] and
self.modeloPlanta[27].bandejaEntrada.tipo == 0 and
self.modeloPlanta[27].bandejaEntrada.materiaPrima and not
self.modeloPlanta[27].bandejaEntrada.finalizado:
                self.modeloPlanta[27].bloquearTiempo(28, 10)
                self.modeloPlanta[27].bandejaEntrada.finalizar()
    def controlR31(self):
```

```
        if self.modeloPlanta[31].peticion:
            if not self.modeloPlanta[31].stop[0] and
self.modeloPlanta[31].bandejaEntrada.tipo == 1 and
self.modeloPlanta[31].bandejaEntrada.materiaPrima and not
self.modeloPlanta[31].bandejaEntrada.finalizado:
                self.modeloPlanta[31].bloquearTiempo(32, 10)
                self.modeloPlanta[31].bandejaEntrada.finalizar()

    # Funcion de carga de datos en influx
    def cargarDatos(self, i):
        if self.almacenajeDatos:
            if self.tiempoReal:
                temporal0 =
self.client.write_points([self.modeloPlanta[i].influxData(self.measurement,
self.tiempoSimulacion, self.logsActivos)])
                if self.logsActivos: print(temporal0)
            else:

self.dataSend.append(self.modeloPlanta[i].influxData(self.measurement,
self.tiempo, self.logsActivos))

    ######## Simulacion del sistema ########
    def simular(self, APIM, salidaResultado, totalSimulacion):
        i = 0

        # Inicializacion temporal
        if self.tiempoReal:
            self.tiempo = time.time_ns()
        else:
            self.tiempo = time.time_ns() - self.desplazamientoTiempoInicio

        # Carga datos iniciales
        if self.almacenajeDatos:
            for i in self.indice:
                self.cargarDatos()

        # Simulacion
        self.paso = 0
        while (self.paso < totalSimulacion or self.tiempoReal) and not
self.paroSimulacion:

            #Entrada de productos
            if not self.tiempoReal:
                if self.paso%20 == 0 and self.paso < 80 and self.paso > 5:
                    self.modeloPlanta[0].llegada(Clases.Bandeja(self.paso, -
1))

            if self.tiempoReal: self.app.update()

            # Copia del estado anterior
            self.modeloPlantaAnterior = copy.deepcopy(self.modeloPlanta)

            # Actualización de self.modeloPlanta en varios pasos
            for n in range(0, 3):
                if n == 2: self.actualizarResultados = True
                else: self.actualizarResultados = False
                # Control de las bandejas
```

```
                self.controlR3()
                self.controlR4()
                self.controlR10(APIM)
                self.controlR15()
                self.controlR18()
                self.controlR23()
                self.controlR27()
                self.controlR31()

                if self.logsActivos: print("")
                if self.logsActivos: Ut.pNoEs(str(self.paso) + "." + str(n))

                for i in self.indice:
                    self.modeloPlanta[i].update(self.modeloPlanta, self.paso,
self.logsActivos, self.resultado, self.actualizarResultados)
                    if n == 2:
                        if self.modeloPlantaAnterior[i] !=
self.modeloPlanta[i]:
                            if self.almacenajeDatos: self.cargarDatos(i,
self.logsActivos)

            # Calculo del tiempo siguiente step
            if self.tiempoReal:
                time.sleep(self.intervaloTiempo / (10 ** 9))
                self.tiempo = time.time_ns()
            else:
                self.tiempo += self.paso * self.intervaloTiempo

            self.paso += 1

        if self.almacenajeDatos:
            for i in self.indice:
                self.cargarDatos(i, self.logsActivos)

        temporal0 = False
        if self.almacenajeDatos and not self.tiempoReal :
            temporal0 = self.client.write_points(self.dataSend)
            if self.logsActivos: print(temporal0)

        if salidaResultado: return self.resultado

class Simulador01:  # Entrenamiento del sistema de control

    def __init__(self, logs):
        ######## Instancion de elementos ########

        # Info base de datos a utilizar
        self.baseDeDatos = 'Pruebas'
        self.measurement = 'Sim'

        # Variables de simulacion
        self.tiempo = 0
        self.paso = 0

        # Parametros de simulacion
        self.desplazamientoTiempoInicio = 240 * 10 ** 9
        self.intervaloTiempo = 1 * 10 ** 9
```

```
        self.tiempoReal = False
        self.almacenajeDatos = False
        self.paroSimulacion = False
        self.actualizarResultados = False
        self.logsActivos = logs
        self.logsNN = True
        self.cargaAnterior = 0

        # Objetos de la simulacion
        self.indice = [*range(34)]
        self.topologia = [[1], [2], [3], [4], [5, 9], [6], [7], [8], [13],
[10], [11], [12], [8], [14], [15], [16, 25],
                          [17], [18], [19, 29], [20], [21], [22], [23], [6,
24], [2], [26], [27], [28], [30], [30],
                          [31], [32], [33], [20]]
        self.tiempomodeloPlanta = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                                   1, 1, 1, 1, 1, 1]
        self.modeloPlanta = []
        self.modeloPlantaAnterior = []
        self.bandejasTotales = 0

        # Inicializacion objetos simulacion
        for i in self.indice:
            self.modeloPlanta += [Clases.Retenedor(self.indice[i],
self.topologia, self.tiempomodeloPlanta[i], len(self.topologia))]

        # Inicializacion almacenamiento resultados
        self.resultado = Clases.Output(2, 34)

    ######## Funciones ########
    # Funciones de control del sistema
    def controlR1(self):
        # Comprobamos el tipo de producto actual
        if self.modeloPlanta[1].peticion:
            if self.modeloPlanta[24].peticion:
                self.modeloPlanta[1].bloquear(2)
            else:
                self.modeloPlanta[1].desbloquear(2)
    def controlR3(self):
        # Comprobamos el tipo de producto actual
        if self.modeloPlanta[3].peticion:
            if self.modeloPlanta[3].bandejaEntrada.finalizado and not
self.modeloPlanta[3].stop[0]:
                self.modeloPlanta[3].bloquearTiempo(4, 10)
                if self.modeloPlanta[3].bandejaEntrada.tipo == 0:
                    self.resultado.producir(0)
                else:
                    self.resultado.producir(1)
                self.modeloPlanta[3].bandejaEntrada.vaciar()
    def controlR4(self):
        if self.modeloPlanta[4].peticion:
            self.modeloPlanta[4].bloquear(5)
            self.modeloPlanta[4].bloquear(9)
            if self.modeloPlanta[4].bandejaEntrada.materiaPrima:
                self.modeloPlanta[4].desbloquear(5)
            else:
```

```
                self.modeloPlanta[4].desbloquear(9)
    def controlR10(self, NN):
        if self.modeloPlanta[10].peticion:
            if not self.modeloPlanta[10].bandejaEntrada.materiaPrima and not
self.modeloPlanta[10].stop[0]:
                NN.actualizarEntrada(self.modeloPlanta)
                NN.calculaSalida()
                self.modeloPlanta[10].bloquearTiempo(11, 10)
                # if self.logsNN: print(NN.salidaModelo)
                # if self.cargaAnterior == 0: self.cargaAnterior = 1
                # else: self.cargaAnterior = 0

self.modeloPlanta[10].bandejaEntrada.nuevoProducto(NN.salidaModelo)
                #
self.modeloPlanta[10].bandejaEntrada.nuevoProducto(self.cargaAnterior)
                self.modeloPlanta[10].bandejaEntrada.cargarMateriaPrima()
    def controlR15(self):
        if self.modeloPlanta[15].peticion:
            self.modeloPlanta[15].bloquear(16)
            self.modeloPlanta[15].bloquear(25)
            if self.modeloPlanta[15].bandejaEntrada.tipo == 0:
                self.modeloPlanta[15].desbloquear(25)
            else:
                self.modeloPlanta[15].desbloquear(16)
    def controlR18(self):
        if self.modeloPlanta[18].peticion:
            self.modeloPlanta[18].bloquear(19)
            self.modeloPlanta[18].bloquear(29)
            if self.modeloPlanta[18].bandejaEntrada.tipo == 1 and not
self.modeloPlanta[18].bandejaEntrada.finalizado and
self.modeloPlanta[18].bandejaEntrada.materiaPrima:
                self.modeloPlanta[18].desbloquear(29)
            else:
                self.modeloPlanta[18].desbloquear(19)
    def controlR23(self):
        if self.modeloPlanta[23].peticion:
            self.modeloPlanta[23].bloquear(24)
            self.modeloPlanta[23].bloquear(6)
            if self.modeloPlanta[23].bandejaEntrada.finalizado or not
self.modeloPlanta[23].bandejaEntrada.materiaPrima:
                self.modeloPlanta[23].desbloquear(24)
            else:
                self.modeloPlanta[23].desbloquear(6)
    def controlR28(self):
        if self.modeloPlanta[28].peticion:
            if not self.modeloPlanta[28].stop[0] and
self.modeloPlanta[28].bandejaEntrada.tipo == 0 and \
                    self.modeloPlanta[28].bandejaEntrada.materiaPrima and not
self.modeloPlanta[
                    28].bandejaEntrada.finalizado:
                self.modeloPlanta[28].bloquearTiempo(30, 100)
                self.modeloPlanta[28].bandejaEntrada.finalizar()
    def controlR31(self):
        if self.modeloPlanta[31].peticion:
            if not self.modeloPlanta[31].stop[0] and
self.modeloPlanta[31].bandejaEntrada.tipo == 1 and
self.modeloPlanta[31].bandejaEntrada.materiaPrima and not
```

```
self.modeloPlanta[31].bandejaEntrada.finalizado:
                self.modeloPlanta[31].bloquearTiempo(32, 100)
                self.modeloPlanta[31].bandejaEntrada.finalizar()

    ######## Simulacion del sistema ########
    def simular(self, NN, totalSimulacion):
        i = 0
        self.paso = 0
        while self.paso < totalSimulacion:
            if self.modeloPlanta[0].reposo and self.bandejasTotales < 100:
                self.modeloPlanta[0].llegada(Clases.Bandeja(self.paso, -1))
                self.bandejasTotales += 1

            # Actualización de self.modeloPlanta en varios pasos
            for n in range(0, 3):

                if n == 2: self.actualizarResultados = True
                else: self.actualizarResultados = False

                # Control de las bandejas
                self.controlR1()
                self.controlR3()
                self.controlR4()
                self.controlR10(NN)
                self.controlR15()
                self.controlR18()
                self.controlR23()
                self.controlR28()
                self.controlR31()

                for i in self.indice:
                    self.modeloPlanta[i].update(self.modeloPlanta, self.paso,
self.logsActivos, self.resultado, self.actualizarResultados)

            self.paso += 1

        return self.resultado

class Simulador02:

    def __init__(self, logs):  # Sistema de control avanzado, fuera scope TFM
        ######## Instancion de elementos ########

        # Info base de datos a utilizar
        self.baseDeDatos = 'Pruebas'
        self.measurement = 'Sim'

        # Variables de simulacion
        self.tiempo = 0
        self.paso = 0

        # Parametros de simulacion
        self.desplazamientoTiempoInicio = 240 * 10 ** 9
        self.intervaloTiempo = 1 * 10 ** 9
        self.tiempoReal = False
        self.almacenajeDatos = False
        self.paroSimulacion = False
```

```
        self.actualizarResultados = False
        self.logsActivos = logs
        self.logsNN = True
        self.cargaAnterior = 0

        # Objetos de la simulacion
        self.indice = [*range(34)]
        self.topologia = [[1], [2], [3], [4], [5, 9], [6], [7], [8], [13],
[10], [11], [12], [8], [14], [15], [16, 25],
                          [17], [18], [19, 29], [20], [21], [22], [23], [6,
24], [2], [26], [27], [28], [30], [30], [31], [32], [33], [20]]
        self.tiempomodeloPlanta = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                                   1, 1, 1, 1, 1, 1]
        self.modeloPlanta = []
        self.modeloPlantaAnterior = []
        self.bandejasTotales = 0

        # Inicializacion objetos simulacion
        for i in self.indice:
            self.modeloPlanta += [Clases.Retenedor(self.indice[i],
self.topologia, self.tiempomodeloPlanta[i], len(self.topologia))]

        # Inicializacion almacenamiento resultados
        self.resultado = Clases.Output(2, 34)

    ######## Funciones ########
    # Funciones de control del sistema


    def controlR1(self):
        # Comprobamos el tipo de producto actual
        if self.modeloPlanta[1].peticion:
            if self.modeloPlanta[24].peticion:
                self.modeloPlanta[1].bloquear(2)
            else:
                self.modeloPlanta[1].desbloquear(2)
    def controlR3(self):
        # Comprobamos el tipo de producto actual
        if self.modeloPlanta[3].peticion:
            if self.modeloPlanta[3].bandejaEntrada.finalizado and not
self.modeloPlanta[3].stop[0]:
                self.modeloPlanta[3].bloquearTiempo(4, 10)
                if self.modeloPlanta[3].bandejaEntrada.tipo == 0:
                    self.resultado.producir(0)
                else:
                    self.resultado.producir(1)
                self.modeloPlanta[3].bandejaEntrada.vaciar()
    def controlR4(self):
        if self.modeloPlanta[4].peticion:
            self.modeloPlanta[4].bloquear(5)
            self.modeloPlanta[4].bloquear(9)
            if self.modeloPlanta[4].bandejaEntrada.materiaPrima:
                self.modeloPlanta[4].desbloquear(5)
            else:
                self.modeloPlanta[4].desbloquear(9)
    def controlR10(self):
```

```
        if self.modeloPlanta[10].peticion:
            if not self.modeloPlanta[10].bandejaEntrada.materiaPrima and not
self.modeloPlanta[10].stop[0]:
                self.modeloPlanta[10].bloquearTiempo(11, 10)
                if self.cargaAnterior == 0: self.cargaAnterior = 1
                else: self.cargaAnterior = 0

self.modeloPlanta[10].bandejaEntrada.nuevoProducto(self.cargaAnterior)
                self.modeloPlanta[10].bandejaEntrada.cargarMateriaPrima()
    def controlR15(self, NN):
        if self.modeloPlanta[15].peticion:
            self.modeloPlanta[15].bloquear(16)
            self.modeloPlanta[15].bloquear(25)
            NN.actualizarEntrada(self.modeloPlanta)
            NN.calculaSalida()
            if NN.salidaModelo == 1 and not
self.modeloPlanta[15].bandejaEntrada.finalizado and
self.modeloPlanta[15].bandejaEntrada.materiaPrima and
self.modeloPlanta[15].bandejaEntrada.tipo == 0:
                self.modeloPlanta[15].desbloquear(25)
            else:
                self.modeloPlanta[15].desbloquear(16)
            # if self.modeloPlanta[15].bandejaEntrada.tipo == 0 and not
self.modeloPlanta[15].bandejaEntrada.finalizado and
self.modeloPlanta[15].bandejaEntrada.materiaPrima:
            #     self.modeloPlanta[15].desbloquear(25)
            # else:
            #     self.modeloPlanta[15].desbloquear(16)
    def controlR18(self, NN):
        if self.modeloPlanta[18].peticion:
            self.modeloPlanta[18].bloquear(19)
            self.modeloPlanta[18].bloquear(29)
            NN.actualizarEntrada(self.modeloPlanta)
            NN.calculaSalida()
            if NN.salidaModelo == 1 and not
self.modeloPlanta[18].bandejaEntrada.finalizado and
self.modeloPlanta[18].bandejaEntrada.materiaPrima and
self.modeloPlanta[15].bandejaEntrada.tipo == 1:
                self.modeloPlanta[18].desbloquear(29)
            else:
                self.modeloPlanta[18].desbloquear(19)
            # if self.modeloPlanta[18].bandejaEntrada.tipo == 1 and not
self.modeloPlanta[18].bandejaEntrada.finalizado and
self.modeloPlanta[18].bandejaEntrada.materiaPrima:
            #     self.modeloPlanta[18].desbloquear(29)
            # else:
            #     self.modeloPlanta[18].desbloquear(19)
    def controlR23(self):
        if self.modeloPlanta[23].peticion:
            self.modeloPlanta[23].bloquear(24)
            self.modeloPlanta[23].bloquear(6)
            if self.modeloPlanta[23].bandejaEntrada.finalizado or not
self.modeloPlanta[23].bandejaEntrada.materiaPrima:
                self.modeloPlanta[23].desbloquear(24)
            else:
                self.modeloPlanta[23].desbloquear(6)
    def controlR28(self):
```

```
        if self.modeloPlanta[28].peticion:
            if not self.modeloPlanta[28].stop[0] and
self.modeloPlanta[28].bandejaEntrada.tipo == 0 and \
                    self.modeloPlanta[28].bandejaEntrada.materiaPrima and not
self.modeloPlanta[
                    28].bandejaEntrada.finalizado:
                self.modeloPlanta[28].bloquearTiempo(30, 100)
                self.modeloPlanta[28].bandejaEntrada.finalizar()
    def controlR31(self):
        if self.modeloPlanta[31].peticion:
            if not self.modeloPlanta[31].stop[0] and
self.modeloPlanta[31].bandejaEntrada.tipo == 1 and
self.modeloPlanta[31].bandejaEntrada.materiaPrima and not
self.modeloPlanta[31].bandejaEntrada.finalizado:
                self.modeloPlanta[31].bloquearTiempo(32, 100)
                self.modeloPlanta[31].bandejaEntrada.finalizar()

    ######## Simulacion del sistema ########
    def simular(self, NN1, NN2, totalSimulacion):
        i = 0
        self.paso = 0
        while self.paso < totalSimulacion:
            if self.modeloPlanta[0].reposo and self.bandejasTotales < 100:
                self.modeloPlanta[0].llegada(Clases.Bandeja(self.paso, -1))
                self.bandejasTotales += 1

            # Actualización de self.modeloPlanta en varios pasos
            for n in range(0, 3):

                if n == 2: self.actualizarResultados = True
                else: self.actualizarResultados = False

                # Control de las bandejas
                self.controlR1()
                self.controlR3()
                self.controlR4()
                self.controlR10()
                self.controlR15(NN1)
                self.controlR18(NN2)
                self.controlR23()
                self.controlR28()
                self.controlR31()

                for i in self.indice:
                    self.modeloPlanta[i].update(self.modeloPlanta, self.paso,
self.logsActivos, self.resultado, self.actualizarResultados)


            self.paso += 1

        return self.resultado
```

2 Red Neuronal

```
import torch
import torch.distributions.normal
import pickle
import Utilities as Ut
import os

class NN:
    def __init__(self):
        self.dtype = torch.float
        self.device = torch.device("cpu")

    def guardarModelo(self, number):
        if os.path.basename(os.getcwd()) != "Modelos":
            os.chdir('Modelos')
        with open(str('Modelo' + number), 'wb') as file:
            print()
            print('Guardado Modelo')
            pickle.dump(self, file)

    def cargarModelo(self, number):
        print(os.getcwd())
        if os.path.basename(os.getcwd()) != "Modelos":
            os.chdir('Modelos')
            print("Cambiando de directorio")
            print(os.getcwd())
        try:
            with open(str('Modelo' + number), 'rb') as file:
                print('Cargando Modelo')
                modelo = pickle.load(file)
                self.__dict__.update(modelo.__dict__)
        except:
            print("Modelo previo no encontrado")

class NN_doscapas_unasalida_nograd(NN):
    def __init__(self, dimensionEntrada):
        super().__init__()
        self.dimensionEntrada = dimensionEntrada
        self.entradaModelo = [[]]
        self.capa1 = torch.empty(self.dimensionEntrada,
self.dimensionEntrada, device=self.device,
                                 dtype=self.dtype).normal_(mean=0, std=1)
        self.capa2 = torch.empty(self.dimensionEntrada, 1,
device=self.device, dtype=self.dtype).normal_(mean=0, std=1)
        self.salidaModelo = 0

    def mutarCapas(self, coeficiente):
        mutacion1 = torch.empty(self.dimensionEntrada,
self.dimensionEntrada).normal_(mean=0.5, std=1)
        mutacion1 = torch.where(mutacion1 < coeficiente, torch.tensor(1),
torch.tensor(0))
        self.capa1 = torch.where(mutacion1 == 0, self.capa1,
                                 torch.empty(1, device=self.device,
dtype=self.dtype).normal_(mean=0, std=1))
```

```
        mutacion2 = torch.empty(self.dimensionEntrada, 1).normal_(mean=0.5,
std=1)
        mutacion2 = torch.where(mutacion2 < coeficiente, torch.tensor(1),
torch.tensor(0))
        self.capa2 = torch.where(mutacion2 == 0, self.capa2,
                                 torch.empty(1, device=self.device,
dtype=self.dtype).normal_(mean=0, std=1))

    def calculaSalida(self):
        entrada = torch.tensor(self.entradaModelo,
device=torch.device("cpu"), dtype=torch.float)
        salida = entrada.mm(self.capa1).sigmoid().mm(self.capa2).item()

        if salida > 0.5:
            self.salidaModelo = 1
        else:
            self.salidaModelo = 0

    def mezcla(nn1, nn2, coeficiente):
        nn1.capa1 = (nn1.capa1 * coeficiente + nn2.capa1 * (1 - coeficiente))
        nn1.capa2 = (nn1.capa2 * coeficiente + nn2.capa2 * (1 - coeficiente))
        return nn1


class NN_doscapas_unasalida_grad(NN):
    def __init__(self, dimensionesEntrada):
        super().__init__()
        self.dimensionesEntrada = dimensionesEntrada
        self.entradaModelo = [[]]
        self.entrada = torch.randn(1, 1, device=self.device,
dtype=self.dtype)
        self.salidaModelo = 0
        self.salida = torch.randn(1, device=self.device, dtype=self.dtype)
        self.salidaDeseada = torch.randn(1, device=self.device,
dtype=self.dtype)

        self.capa1 = torch.randn(self.dimensionEntrada,
self.dimensionEntrada, device=self.device,
                                 dtype=self.dtype, requires_grad=True)
        self.capa2 = torch.randn(self.dimensionEntrada, 1,
device=self.device,
                                 dtype=self.dtype, requires_grad=True)

    def calculaSalida(self):

        self.entrada = torch.tensor(self.entradaModelo, device=self.device,
dtype=torch.float)
        self.salida = self.entrada.mm(self.capa1).sigmoid().mm(self.capa2)

        if self.salida.item() > 0.5:
            self.salidaModelo = 1
        else:
            self.salidaModelo = 0

    def entrenar(self, salidaDeseada, coeficienteAprendizage):
        self.salidaDeseada = torch.tensor(salidaDeseada, device=self.device,
dtype=self.dtype)
```

```
        coste = (self.salida - self.salidaDeseada).pow(2).sum()
        # print("coste")
        # print(coste)
        coste.backward()
        with torch.no_grad():
            self.capa1 -= coeficienteAprendizage * self.capa1.grad
            self.capa2 -= coeficienteAprendizage * self.capa2.grad
            self.capa1.grad.zero_()
            self.capa2.grad.zero_()
        return coste


class NN_trescapas_unasalida_grad(NN):
    def __init__(self, dimensionesEntrada):
        super().__init__()
        self.dimensionesEntrada = dimensionesEntrada
        self.entradaModelo = [[]]
        self.entrada = torch.randn(1, 1, device=self.device,
dtype=self.dtype)
        self.salidaModelo = 0
        self.salida = torch.randn(1, device=self.device, dtype=self.dtype)
        self.salidaDeseada = torch.randn(1, device=self.device,
dtype=self.dtype)

        self.capa1 = torch.randn(self.dimensionEntrada,
self.dimensionEntrada, device=self.device,
                                 dtype=self.dtype, requires_grad=True)
        self.capa2 = torch.randn(self.dimensionEntrada,
self.dimensionEntrada, device=self.device,
                                 dtype=self.dtype, requires_grad=True)
        self.capa3 = torch.randn(self.dimensionEntrada, 1,
device=self.device,
                                 dtype=self.dtype, requires_grad=True)

    def calculaSalida(self):

        self.entrada = torch.tensor(self.entradaModelo, device=self.device,
dtype=torch.float)
        self.salida =
self.entrada.mm(self.capa1).sigmoid().mm(self.capa2).sigmoid().mm(self.capa3)

        if self.salida.item() > 0.5:
            self.salidaModelo = 1
        else:
            self.salidaModelo = 0

    def entrenar(self, salidaDeseada, coeficienteAprendizage):
        self.salidaDeseada = torch.tensor(salidaDeseada, device=self.device,
dtype=self.dtype)
        coste = (self.salida - self.salidaDeseada).pow(2).sum()
        coste.backward()
        with torch.no_grad():
            self.capa1 -= coeficienteAprendizage * self.capa1.grad
            self.capa2 -= coeficienteAprendizage * self.capa2.grad
            self.capa1.grad.zero_()
            self.capa2.grad.zero_()
        return coste
```

```
class NN01_01(NN_doscapas_unasalida_nograd):

    def __init__(self, retenedoresEntrada, sistema):
        self.retenedoresEntrada = retenedoresEntrada
        dimensionEntrada = 0
        for i in self.retenedoresEntrada:
            dimensionEntrada += 2
            for j in range(0, len(sistema[i].salidas)):
                dimensionEntrada += 2
        super().__init__(dimensionEntrada)

    def __entradaDatos(self, i, sistema):
        self.entradaModelo[0].append(sistema[i].peticion)
        for j in range(0, len(sistema[i].salidas)):
            self.entradaModelo[0].append(sistema[i].avance[j])
        self.entradaModelo[0].append(sistema[i].bandejaEntrada.tipo)
        for j in range(0, len(sistema[i].salidas)):
            self.entradaModelo[0].append(sistema[i].bandejasSalida[j].tipo)

    def actualizarEntrada(self, sistema):
        self.entradaModelo[0] = []
        for i in self.retenedoresEntrada:
            self.__entradaDatos(i, sistema)

    def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
        aux1 = sum(resultados.produccion)
        aux2 = sum(resultados.tiempoPeticion)
        aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
        data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
        return data


class NN01_02(NN_doscapas_unasalida_nograd):

    def __init__(self, colas):
        self.colas = colas
        self.dimensionEntrada = 2*len(colas)
        super().__init__(self.dimensionEntrada)

    def __entradaDatos(self, i, sistema):
        ocupacion = [0, 0]
        total = 0
        for j in i:
            if sistema[j].bandejaEntrada.tipo != -1:
                ocupacion[sistema[j].bandejaEntrada.tipo] += 1
            total += 1
            for n in range(0, len(sistema[j].salidas)):
                if sistema[j].bandejasSalida[n].tipo != -1:
                    ocupacion[sistema[j].bandejaEntrada.tipo] += 1
                total += 1
        self.entradaModelo[0].append(ocupacion[0] / total)
        self.entradaModelo[0].append(ocupacion[1] / total)

    def actualizarEntrada(self, sistema):
        self.entradaModelo[0] = []
```

```
            for i in self.colas:
                self.__entradaDatos(i, sistema)

        def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
            aux1 = sum(resultados.produccion)
            aux2 = sum(resultados.tiempoPeticion)
            aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
            data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
            return data


    class NN01_03(NN_doscapas_unasalida_grad):

        def __init__(self, colas):
            self.colas = colas
            self.dimensionEntrada = 2*len(colas)
            super().__init__(self.dimensionEntrada)

        def __entradaDatos(self, i, sistema):
            ocupacion = [0, 0]
            total = 0
            for j in i:
                if sistema[j].bandejaEntrada.tipo != -1:
                    ocupacion[sistema[j].bandejaEntrada.tipo] += 1
                total += 1
                for n in range(0, len(sistema[j].salidas)):
                    if sistema[j].bandejasSalida[n].tipo != -1:
                        ocupacion[sistema[j].bandejasSalida[n].tipo] += 1
                    total += 1
            self.entradaModelo[0].append(ocupacion[0]/total)
            self.entradaModelo[0].append(ocupacion[1]/total)

        def actualizarEntrada(self, sistema):
            self.entradaModelo[0] = []
            for i in self.colas:
                self.__entradaDatos(i, sistema)

        def setEntrada(self, entrada):
            self.entradaModelo = entrada

        def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
            aux1 = sum(resultados.produccion)
            aux2 = sum(resultados.tiempoPeticion)
            aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
            data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
            return data


    class NN02_01(NN_doscapas_unasalida_nograd):

        def __init__(self, retenedoresEntrada, sistema):
            self.retenedoresEntrada = retenedoresEntrada
            self.dimensionEntrada = 0
            for i in self.retenedoresEntrada:
                self.dimensionEntrada += 1
                for j in range(0, len(sistema[i].salidas)):
                    self.dimensionEntrada += 1
```

```
        super().__init__(self.dimensionEntrada)

    def __entradaDatos(self, i, sistema):
        self.entradaModelo[0].append(sistema[i].bandejaEntrada.tipo)
        for j in range(0, len(sistema[i].salidas)):
            self.entradaModelo[0].append(sistema[i].bandejasSalida[j].tipo)

    def actualizarEntrada(self, sistema):
        self.entradaModelo[0] = []
        for i in self.retenedoresEntrada:
            self.__entradaDatos(i, sistema)


class NN02_01_01(NN02_01):

    def __init__(self, retenedoresEntrada, sistema):
        super().__init__(retenedoresEntrada, sistema)

    def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
        # Ut.pNoEs("Modelo 1 Produccion: ")
        # Ut.pNoEs(resultados.produccion[0])
        aux1 = resultados.produccion[0]
        aux2 = sum(resultados.tiempoPeticion)
        aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
        data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
        # Ut.pNoEs(' Fitness: ')
        # print(data)
        return data


class NN02_01_02(NN02_01):

    def __init__(self, retenedoresEntrada, sistema):
        super().__init__(retenedoresEntrada, sistema)

    def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
        # Ut.pNoEs("Modelo 2 Produccion: ")
        # Ut.pNoEs(resultados.produccion[1])
        aux1 = resultados.produccion[1]
        aux2 = sum(resultados.tiempoPeticion)
        aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
        data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
        # Ut.pNoEs(' Fitness: ')
        # print(data)
        return data


class NN02_02(NN_doscapas_unasalida_nograd):

    def __init__(self, colas):
        self.colas = colas
        self.dimensionEntrada = 2 * len(colas)
        super().__init__(self.dimensionEntrada)

    def __entradaDatos(self, i, sistema):
        ocupacion = [0, 0]
        total = 0
```

```
            for j in i:
                if sistema[j].bandejaEntrada.tipo != -1:
                    ocupacion[sistema[j].bandejaEntrada.tipo] += 1
                total += 1
                for n in range(0, len(sistema[j].salidas)):
                    if sistema[j].bandejasSalida[n].tipo != -1:
                        ocupacion[sistema[j].bandejaEntrada.tipo] += 1
                    total += 1
            self.entradaModelo[0].append(ocupacion[0] / total)
            self.entradaModelo[0].append(ocupacion[1] / total)

        def actualizarEntrada(self, sistema):
            self.entradaModelo[0] = []
            for i in self.colas:
                self.__entradaDatos(i, sistema)


    class NN02_02_01(NN02_02):

        def __init__(self, colas):
            super().__init__(colas)

        def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
            # Ut.pNoEs("Modelo 1 Produccion: ")
            # Ut.pNoEs(resultados.produccion[0])
            aux1 = resultados.produccion[0]
            aux2 = sum(resultados.tiempoPeticion)
            aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
            data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
            # Ut.pNoEs(' Fitness: ')
            # print(data)
            return data


    class NN02_02_02(NN02_02):

        def __init__(self, colas):
            super().__init__(colas)

        def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
            # Ut.pNoEs("Modelo 2 Produccion: ")
            # Ut.pNoEs(resultados.produccion[1])
            aux1 = resultados.produccion[1]
            aux2 = sum(resultados.tiempoPeticion)
            aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
            data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
            # Ut.pNoEs(' Fitness: ')
            # print(data)
            return data


    class NN02_03(NN_doscapas_unasalida_grad):

        def __init__(self, colas):
            self.colas = colas
            self.dimensionEntrada = 2*len(colas)
            super().__init__(self.dimensionEntrada)
```

```
    def __entradaDatos(self, i, sistema):
        ocupacion = [0, 0]
        total = 0
        for j in i:
            if sistema[j].bandejaEntrada.tipo != -1:
                ocupacion[sistema[j].bandejaEntrada.tipo] += 1
            total += 1
            for n in range(0, len(sistema[j].salidas)):
                if sistema[j].bandejasSalida[n].tipo != -1:
                    ocupacion[sistema[j].bandejasSalida[n].tipo] += 1
                total += 1
        self.entradaModelo[0].append(ocupacion[0]/total)
        self.entradaModelo[0].append(ocupacion[1]/total)

    def actualizarEntrada(self, sistema):
        self.entradaModelo[0] = []
        for i in self.colas:
            self.__entradaDatos(i, sistema)

    def setEntrada(self, entrada):
        self.entradaModelo = entrada


class NN02_03_01(NN02_03):

    def __init__(self, colas):
        super().__init__(colas)

    def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
        # Ut.pNoEs("Modelo 1 Produccion: ")
        # Ut.pNoEs(resultados.produccion[0])
        aux1 = resultados.produccion[0]
        aux2 = sum(resultados.tiempoPeticion)
        aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
        data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
        # Ut.pNoEs(' Fitness: ')
        # print(data)
        return data


class NN02_03_02(NN02_03):

    def __init__(self, colas):
        super().__init__(colas)

    def fitness(self, resultados, coeficienteProduccion, coeficienteColas):
        # Ut.pNoEs("Modelo 2 Produccion: ")
        # Ut.pNoEs(resultados.produccion[1])
        aux1 = resultados.produccion[1]
        aux2 = sum(resultados.tiempoPeticion)
        aux2 = aux2 / ((aux2 ** 2 + 1) ** 0.5)
        data = aux1 * coeficienteProduccion + aux2 * coeficienteColas
        # Ut.pNoEs(' Fitness: ')
        # print(data)
        return data
```

3 Nucleos de entrenamiento

3.1 Algoritmo Genético

```
import Sim
import Clases
import copy
import RedNeuronal
import Utilities as Ut

# Modelo01_01 control de producto a fabricar - Genetico
individuosPorEpoca = 10
generaciones = 40
tiempoSimulacion = 2000
individuosSeleccion = 2
coeficienteProduccion = 0.8
coeficienteColas = 0.2
coeficienteMutacion = 0.15
seleccion = [0, 0]

NN = []
resultados = []
fitness = []

simulador = Sim.Simulador01(logs=True)

retenedoresEntrada01 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19,
                        20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33]

for i in range(0, individuosPorEpoca):
    NN.append(RedNeuronal.NN01_01(retenedoresEntrada01,
simulador.modeloPlanta))

for individuo in range(0, len(NN)):
    NN[individuo].cargarModelo('01_01')

for generacion in range(0, 200):
    seleccion = [0, 0]
    print()
    Ut.pNoEs('Generacion ')
    print(generacion)
    resultados.append([])
    fitness.append([])
    for individuo in NN:

        simulador = Sim.Simulador01(logs=True)
        resultados[generacion].append(simulador.simular(individuo,
tiempoSimulacion))

        fitness[generacion].append(individuo.fitness(resultados[generacion][-
1], coeficienteProduccion, coeficienteColas))
```

```
            print(resultados[-1][-1].produccion)
            print(fitness[-1][-1])
        indice = 0

        for fitnessIndividuo in fitness[generacion]:
            if fitnessIndividuo > fitness[generacion][seleccion[0]]:
                seleccion[1] = copy.deepcopy(seleccion[0])
                seleccion[0] = copy.copy(indice)
            elif fitnessIndividuo > fitness[generacion][seleccion[1]]:
                seleccion[1] = copy.copy(indice)
            indice += 1

        seleccion[0] = copy.deepcopy(NN[seleccion[0]])
        seleccion[1] = copy.deepcopy(NN[seleccion[1]])

        factor = 0

        NN[0] = copy.deepcopy(seleccion[0])
        NN[1] = copy.deepcopy(seleccion[1])

        for individuo in range(2, len(NN)):
            if individuo%2 == 0: NN[individuo] = copy.deepcopy(seleccion[0])
            else: NN[individuo] = copy.deepcopy(seleccion[1])
            NN[individuo].mutarCapas(coeficienteMutacion)

    NN[0].guardarModelo('01_01')
```

3.2 Algoritmo Genético Modelo Reducido

```
import Sim
import Clases
import copy
import RedNeuronal
import Utilities as Ut

# Modelo01_02 control de producto a fabricar - Genetico Colas
individuosPorEpoca = 10
generaciones = 40
tiempoSimulacion = 2000
individuosSeleccion = 2
coeficienteProduccion = 0.8
coeficienteColas = 0.2
coeficienteMutacion = 0.15
seleccion = [0, 0]

NN = []
resultados = []
fitness = []

simulador = Sim.Simulador01(logs=True)

colas = [[11, 12], [6, 7, 19, 20, 21, 22, 23, 32, 33], [8, 13, 14], [15, 25,
26, 27], [16, 17, 18], [29, 30]]
```

```
for i in range(0, individuosPorEpoca):
    NN.append(RedNeuronal.NN01_02(colas))

for individuo in range(0, len(NN)):
    NN[individuo].cargarModelo('01_02_prueba')

for generacion in range(0, 200):
    seleccion = [0, 0]
    print()
    Ut.pNoEs('Generacion ')
    print(generacion)
    resultados.append([])
    fitness.append([])
    for individuo in NN:

        simulador = Sim.Simulador01(logs=True)
        resultados[generacion].append(simulador.simular(individuo,
tiempoSimulacion))

        fitness[generacion].append(individuo.fitness(resultados[generacion][-
1], coeficienteProduccion, coeficienteColas))
        print(resultados[-1][-1].produccion)
        print(fitness[-1][-1])
    indice = 0

    for fitnessIndividuo in fitness[generacion]:
        if fitnessIndividuo > fitness[generacion][seleccion[0]]:
            seleccion[1] = copy.deepcopy(seleccion[0])
            seleccion[0] = copy.copy(indice)
        elif fitnessIndividuo > fitness[generacion][seleccion[1]]:
            seleccion[1] = copy.copy(indice)
        indice += 1

    seleccion[0] = copy.deepcopy(NN[seleccion[0]])
    seleccion[1] = copy.deepcopy(NN[seleccion[1]])

    factor = 0

    NN[0] = copy.deepcopy(seleccion[0])
    NN[1] = copy.deepcopy(seleccion[1])

    for individuo in range(2, len(NN)):
        if individuo%2 == 0: NN[individuo] = copy.deepcopy(seleccion[0])
        else: NN[individuo] = copy.deepcopy(seleccion[1])
        NN[individuo].mutarCapas(coeficienteMutacion)

NN[0].guardarModelo('01_02_prueba')
```

3.3 Descenso del gradiente estocástico

```
import Sim
import Clases
import copy
import RedNeuronal
```

```
import Utilities as Ut

# Modelo01_03 control de producto a fabricar - Genetico Backprop
individuosPorEpoca = 10
generaciones = 40
tiempoSimulacion = 2000
individuosSeleccion = 2
coeficienteProduccion = 0.8
coeficienteColas = 0.2
coeficienteMutacion = 0.15
seleccion = [0, 0]

NN = 0
resultados = 0
fitness = 0
coste = 0

simulador = Sim.Simulador01(logs=True)

colas = [[11, 12],
         [6, 7, 19, 20, 21, 22, 23, 32, 33],
         [8, 13, 14],
         [15, 25, 26, 27],
         [16, 17, 18],
         [29, 30]]

# 0 -> Ocupacion de tipo 0
# 1 -> Ocupacion de tipo 1
############### 01 ## 02 ## 03 ## 04 ## 05 ## 06 ##
###############0  1  0  1  0  1  0  1  0  1  0  1######
dataset =  [[[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], 1],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], 0],
            [[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0], 1],
            [[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], 0],
            [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], 1],
            [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], 0],
            [[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 1],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], 0],
            [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 1],
            [[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 0],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 0]
            ]

NN = RedNeuronal.NN01_03(colas)

NN.cargarModelo('01_03_prueba')

for epoch in range(0, 5000):
    if epoch % 500 == 0:
        Ut.pNoEs('Epoch ')
        print(epoch)
    if epoch % 100 == 99:
        Ut.pNoEs('Error ')
        print(costeAv/len(dataset))
    try: datafile.writelines(str(costeAv/len(dataset)) + '\n')
    except: pass
    costeAv = 0
```

```
    for sample in dataset:
        NN.setEntrada([sample[0]])
        NN.calculaSalida()
        coste = NN.entrenar(sample[1], 3e-3)

        try: costeAv += coste.item()
        except: pass


simulador = Sim.Simulador01(logs=True)
resultados = simulador.simular(NN, tiempoSimulacion)

fitness = NN.fitness(resultados, coeficienteProduccion, coeficienteColas)
print(resultados.produccion)
print(fitness)

NN.guardarModelo('01_03_prueba')
```

3.4 Sistemas de control avanzados

```
import Sim
import Clases
import copy
import RedNeuronal
import Utilities as Ut

# Modelo02_01 control de bifurcaciones - Genetico
individuosPorGeneracion = 15
generaciones = 40
tiempoSimulacion = 2000
individuosSeleccion = 2
coeficienteProduccion = 0.8
coeficienteColas = 0.2
coeficienteMutacion = 0.15
seleccion = [[0, 0], [0, 0]]

NN1 = []
NN2 = []
resultados = []
fitness = []

simulador = Sim.Simulador02(logs=True)

retenedoresEntrada01 = [6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                        20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33]
retenedoresEntrada02 = [6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                        20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33]

for i in range(0, individuosPorGeneracion):
    NN1.append(RedNeuronal.NN02_01_01(retenedoresEntrada01,
simulador.modeloPlanta))
for i in range(0, individuosPorGeneracion):
    NN2.append(RedNeuronal.NN02_01_02(retenedoresEntrada02,
simulador.modeloPlanta))
```

```
for individuo in range(0, len(NN1)):
    NN1[individuo].cargarModelo('02_01_01_prueba')

for individuo in range(0, len(NN2)):
    NN2[individuo].cargarModelo('02_01_02_prueba')

for generacion in range(0, generaciones):
    seleccion = [[0, 0], [0, 0]]
    print()
    Ut.pNoEs('Generacion ')
    print(generacion)
    resultados.append([])
    fitness.append([])
    for individuo in range(0, len(NN1)):
        simulador = Sim.Simulador02(logs=True)
        resultados[generacion].append(simulador.simular(NN1[individuo],
NN2[individuo], tiempoSimulacion))

        fitness[generacion].append(
            [NN1[individuo].fitness(resultados[generacion][-1],
coeficienteProduccion, coeficienteColas),
             NN2[individuo].fitness(resultados[generacion][-1],
coeficienteProduccion, coeficienteColas)])
        Ut.pNoEs("Produccion: ")
        Ut.pNoEs(resultados[-1][0].produccion[0])
        Ut.pNoEs(" : ")
        print(resultados[-1][0].produccion[1])
    indice = 0
    for fitnessIndividuo in fitness[generacion]:
        if fitnessIndividuo[0] > fitness[generacion][seleccion[0][0]][0]:
            seleccion[0][1] = copy.deepcopy(seleccion[0][0])
            seleccion[0][0] = copy.copy(indice)
        elif fitnessIndividuo[0] > fitness[generacion][seleccion[0][1]][0]:
            seleccion[0][1] = copy.copy(indice)
        if fitnessIndividuo[1] > fitness[generacion][seleccion[1][0]][1]:
            seleccion[1][1] = copy.deepcopy(seleccion[0][0])
            seleccion[1][0] = copy.copy(indice)
        elif fitnessIndividuo[1] > fitness[generacion][seleccion[1][1]][1]:
            seleccion[1][1] = copy.copy(indice)
        indice += 1

    seleccion[0][0] = copy.deepcopy(NN1[seleccion[0][0]])
    seleccion[0][1] = copy.deepcopy(NN1[seleccion[0][1]])
    seleccion[1][0] = copy.deepcopy(NN2[seleccion[1][0]])
    seleccion[1][1] = copy.deepcopy(NN2[seleccion[1][1]])

    factor = 0

    NN1[0] = copy.deepcopy(seleccion[0][0])
    NN1[1] = copy.deepcopy(seleccion[0][1])
    NN2[0] = copy.deepcopy(seleccion[1][0])
    NN2[1] = copy.deepcopy(seleccion[1][1])

    for individuo in range(2, len(NN1)):
        if individuo % 2 == 0:
            NN1[individuo] = copy.deepcopy(seleccion[0][0])
```

```
            NN2[individuo] = copy.deepcopy(seleccion[1][0])
        else:
            NN1[individuo] = copy.deepcopy(seleccion[0][1])
            NN2[individuo] = copy.deepcopy(seleccion[1][1])

        NN1[individuo].mutarCapas(coeficienteMutacion)
        NN2[individuo].mutarCapas(coeficienteMutacion)

    Ut.pNoEs("Produccion: ")
    Ut.pNoEs(resultados[-1][0].produccion[0])
    Ut.pNoEs(" : ")
    Ut.pNoEs(resultados[-1][0].produccion[1])

NN1[0].guardarModelo('02_01_01_prueba')
NN2[0].guardarModelo('02_01_02_prueba')
```

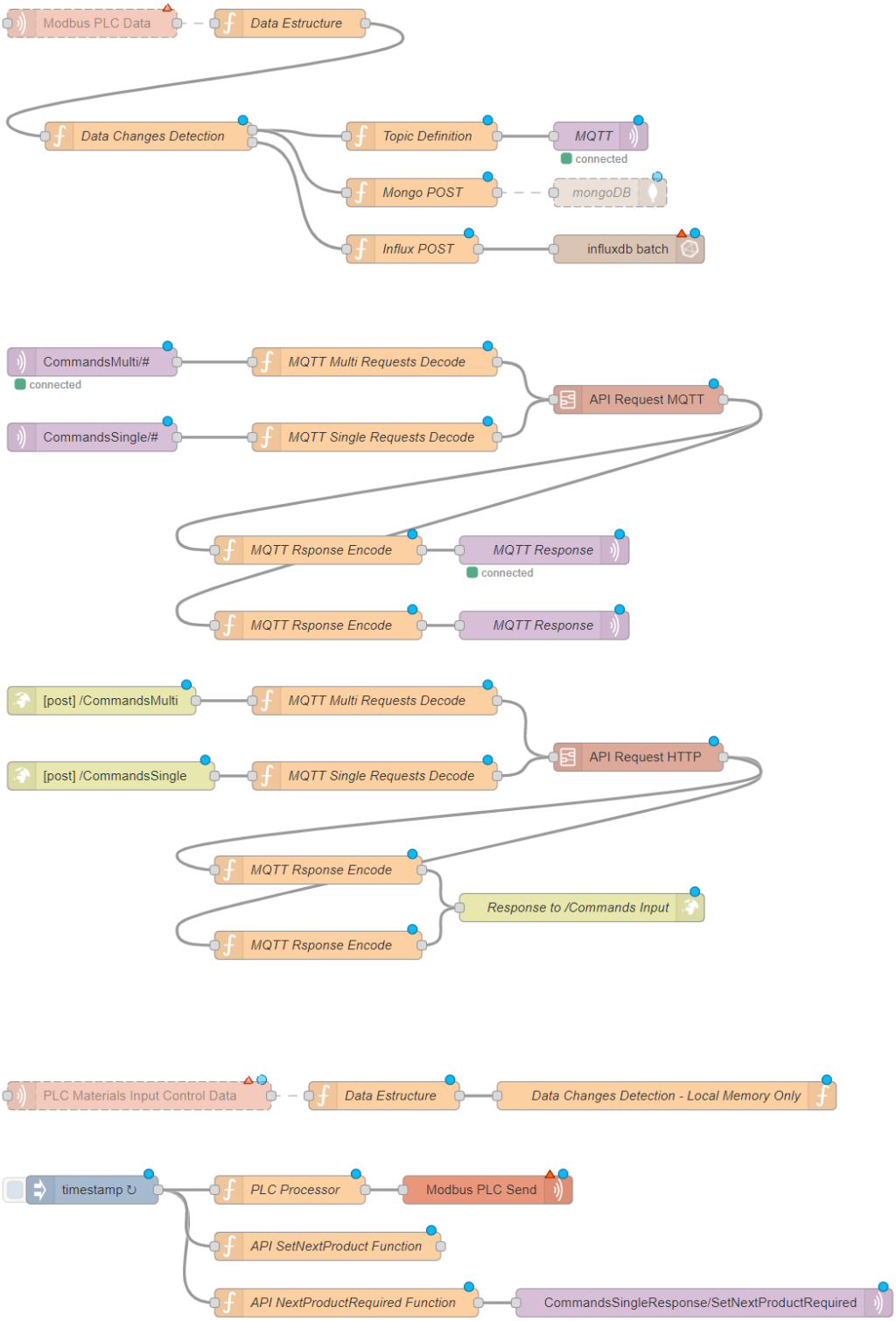
```
import Sim
import Clases
import copy
import RedNeuronal
import Utilities as Ut

# Modelo02_02 control de bifurcaciones - Genetico Colas
individuosPorGeneracion = 6
generaciones = 10
tiempoSimulacion = 2000
coeficienteProduccion = 0.8
coeficienteColas = 0.2
coeficienteMutacion = 0.15
seleccion = [[0, 0], [0, 0]]

NN1 = []
NN2 = []
resultados = []
fitness = []

simulador = Sim.Simulador02(logs=True)

colas1 = [[11, 12], [6, 7, 19, 20, 21, 22, 23, 32, 33], [8, 13, 14], [15, 25,
26, 27], [16, 17, 18], [29, 30]]
colas2 = [[11, 12], [6, 7, 19, 20, 21, 22, 23, 32, 33], [8, 13, 14], [15, 25,
26, 27], [16, 17, 18], [29, 30]]

for i in range(0, individuosPorGeneracion):
    NN1.append(RedNeuronal.NN02_02_01(colas1))
for i in range(0, individuosPorGeneracion):
    NN2.append(RedNeuronal.NN02_02_02(colas2))

for individuo in range(0, len(NN1)):
    NN1[individuo].cargarModelo('02_03_01_v2')

for individuo in range(0, len(NN2)):
    NN2[individuo].cargarModelo('02_03_02_v2')
```

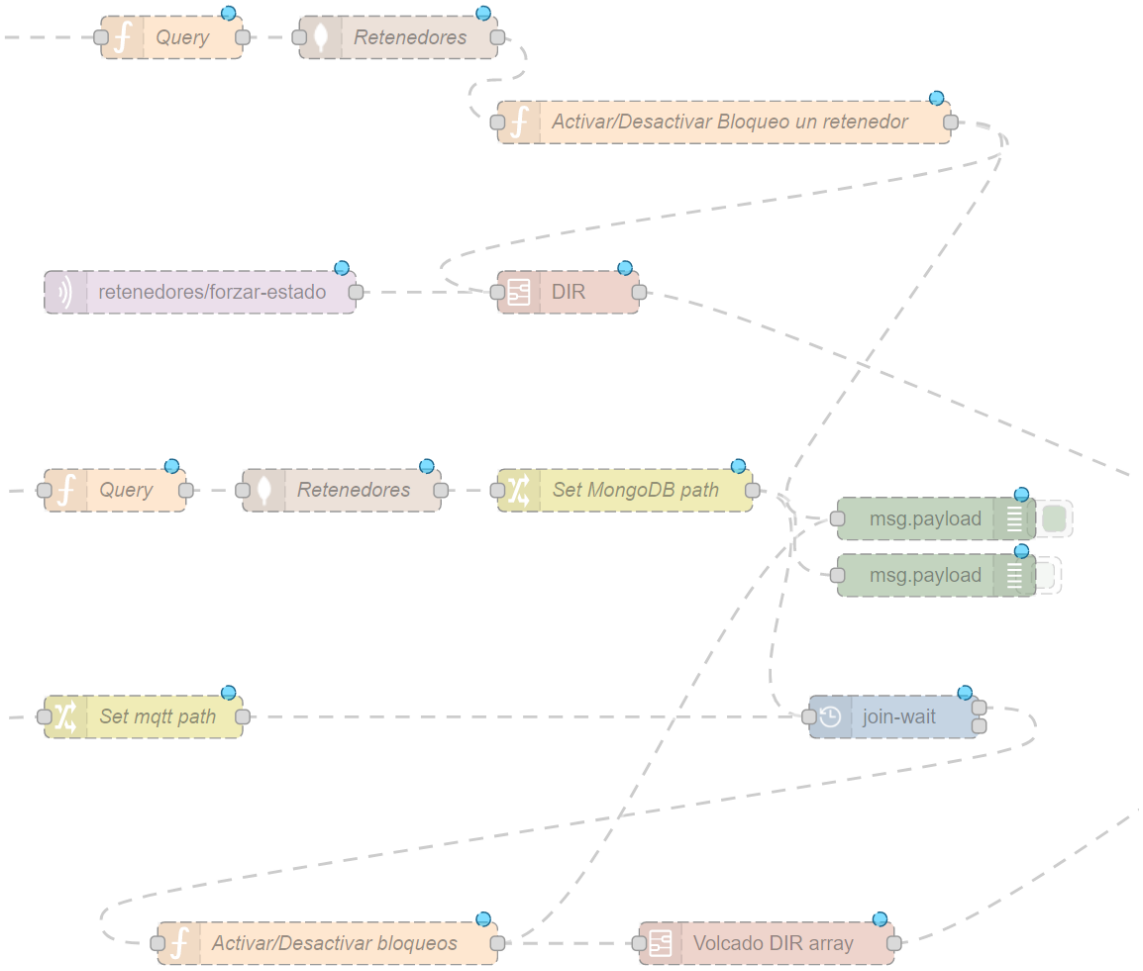
```
for generacion in range(0, generaciones):
    seleccion = [[0, 0], [0, 0]]
    print()
    Ut.pNoEs('Generacion ')
    print(generacion)
    resultados.append([])
    fitness.append([])
    for individuo in range(0, len(NN1)):
        simulador = Sim.Simulador02(logs=True)
        resultados[generacion].append(simulador.simular(NN1[individuo],
NN2[individuo], tiempoSimulacion))
        fitness[generacion].append(
            [NN1[individuo].fitness(resultados[generacion][-1],
coeficienteProduccion, coeficienteColas),
             NN2[individuo].fitness(resultados[generacion][-1],
coeficienteProduccion, coeficienteColas)])
        Ut.pNoEs("Produccion: ")
        Ut.pNoEs(resultados[-1][-1].produccion[0])
        Ut.pNoEs(" : ")
        print(resultados[-1][-1].produccion[1])
    indice = 0
    for fitnessIndividuo in fitness[generacion]:
        if fitnessIndividuo[0] > fitness[generacion][seleccion[0][0]][0]:
            seleccion[0][1] = copy.copy(seleccion[0][0])
            seleccion[0][0] = copy.copy(indice)
        elif fitnessIndividuo[0] > fitness[generacion][seleccion[0][1]][0]:
            seleccion[0][1] = copy.copy(indice)
        if fitnessIndividuo[1] > fitness[generacion][seleccion[1][0]][1]:
            seleccion[1][1] = copy.copy(seleccion[0][0])
            seleccion[1][0] = copy.copy(indice)
        elif fitnessIndividuo[1] > fitness[generacion][seleccion[1][1]][1]:
            seleccion[1][1] = copy.copy(indice)
        indice += 1

    seleccion[0][0] = copy.copy(NN1[seleccion[0][0]])
    seleccion[0][1] = copy.copy(NN1[seleccion[0][1]])
    seleccion[1][0] = copy.copy(NN2[seleccion[1][0]])
    seleccion[1][1] = copy.copy(NN2[seleccion[1][1]])

    NN1[0] = copy.copy(seleccion[0][0])
    NN1[1] = copy.copy(seleccion[0][1])
    NN2[0] = copy.copy(seleccion[1][0])
    NN2[1] = copy.copy(seleccion[1][1])

    for individuo in range(2, len(NN1)):
        if individuo % 2 == 0:
            NN1[individuo] = copy.copy(seleccion[0][0])
            NN2[individuo] = copy.copy(seleccion[1][0])
        else:
            NN1[individuo] = copy.copy(seleccion[0][1])
            NN2[individuo] = copy.copy(seleccion[1][1])

        NN1[individuo].mutarCapas(coeficienteMutacion)
        NN2[individuo].mutarCapas(coeficienteMutacion)

NN1[0].guardarModelo('02_04_01')
NN2[0].guardarModelo('02_04_02')
```

```
import Sim
import Clases
import copy
import RedNeuronal
import Utilities as Ut

# Modelo02_02 control de bifurcaciones - Genetico Colas
individuosPorGeneracion = 6
generaciones = 10
tiempoSimulacion = 2000
coeficienteProduccion = 0.8
coeficienteColas = 0.2

NN1 = []
NN2 = []
resultados = 0
fitness = [0, 0]
coste1 = 0
coste2 = 0

simulador = Sim.Simulador02(logs=True)

colas1 = [[11, 12], [6, 7, 19, 20, 21, 22, 23, 32, 33], [8, 13, 14], [15, 25,
26, 27], [16, 17, 18], [29, 30]]
colas2 = [[11, 12], [6, 7, 19, 20, 21, 22, 23, 32, 33], [8, 13, 14], [15, 25,
26, 27], [16, 17, 18], [29, 30]]

# 0 -> % Ocupacion de tipo 0
# 1 -> % Ocupacion de tipo 1
################ 01 ## 02 ## 03 ## 04 ## 05 ## 06 ##
################0  1  0  1  0  1  0  1  0  1  0  1######
dataset01 = [
            [[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], 0],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 1],
            [[0, 0, 0, 0, 0, 0, 0.7, 0, 0, 0, 0, 0], 0],
            [[0, 0, 0, 0, 0, 0, 0.3, 0, 0, 0, 0, 0], 1]
            ]

dataset02 = [
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0], 0],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 1],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.7, 0], 0],
            [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.3, 0], 1]
            ]

NN1 = RedNeuronal.NN02_03_01(colas1)
NN2 = RedNeuronal.NN02_03_02(colas2)

NN1.cargarModelo('02_03_01_v2')
NN2.cargarModelo('02_03_02_v2')

for epoch in range(0, 25000):
    if epoch % 5000 == 0:
```

```
            Ut.pNoEs('Epoch ')
            print(epoch)
        if epoch % 100 == 99:
            Ut.pNoEs('Error 1: ')
            try: Ut.pNoEs(coste1.item())
            except: pass
            Ut.pNoEs('Error 2: ')
            try: print(coste2.item())
            except: pass

        for sample in dataset01:
            NN1.setEntrada([sample[0]])
            NN1.calculaSalida()
            coste1 = NN1.entrenar(sample[1], 0.1)

        for sample in dataset02:
            NN2.setEntrada([sample[0]])
            NN2.calculaSalida()
            coste2 = NN2.entrenar(sample[1], 0.1)


simulador = Sim.Simulador02(logs=True)
resultados = simulador.simular(NN1, NN2, tiempoSimulacion)

fitness[0] = NN1.fitness(resultados, coeficienteProduccion, coeficienteColas)
fitness[1] = NN2.fitness(resultados, coeficienteProduccion, coeficienteColas)
print(resultados.produccion)
print(fitness)


NN1.guardarModelo('02_03_01_v3')
NN2.guardarModelo('02_03_02_v3')
```

4 Node-RED Flows




Query
Retenedores
Activar/Desactivar Bloqueo un retenedor
retenedores/forzar-estado
DIR
Query
Retenedores
Set MongoDB path
msg.payload
msg.payload
Set mqtt path
join-wait
Activar/Desactivar bloqueos
Volcado DIR array