```
classdef MyMultiPerceptron < handle
    properties
        weights
        gamma
        fi
        fa
        fd
        fal
        fdl
        arq
        mode
        momentum
    end

    methods
        function this = MyMultiPerceptron(arq, gamma, mode, fi, fa, fd, fal, fdl)
            if nargin < 3
                mode = 'bipolar';
            end

            % fi y fa son funciones lambda.
            % * fi es utilizada para la inicializacion y se le pasa por
            %   parametro un valor proveniente de una normal [0,1]
            % * fa es la funcion de activacion de las neuronas
            % * fd es la derivada de la funcion de activacion
            %fa = @(t) (t > 45) * 1 + (-45 < t && t < 45) * (1/(1+exp(1)^(-t)));
            if not(strcmp(mode, 'custom'))
                switch mode
                    case 'binary-regresion'
                        fi = @(x) x; % entre [0,1]
                        fa = @(t) 1./(1+exp(-t));
                        fd = @(t) t.*(1-t); %fd = @(t) fa(t).*(1-fa(t));
                        fal = @(t) t;
                        fdl = @(t) ones(size(t));
                    case 'binary'
                        fi = @(x) x; % entre [0,1]
                        fa = @(t) 1./(1+exp(-t));
                        fd = @(t) t.*(1-t); %fd = @(t) fa(t).*(1-fa(t));
                        fal = fa;
                        fdl = fd;
                    case 'bipolar-regresion'
                        fi = @(x) x*2-1; % entre [-1,1]
                        fa = @(t) tanh(t);
                        fd = @(t) 1-(tanh(t).^2);
                        fal = @(t) t;
                        fdl = @(t) ones(size(t));
                    otherwise % bipolar
                        fi = @(x) x*2-1; % entre [-1,1]
                        fa = @(t) tanh(t);
                        fd = @(t) 1-(tanh(t).^2);
                        fal = fa;
                        fdl = fd;
                end
            end

            this.gamma = gamma;
            this.fi = fi;
            this.fa = fa;
            this.fd = fd;
            this.fal = fal;
            this.fdl = fdl;
            this.arq = arq;
            this.mode = mode;
            this.momentum = 0;

            this.initWeights(arq);
        end



        function initWeights(this, arq)
            % Construye las matrices de pesos relacionando
            % cada capa con la siguiente.
            for i = 1:(size(arq,2)-1)
                % Aplico la funcion de inicializacion
                %this.weights{i} = 0.1*rand(arq(i)+1, arq(i+1))*2-1;
                this.weights{i} = randn(arq(i)+1, arq(i+1));
            end
        end

        % x debe ser de tamanio size(this.layers{1},1)(n) y vector fila
        function y = feedForward(this, x)
            Y = this.propagateFeed(x);
            y = Y{length(Y)};
        end

        function [Y] = propagateFeed(this, xs)
            % Incializamos la cell para guardar los datos
            Y = cell(length(this.weights)+1,1);
            Y{1} = xs;

            % Aplica activacion al resultado de Y*W
            for i = 1:length(this.weights)
                if i == length(this.weights)
                    Y{i+1} = this.fal([Y{i} 1] * this.weights{i});
                else
                    Y{i+1} = this.fa([Y{i} 1] * this.weights{i});
                end
            end
        end

        % xs es una matriz en donde cada fila es un input
        % zs es una matriz en donde cada fila coincide con el resultado
        function [ep_errors] = train(this, xs, zs, min_error, max_epoch, momentum)
            if nargin > 5
                this.momentum = momentum;
            end

            total_run_error = min_error + 1;
            ntrain = length(xs);
            ep_errors = [];
            epoch = 0;

            last_ldeltas = cell(length(this.weights),1);
            [last_ldeltas{:}] = deal(0);

            while (total_run_error > min_error) && (epoch < max_epoch)
                order = randperm(ntrain);
                total_run_error = 0;
                epoch = epoch + 1;
                for j = 1:ntrain
                    Y = this.propagateFeed(xs(order(j),:));
                    [run_error, ldeltas] = this.correction(Y, zs(order(j),:));
                    this.adaptation(ldeltas, last_ldeltas);
                    total_run_error = total_run_error + run_error;
                    if this.momentum ~= 0
                        last_ldeltas = ldeltas;
                    end
                end
                ep_errors = [ep_errors total_run_error];
            end

            ep_errors = ep_errors / size(xs,1);
        end





        function [output_error, ldeltas] = correction(this, Y, target)

            % Guardamos las diferencias a aplicar en cada nivel
            ldeltas = cell(length(this.weights),1);

            % Propagamos el error y tomamos nota de las deferencias
            % Guarda el error en cada loop
            run_error = (target - Y{end});
            output_error = sum(abs(run_error))/length(run_error);
            for i = length(this.weights):-1:1

                if i == length(this.weights)
                    run_error = run_error .* this.fdl(Y{i+1});
                else
                    run_error = run_error .* this.fd(Y{i+1});
                end

                ldeltas{i} = this.gamma * [Y{i} 1]' * run_error;

                % Dejamos afuera la ultima fila ya que es la que tiene los bias
                run_error = run_error * (this.weights{i}(1:end-1,:))';
            end
        end

        function adaptation(this, ldeltas, last_ldeltas)
            for i = 1:length(this.weights)
                this.weights{i} = this.weights{i} + ldeltas{i};
                if this.momentum ~= 0
                    this.weights{i} = this.weights{i} + this.momentum * last_ldeltas{i};
                end
            end
        end

    end
end
```