Angel Marcelino Gonzalez Mayoral

Código 218292998

Inteligencia Artificial II 2022A

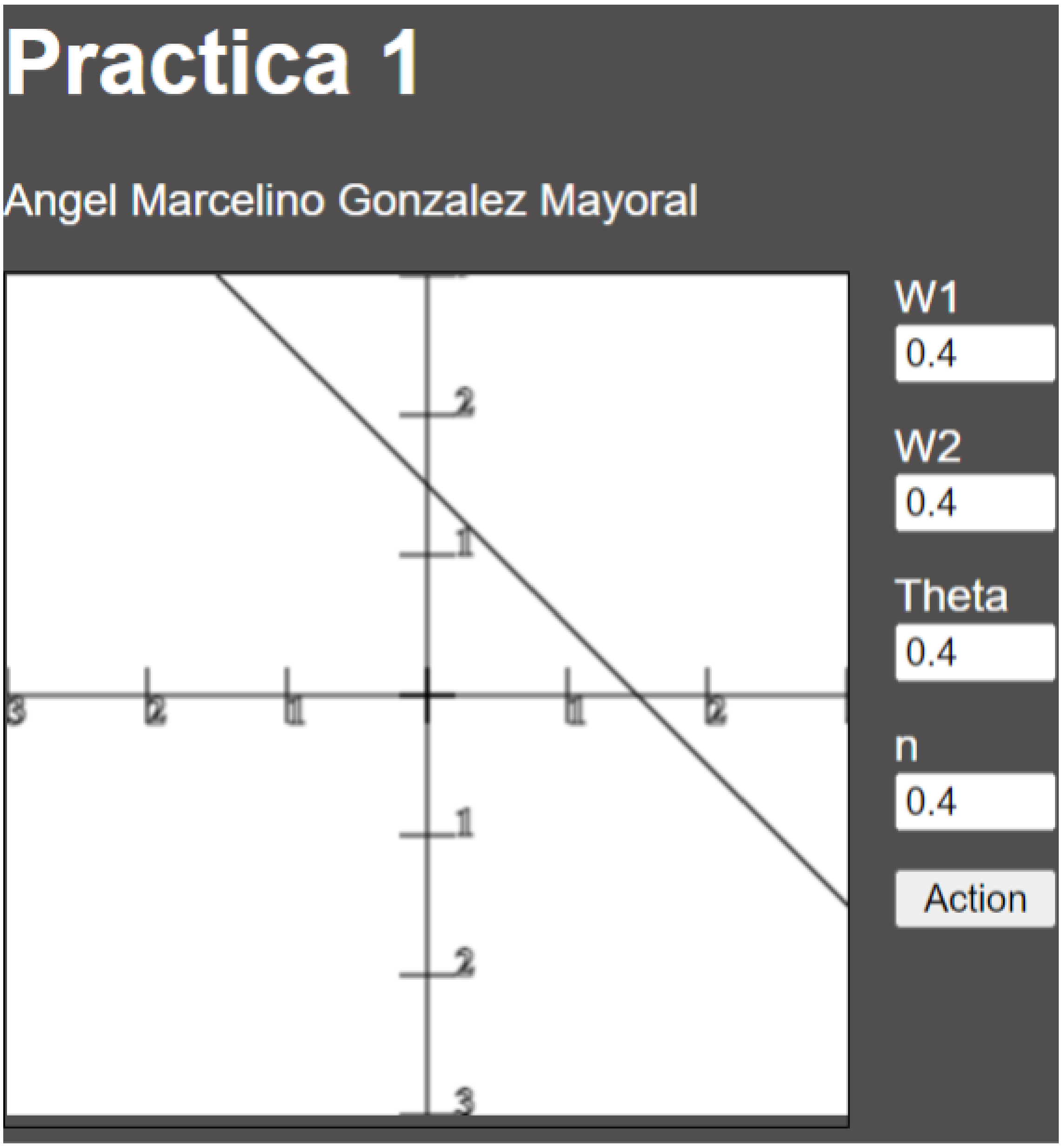
Práctica 1

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 entrenaremos al perceptrón para poder clasificar puntos en un plano. Lo primero será introducir los puntos de entrenamiento. Estos puntos tendrán asignada una clase desde el momento que son creados. Esto se logra creando los puntos haciendo clic con el botón derecho e izquierdo del ratón. Posterior al ingreso de los puntos de entrenamiento se deberá hacer clic en el botón de entrenamiento para que se empiece a ejecutar el algoritmo de entrenamiento por corrección del error.

Capturas


Practica 1

Angel Marcelino Gonzalez Mayoral

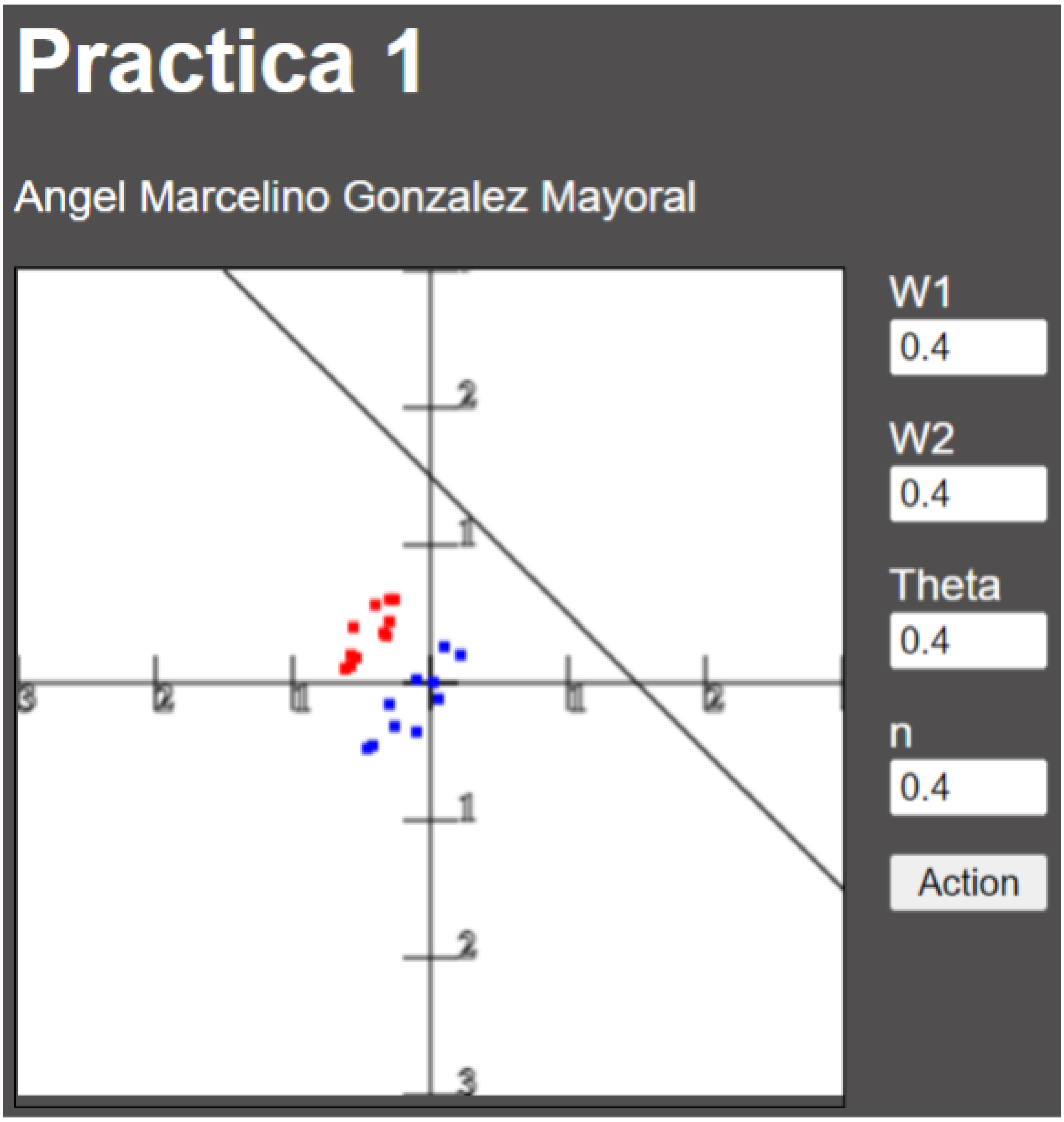
W1
0.4

W2
0.4

Theta
0.4

n
0.4

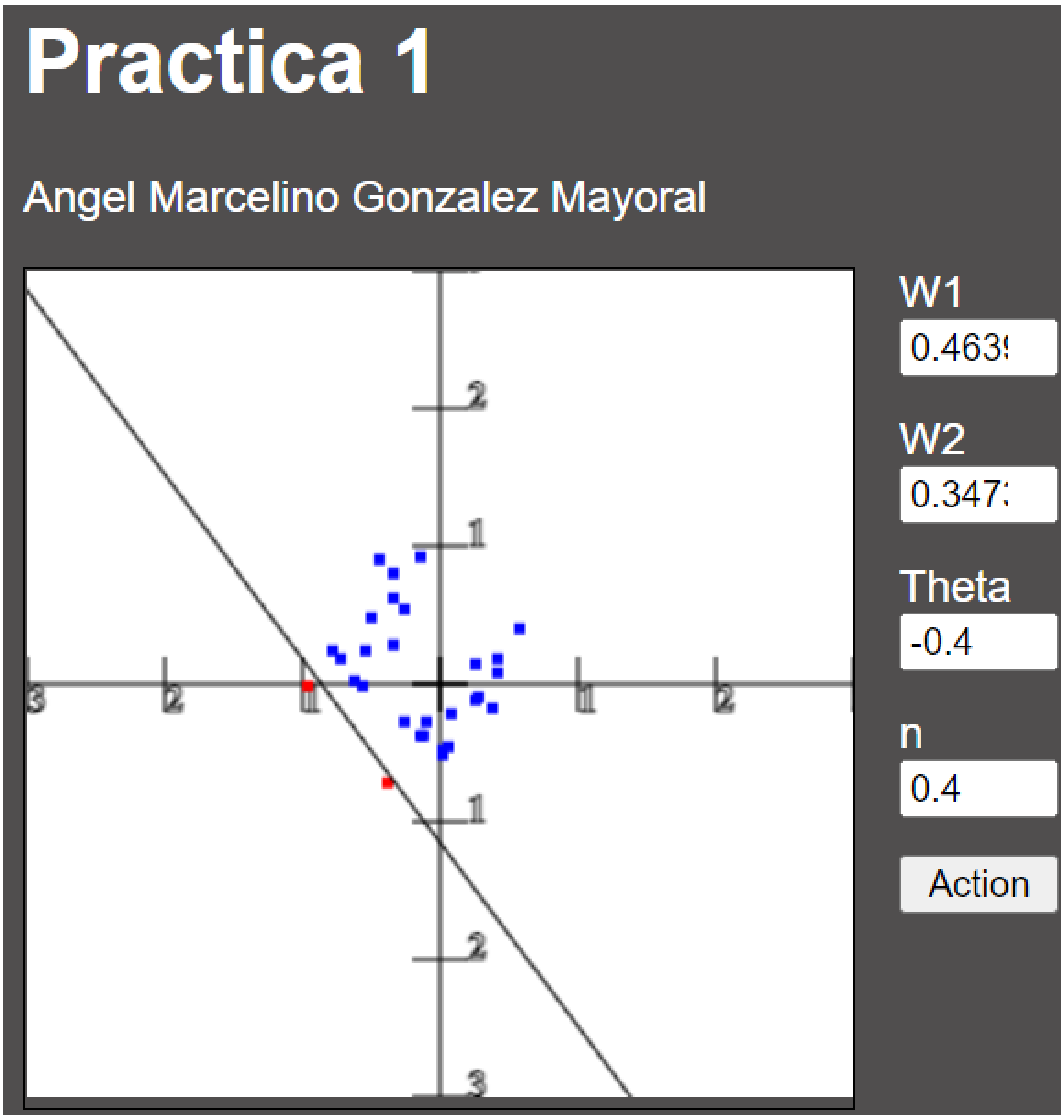
Inicio de la práctica


Practica 1

Angel Marcelino Gonzalez Mayoral

Se pueden colocar puntos de diferentes clases haciendo clic izquierdo o clic derecho sobre el plano

Inicia el proceso de entrenamiento. Las siguientes imágenes muestra cómo se va moviendo la línea en cada iteración del algoritmo de entrenamiento


Practica 1

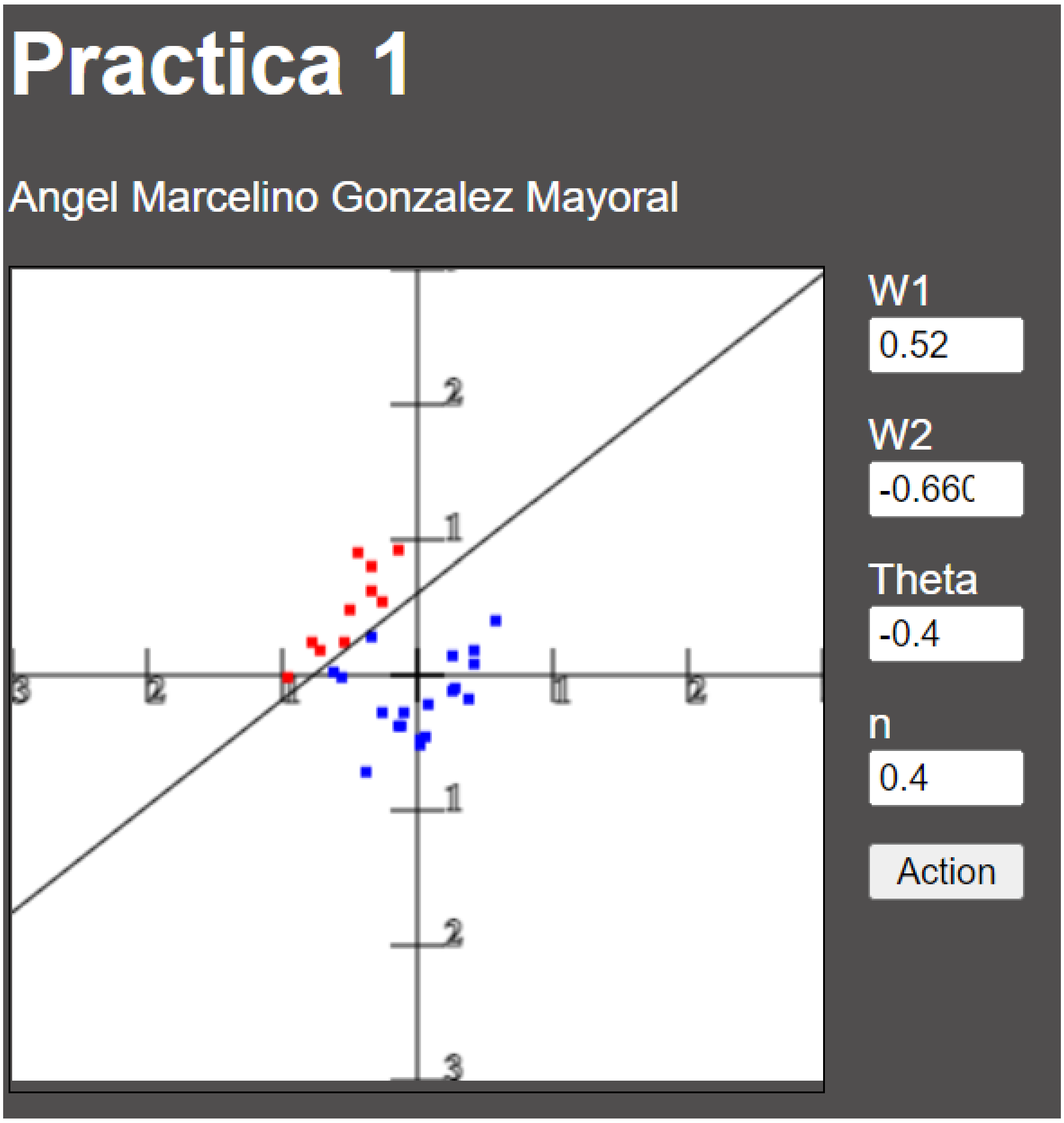
Angel Marcelino Gonzalez Mayoral

W1 0.4639

W2 0.3473

Theta -0.4

n 0.4



Practica 1

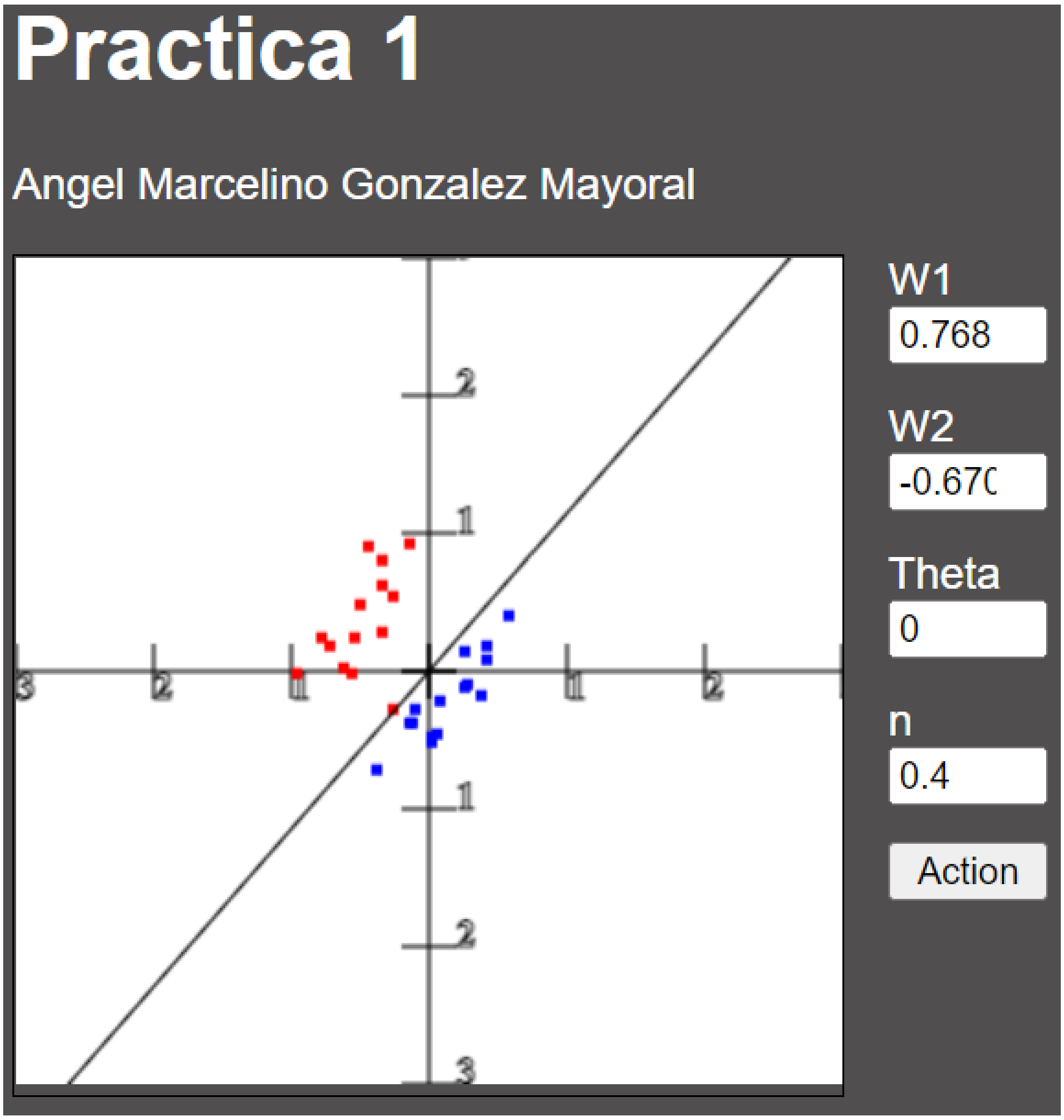
Angel Marcelino Gonzalez Mayoral

W1
0.52

W2
-0.66(

Theta
-0.4

n
0.4



Practica 1

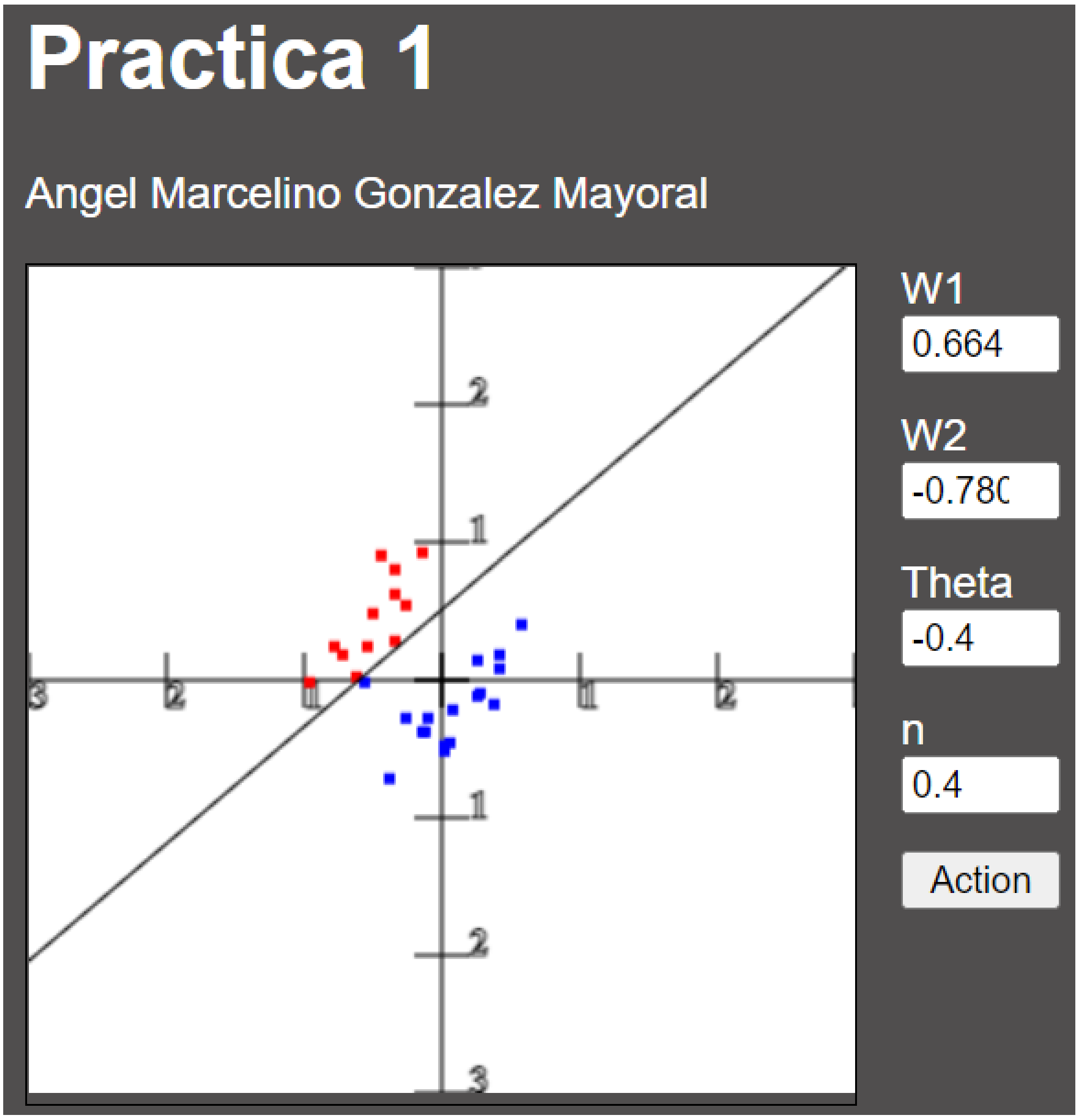
Angel Marcelino Gonzalez Mayoral

W1
0.768

W2
-0.670

Theta
0

n
0.4



Practica 1

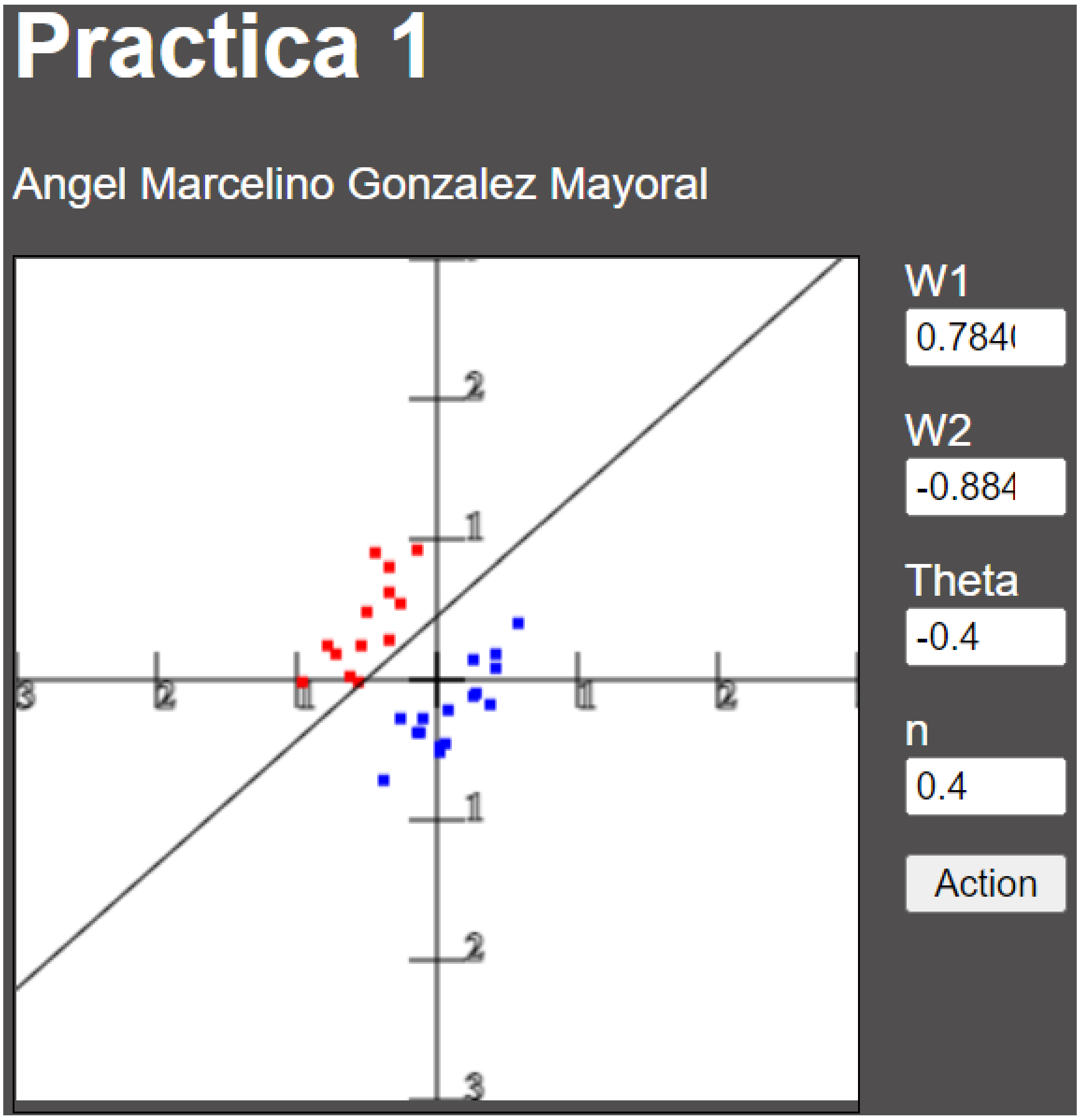
Angel Marcelino Gonzalez Mayoral

W1
0.664

W2
-0.780

Theta
-0.4

n
0.4



Practica 1

Angel Marcelino Gonzalez Mayoral

W1
0.784

W2
-0.884

Theta
-0.4

n
0.4

Termina el algoritmo de entrenamiento

Código fuente

```
//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];
  }
```

```
}

//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
      );
    });
  }
}

//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();
  }
  /**
   * @type {Line}
   */
  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 current = new Point(
        point.x / this.#scale,
        point.y / this.#scale,
        point.clas
      );
      let neuronCalculator = new NeuronCalculator();
      let calculation = neuronCalculator.calculateNeuronOuput(current,
[
        -this.#currentLine.threshold,
        this.#currentLine.w1,
        this.#currentLine.w2,
      ]);
      let clas;
      if (calculation[0] == 0) {
        clas = CLASS_A;
      } else {
        clas = CLASS_B;
      }
      point.clas = clas;
    });
    this.reRender();
  }

  /**
   * @param {any} value
   */
  set reRenderCallback(value) {
    this.#points.forEach((point) => (point.reRenderCallback = value));
    if (this.#currentLine) {
      this.#currentLine.reRenderCallback = value;
    }
    super.reRenderCallback = value;
  }

  #disableRenderPoints() {
    this.#points.forEach((p) => (p.reRenderCallback = null));
  }
  #enableRenderPoints() {
    this.#points.forEach((p) => (p.reRenderCallback =
super.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;
  }
}
//neuron-calculator.js
class NeuronCalculator {
  /**
   *
   * @param {Point} point
   * @param {number[]} weights
   * @param {number} bias
   */
  calculateNeuronOuput(point, weights, verbose = false) {
    let result = this.#calculateV(point, weights, verbose);
    let v = result;
    if (v < 0) {
      return [0, v];
    } else {
      return [1, v];
    }
  }

  /**
   *
   * @param {Point} point
   * @param {number[]} weights
   * @returns
   */
  #calculateV(point, weights, verbose = false) {
    let pointPropertyNumberMap = {
      0: 1,
      1: point.x,
      2: point.y,
    };
    if (verbose) {
      console.log(`point (${point.x}, ${point.y})`);
    }
    let result = weights.reduce((accum, current, i) => {
      let result = accum + current * pointPropertyNumberMap[i];
      if (verbose) {
        console.log(
          `${accum} + ${current} * ${pointPropertyNumberMap[i]} =
${result}`
        );
      }
      return result;
    }, 0);
    if (verbose) {
      console.log(`v = ${result}`);
    }
    return result;
  }
}

//point.js
class Point extends Renderable {
  #clas;
  #showClas;
  #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;
    this.showClas = clas;
  }
  set clas(clas) {
    this.#clas = clas;
    this.reRender();
  }
  get clas() {
    return this.#clas;
  }
  get showClas() {
    return this.#showClas;
  }
  set showClas(value) {
    this.#showClas = value;
    this.reRender();
  }
  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;
  }
}

//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();
    }
  }
}
//script.js
"use strict";
/** @type { [
 * HTMLButtonElement,
 * HTMLInputElement,
 * HTMLInputElement,
 * HTMLInputElement,
 * HTMLInputElement,
 * HTMLCanvasElement]} */
let [button, w1Input, w2Input, thresholdInput, nInput, canvas] = [
  "actionButton",
  "w1",
  "w2",
  "threshold",
  "n",
  "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 values = [-0.4, 0.4, 0.4];
let environmentRenderer = new EnvironmentRenderer(
  canvas,
  coordinatesTranslator,
  environment,
  scale
);

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();
  });
});
setValues();
let isWorking = false;
button.addEventListener("click", async () => {
  if (!isWorking) {
    isWorking = true;
    let trainer = new Trainer(
      nInput.value,
      1000,
      (newWeights) => {
        values = [...newWeights];
        setValues();
        mutateLine();
        environment.classify();
      },
      () => {
        isWorking = false;
      },
      scale
    );
    await trainer.train(environment.points, values);
  }
});

// setInterval(() => {
//   environment.classify();
// }, 1000);

function setValues() {
  thresholdInput.value = -values[0];
  w1Input.value = values[1];
  w2Input.value = values[2];
}

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) => {
  if (!isWorking) {
    putPoint(event, CLASS_A);
  }
});
canvas.addEventListener("contextmenu", (event) => {
  if (!isWorking) {
    event.preventDefault();
    putPoint(event, CLASS_B);
  }
});
function putPoint(event, classe) {
  let [xC, yC] = getCursorPosition(canvas, event);
  let [x, y] = coordinatesTranslator.canvasToMath(xC, yC);
  let point = new Point(x, y, classe, null);
  environment.addPoint(point);
}
//trainer.js
const interval = 20;
```

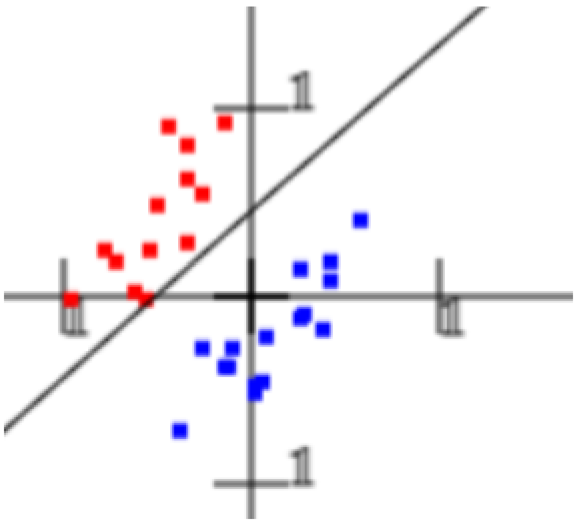
```
class Trainer {
  #n;
  #epoch;
  #trainerCallback;
  #endCallback;
  #neuronCalculator;
  #scale;
  /**
   *
   * @param {number} n
   * @param {number} epoch
   */
  constructor(n, epoch, trainerCallback, endCallback, scale) {
    this.#n = n;
    this.#epoch = 10000000;
    this.#trainerCallback = trainerCallback;
    this.#endCallback = endCallback;
    this.#neuronCalculator = new NeuronCalculator();
    this.#scale = scale;
  }

  /**
   *
   * @param {Point[]} dataSet
   * @param {number[]} startWeights
   * @param {number} startBias
   */
  async train(dataSet, startWeights) {
    let currentWeights = startWeights;
    let classNumberMap = {
      [CLASS_A]: 0,
      [CLASS_B]: 1,
    };
    let nOfErrors = 0;
    let epoch = 0;
    do {
      nOfErrors = 0;
      await this.#iterateWithPause((i) => {
        let current = new Point(
          dataSet[i].x / this.#scale,
          dataSet[i].y / this.#scale,
          dataSet[i].showClas
        );
        let [output, v] = this.#neuronCalculator.calculateNeuronOuput(
          current,
          currentWeights
        );
        let error = classNumberMap[current.showClas] - output;
        if (error !== 0) {
          let newWeights = this.#modifyWeight(
            [1, current.x, current.y],
            currentWeights,
            error
          );
          currentWeights = newWeights;
          this.#trainerCallback(newWeights);
          nOfErrors++;
        }
      }, dataSet.length);
      epoch++;
    } while (nOfErrors > 0);
    this.#endCallback();
    for (let i = 0; i < dataSet.length; i++) {
      let current = dataSet[i];
      console.log(
        this.#neuronCalculator.calculateNeuronOuput(
          current,
          currentWeights,
          true
        )
      );
      console.log(current, currentWeights);
    }
    return currentWeights;
  }
  async #iterateWithPause(fn, times) {
    for (let i = 0; i < times; i++) {
      await this.#wait(() => fn(i));
    }
  }
  #wait(fn) {
    return new Promise((resolve, reject) => {
      try {
        setTimeout(() => {
          fn();
          resolve();
        }, interval);
      } catch (ex) {
        reject(ex);
      }
    });
  }

  /**
   *
   * @param {number[]} inputs
   * @param {number[]} currentWeights
   * @param {number} bias
   * @param {number} error
   */
  #modifyWeight(inputs, currentWeights, error) {
    let newWeights = currentWeights.map((weight, index) =>
      this.#getNewValue.bind(this)(weight, inputs[index], error)
    );
    return newWeights;
  }

  /**
   *
   * @param {number} currentValue
   * @param {number} input
   * @param {number} error
   * @returns
   */
  #getNewValue(currentValue, input, error) {
    let newValue = currentValue + this.#n * error * input;
    return newValue;
  }
}

/* style.css */
body {
  font-family: sans-serif;
  background-color: rgb(80, 78, 78);
  color: white;
  margin: 0 3em;
}
.canvas-container {
  border: 1px solid #000;
```

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

```css
/* style.css */
body {
  font-family: sans-serif;
  background-color: rgb(80, 78, 78);
  color: white;
  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

Esta práctica fue aún más interesante que la anterior ya que en esta se pudo lograr el aprendizaje del perceptrón. La forma en que se ajustan los pesos va en función de los errores que se detectan, es por eso que siempre se llega a una respuesta (Siempre y cuando los valores sean divisibles por una línea). Hay casos en los que la línea queda muy cercana a un punto extremo, es decir no queda justamente a la mitad separando los puntos. Esto para mi, un ser inteligente es extraño ya que si me pusieran manualmente particionar con una línea recta los puntos haría una línea cuya distancia a los dos cluster de puntos sea más o menos igual. Esto no fue así en esta práctica.



En la figura anterior podemos observar cómo la línea se encuentra mucho más cerca del cluster de los puntos clase rojo. yo haría algo así