Angel Marcelino Gonzalez Mayoral

Código 218292998

Inteligencia Artificial II 2022A

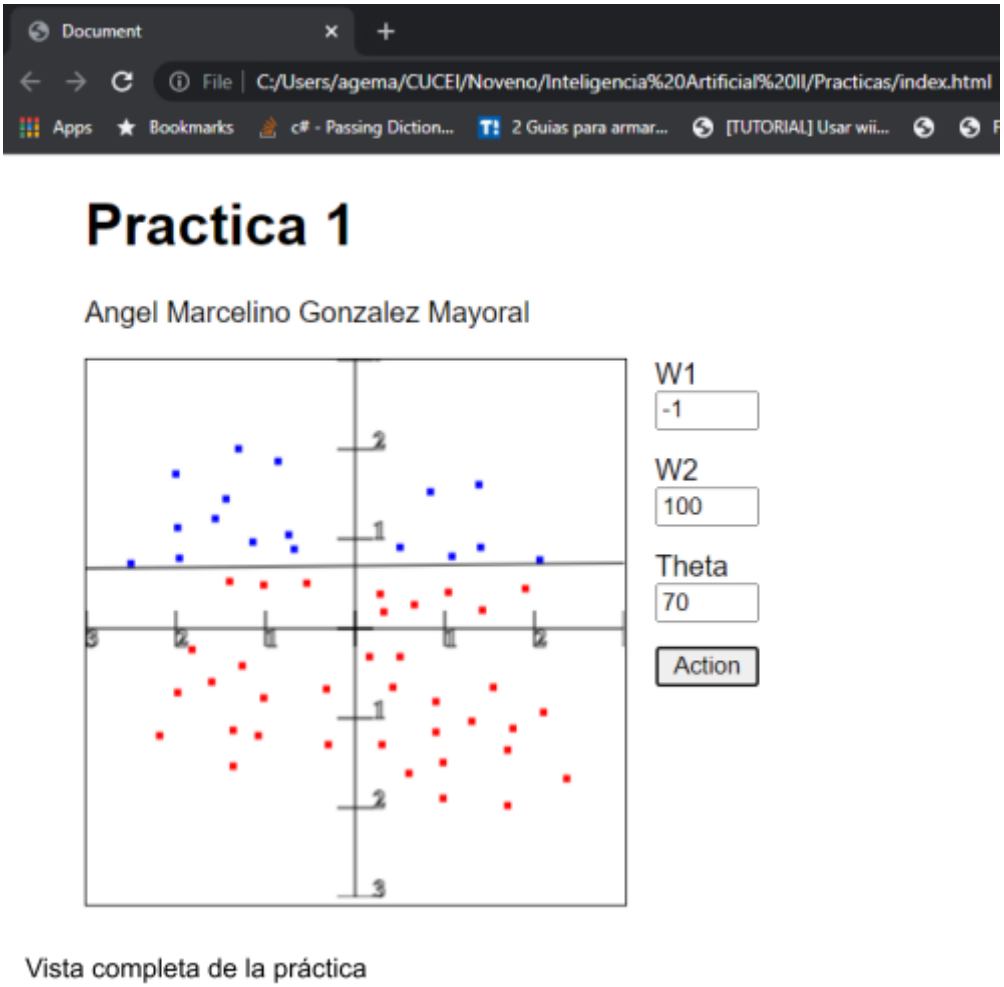
Práctica 1

Introducción

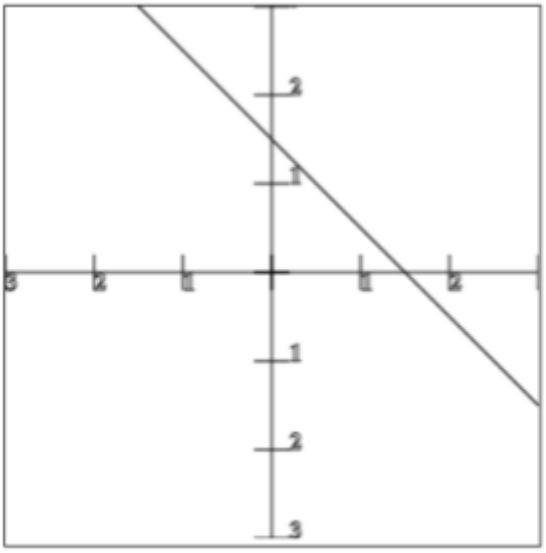
Una neurona de dos entradas puede representarse como una línea recta en un plano. Recordemos que la ecuación de la línea recta se describe con la siguiente ecuación. y = Ax + B en donde A es la pendiente y B es el desplazamiento vertical. Pues bien, dados los pesos (W1, W2) y el umbral (T) de una neurona de dos entradas (X1, X2) la ecuación de la línea que la describe es X2 = -W1/W2 * X1 + T/W2 en donde la pendiente y el desplazamiento vertical vienen dadas por -W1/W2 y el desplazamiento por T/W2.

En esta práctica graficamos la línea que representa a la neurona en un plano. Se graficará en base a los valores de los pesos W1, W2 y al valor del umbral. También se podrán posicionar puntos en el plano los cuales serán clasificados dependiendo de los valores de los pesos y del umbral.

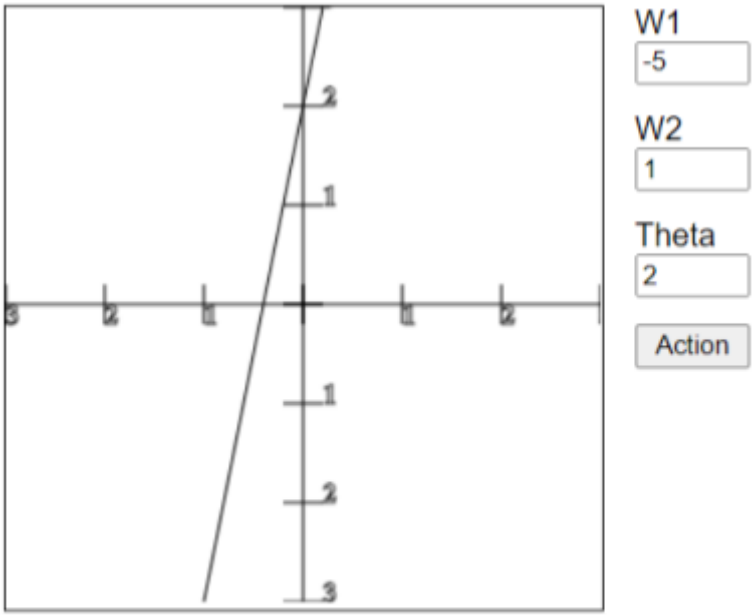
Capturas



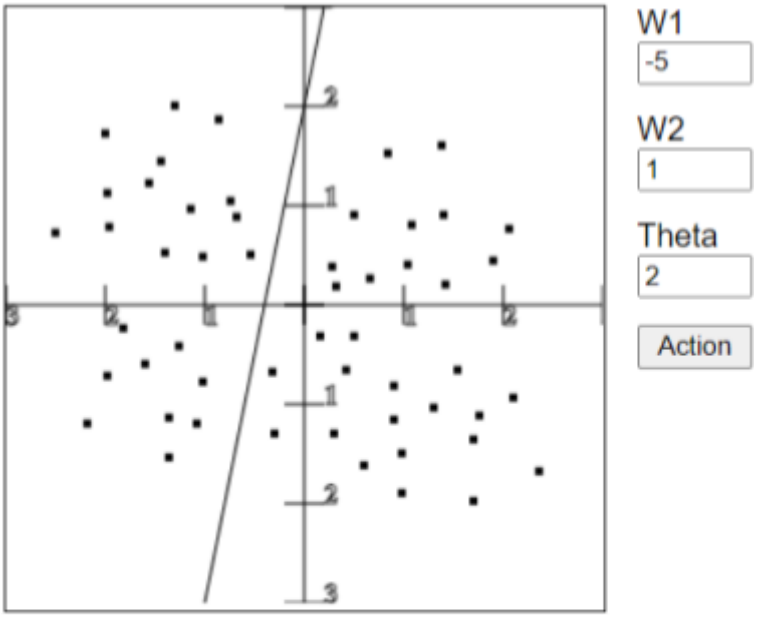
Vista completa de la práctica



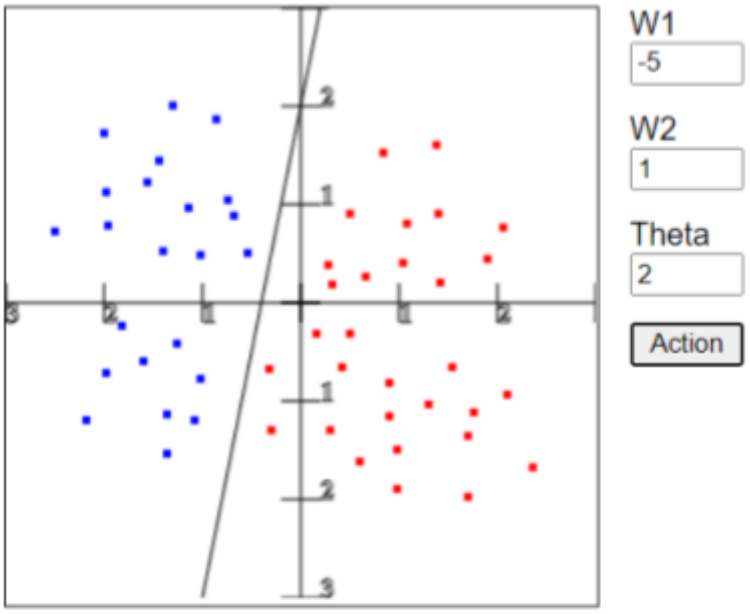
Configuración por defecto de los pesos.



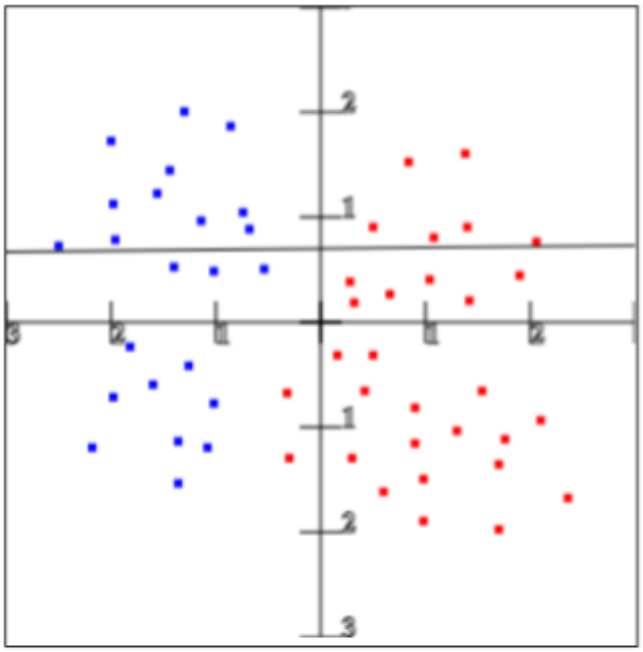
En esta imagen se puede observar cómo al cambiar los valores de los pesos y del umbral la línea actualiza su posición.



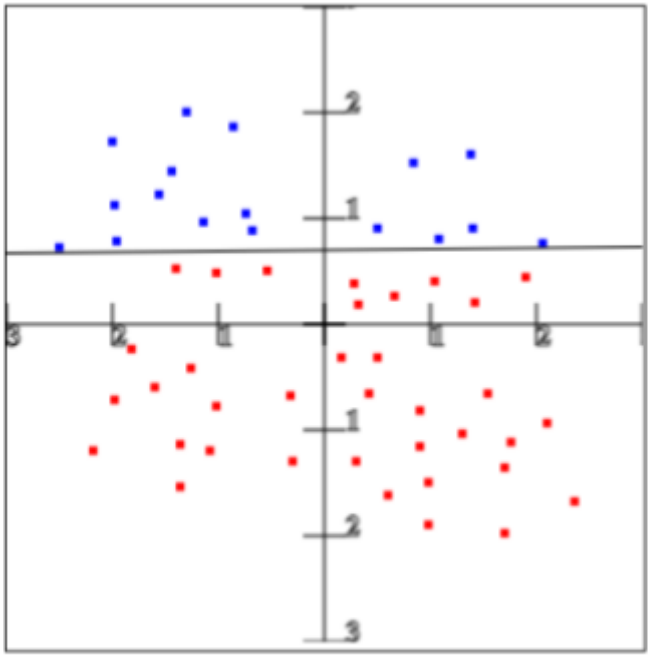
Es posible graficar puntos en el plano haciendo click en el lugar del plano en donde se quiera establecer el punto. Los puntos recién agregados se dibujarán de color negro lo que significa que carecen de clase.



Después de hacer clic en el botón Action se clasifica cada punto gráficamente dependiendo de su posición con respecto a la línea



Es posible cambiar los pesos y el umbral para actualizar la posición de la línea



Será necesario hacer clic de nuevo en action para volver a clasificar los puntos

Código fuente

```
//renderable.js
class Renderable {
  /**
   * @type { () => void}
   */
  #reRenderCallback;
  /**
   *
   * @param {() => void)} reRenderCallback
   */
  constructor(reRenderCallback) {
    this.#reRenderCallback = reRenderCallback;
  }
  /**
   * @type {() => void}
   * @param {() => void} value
   */
  set reRenderCallback(value) {
    this.#reRenderCallback = value;
  }
  get reRenderCallback() {
    return this.#reRenderCallback;
  }
  reRender() {
    if (this.#reRenderCallback) {
      this.#reRenderCallback();
    }
  }
}
//line.js
class Line extends Renderable {
  #w1;
  #w2;
  #threshold;
  constructor(w1, w2, threshold, reRenderCallback) {
    super(reRenderCallback);
    this.w1 = w1;
    this.w2 = w2;
    this.threshold = threshold;
  }
  set w1(value) {
    this.#w1 = value;
    this.reRender();
  }
  get w1() {
```

```
    return this.#w1;
  }
  get m() {
    let m = -this.#w1 / this.#w2;
    return m;
  }
  set w2(value) {
    this.#w2 = value;
    this.reRender();
  }
  get w2() {
    return this.#w2;
  }
  set threshold(value) {
    this.#threshold = value;
  }
  set threshold(value) {
    this.#threshold = value;
    this.reRender();
  }
  get threshold() {
    return this.#threshold;
  }
  get b() {
    let b = this.#threshold / this.#w2;
    return b;
  }
}
//point.js
class Point extends Renderable {
  #clas;
  #x;
  #y;
  /**
   *
   * @param {number} x
   * @param {number} y
   * @param {string} clas
   * @param {() => void} reRenderCallback
   */
  constructor(x, y, clas, reRenderCallback) {
    super(reRenderCallback);
    this.x = x;
    this.y = y;
    this.clas = clas;
  }
  set clas(clas) {
    this.#clas = clas;
```

```
    this.reRender();
  }
  get clas() {
    return this.#clas;
  }
  set x(value) {
    this.#x = value;
    this.reRender();
  }
  get x() {
    return this.#x;
  }
  set y(value) {
    this.#y = value;
    this.reRender();
  }
  get y() {
    return this.#y;
  }
}
//environment.js
class Environment extends Renderable {
  /**
   * @type { Line }
   */
  #currentLine;
  /**
   * @type { Point[] }
   */
  #points;
  #scale;
  constructor(reRenderCallback, scale) {
    super(reRenderCallback);
    this.#currentLine = null;
    this.#points = [];
    this.#scale = scale;
  }
  set line(value) {
    this.#currentLine = value;
    this.reRender();
  }
  get line() {
    return this.#currentLine;
  }
  /**
   * @param {Point} point
   */
  addPoint(point) {
```

```
    point.reRenderCallback = super.reRenderCallback;
    this.#points.push(point);
    this.reRender();
  }
  removePointAt(index) {
    this.#points.splice(index, 1);
    this.reRender();
  }
  #cleanPoints() {
    this.#points = [];
    this.reRender();
  }
  get points() {
    return this.#points;
  }
  clean() {
    this.line = null;
    this.#cleanPoints();
  }

  classify() {
    this.#points.forEach((point) => {
      let clas = CLASS_A;
      let calculation =
        point.y - (point.x * this.line.m + this.line.b * this.#scale);
      if (calculation >= 0) {
        clas = CLASS_B;
      }
      point.clas = clas;
    });
  }

  /**
   * @param {any} value
   */
  set reRenderCallback(value) {
    this.#points.forEach((point) => (point.reRenderCallback = value));
    if (this.#currentLine) {
      this.#currentLine.reRenderCallback = value;
    }
    super.reRenderCallback = value;
  }
}
//environment-renderer.js
class EnvironmentRenderer {
  /** @type { CanvasRenderingContext2D } */
  #gContext;
  #scale;
```

```
/** @type {HTMLCanvasElement} */
#canvas;
#coordinatesTranslator;
#environment;

/**
 *
 * @param {HTMLCanvasElement} canvas
 * @param {CoordinatesTranslator} coordinatesTranslator
 * @param {Environment} environment
 */
constructor(canvas, coordinatesTranslator, environment, scale) {
  this.#canvas = canvas;
  this.#gContext = this.#gContext = canvas.getContext("2d");
  this.#clean();
  this.#scale = scale;
  this.#gContext.scale(this.#scale, this.#scale);
  this.#gContext.lineWidth = 1 / this.#scale;
  this.#coordinatesTranslator = coordinatesTranslator;
  this.#environment = environment;
  environment.reRenderCallback = this.render.bind(this);
  this.#drawAxisPlaceHolders();
}

/** @param { Environment } environment */
render() {
  this.#clean();
  this.#drawAxisPlaceHolders();
  this.#environment.points.forEach((point) => {
    this.#drawPoint(point);
  });
  if (this.#environment.line) {
    this.#drawLine(this.#environment.line);
  }
}

#clean() {
  this.#gContext.fillRect(0, 0, 30, 30);
  this.#gContext.fillStyle = "white";
  this.#gContext.fillRect(0, 0, canvas.width, canvas.height);
}

#preserverColor(fn) {
  let originalColor = context.fillStyle;
  fn();
  context.fillStyle = originalColor;
}
```

```
#drawAxisPlaceHolders() {
  this.#canvasLine(
    [this.#canvas.width / 2 / this.#scale, this.#canvas.height],
    [this.#canvas.width / 2 / this.#scale, 0]
  );
  this.#canvasLine(
    [this.#canvas.width, this.#canvas.height / 2 / this.#scale],
    [0, this.#canvas.height / 2 / this.#scale]
  );
  for (let i = 0; i <= this.#canvas.width; i += this.#scale) {
    let axisSide = [1, -1];
    axisSide.forEach((e) => {
      for (let j = 0; j < 2; j++) {
        let from = this.#coordinatesTranslator
          .mathToCanvas(-10, i * e);
        let to = this.#coordinatesTranslator
          .mathToCanvas(10, i * e);
        if (j == 0) {
          from.reverse();
          to.reverse();
        }
        this.#canvasLine(
          [from[0] / this.#scale, from[1] / this.#scale],
          [to[0] / this.#scale, to[1] / this.#scale]
        );
        this.#gContext.font = `bold ${14 / this.#scale}px serif`;
        if (i != 0) {
          this.#gContext.strokeText(
            "" + i / this.#scale,
            ...[to[0] / this.#scale, to[1] / this.#scale]
          );
        }
      }
    });
  }
}

/**
 *
 * @param {Line} line
 */
#drawLine(line) {
  let from = this.#coordinatesTranslator.mathToCanvas(
    -5000,
    -5000 * line.m + line.b * this.#scale
  );
  let to = this.#coordinatesTranslator.mathToCanvas(
    5000,
```

```
      5000 * line.m + line.b * this.#scale
    );
    this.#canvasLine(
      from.map((e) => e / this.#scale),
      to.map((e) => e / this.#scale)
    );
  }
  /**
   *
   * @param {[number, number]} from
   * @param {[number, number]} to
   */
  #canvasLine(from, to) {
    let [xFrom, yFrom] = from;
    let [xTo, yTo] = to;
    this.#gContext.beginPath();
    this.#gContext.moveTo(xFrom, yFrom);
    this.#gContext.lineTo(xTo, yTo);
    this.#gContext.stroke();
  }

  /**
   *
   * @param {Point} point
   */
  #drawPoint(point) {
    let [x, y] = this.#coordinatesTranslator
      .mathToCanvas(point.x, point.y);
    let pointWidth = 4;
    let pointHeight = 4;
    this.#preserverColor(() => {
      this.#gContext.fillStyle = point.clas;
      this.#gContext.fillRect(
        (x - pointWidth / 2) / this.#scale,
        (y - pointHeight / 2) / this.#scale,
        pointWidth / this.#scale,
        pointHeight / this.#scale
      );
    });
  }
}
//coordinates-translator.js
let CLASS_A = "#FF0000";
let CLASS_B = "#0000FF";
let CLASS_NONE = "#000000";
class CoordinatesTranslator {
  #width;
  #height;
```

```
  #yCenter;
  #xCenter;
  /**
   *
   * @param {number} width
   * @param {number} height
   */
  constructor(width, height) {
    this.#width = width;
    this.#height = height;
    this.#calculateCenter();
  }

  #calculateCenter() {
    this.#yCenter = this.#height / 2;
    this.#xCenter = this.#width / 2;
  }
  /**
   * @param {number} x
   * @param {number} y
   */
  canvasToMath(x, y) {
    return [x - this.#xCenter, this.#yCenter - y];
  }

  /**
   *
   * @param {number} x
   * @param {number} y
   * @returns {[number, number]}
   */
  mathToCanvas(x, y) {
    return [x + this.#xCenter, this.#yCenter - y];
  }
}
//script.js
"use strict";
/** @type { [
 * HTMLButtonElement,
 * HTMLInputElement,
 * HTMLInputElement,
 * HTMLInputElement,
 * HTMLCanvasElement]} */
let [button, w1Input, w2Input, thresholdInput, canvas] = [
  "actionButton",
  "w1",
  "w2",
  "threshold",
```

```
  "canvas",
].map((id) => document.getElementById(id));
let context = canvas.getContext("2d");
let scale = 50;
let environment = new Environment(context, scale);
let coordinatesTranslator = new CoordinatesTranslator(
  canvas.width,
  canvas.height
);
let environmentRenderer = new EnvironmentRenderer(
  canvas,
  coordinatesTranslator,
  environment,
  scale
);
setInterval(() => {
  console.log(environment.points);
}, 1000);

function mutateLine() {
  let line = new Line(w1Input.value, w2Input.value, thresholdInput.value);
  environment.line = line;
}
mutateLine();

[w1Input, w2Input, thresholdInput].forEach((input) => {
  input.addEventListener("change", () => {
    mutateLine();
  });
});

button.addEventListener("click", () => {
  environment.classify();
});

function getCursorPosition(canvas, event) {
  const rect = canvas.getBoundingClientRect();
  const x = event.clientX - rect.left;
  const y = event.clientY - rect.top;
  return [x, y];
}
canvas.addEventListener("click", (event) => {
  let [xC, yC] = getCursorPosition(canvas, event);
  let [x, y] = coordinatesTranslator.canvasToMath(xC, yC);
  let point = new Point(x, y, CLASS_NONE, null);
  environment.addPoint(point);
});
<!-- index.html -->
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link href="style.css" rel="stylesheet" />
  </head>
  <body>
    <h1>Práctica 1</h1>
    <p>Angel Marcelino Gonzalez Mayoral</p>
    <div class="container">
      <div class="canvas-container">
        <canvas
          id="canvas"
          height="300px"
          width="300px">
        </canvas>
      </div>
      <div class="input-container">
        <label for="w1">W1</label>
        <input id="w1" type="number" value="1" />
        <label for="w2">W2</label>
        <input id="w2" type="number" value="1" />
        <label for="threshold">Theta</label>
        <input id="threshold" type="number" value="1.5" />
        <button type="button" id="actionButton">
          Action
        </button>
      </div>
    </div>
    <script src="coordinates-translator.js"></script>
    <script src="environment-renderer.js"></script>
    <script src="renderable.js"></script>
    <script src="point.js"></script>
    <script src="line.js"></script>
    <script src="environment.js"></script>
    <script src="script.js"></script>
  </body>
</html>
/* style.css */
body {
  font-family: sans-serif;
  margin: 0 3em;
}
.canvas-container {
```

```
  border: 1px solid #000;
}
.container {
  display: flex;
}
.input-container {
  display: flex;
  flex-direction: column;
  margin-left: 1em;
}
.input-container > input {
  margin-bottom: 1em;
  width: 50px;
}
```

Conclusiones

Gracias a esta práctica es claro ver cómo es que una sola neurona clasifica. En caso de tener más entradas solo hay que extrapolar la solución a más dimensiones. El hecho de poder observar cómo es que se modifica la línea que representa la neurona según se mueven los parámetros ayuda a comprender de mejor forma el funcionamiento de la neurona.

Creo que esta práctica sirve para reafirmar los conocimientos de las primeras clases. Personalmente diseñe la práctica pensando en extenderla para utilizarla de base fácilmente en las prácticas que se avecinan.