Angel Marcelino Gonzalez Mayoral

Código 218292998

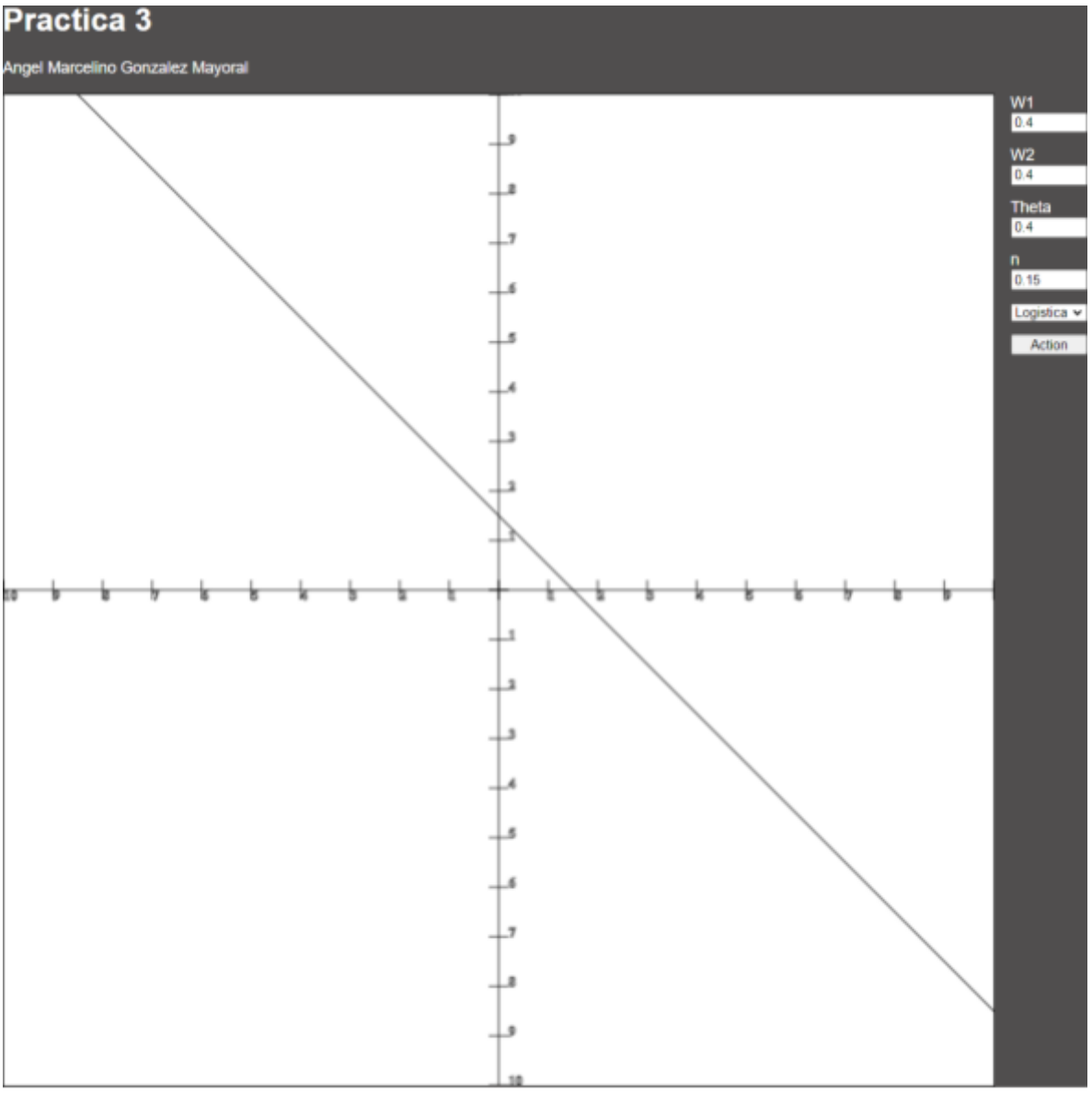
Inteligencia Artificial II 2022A

Práctica 3

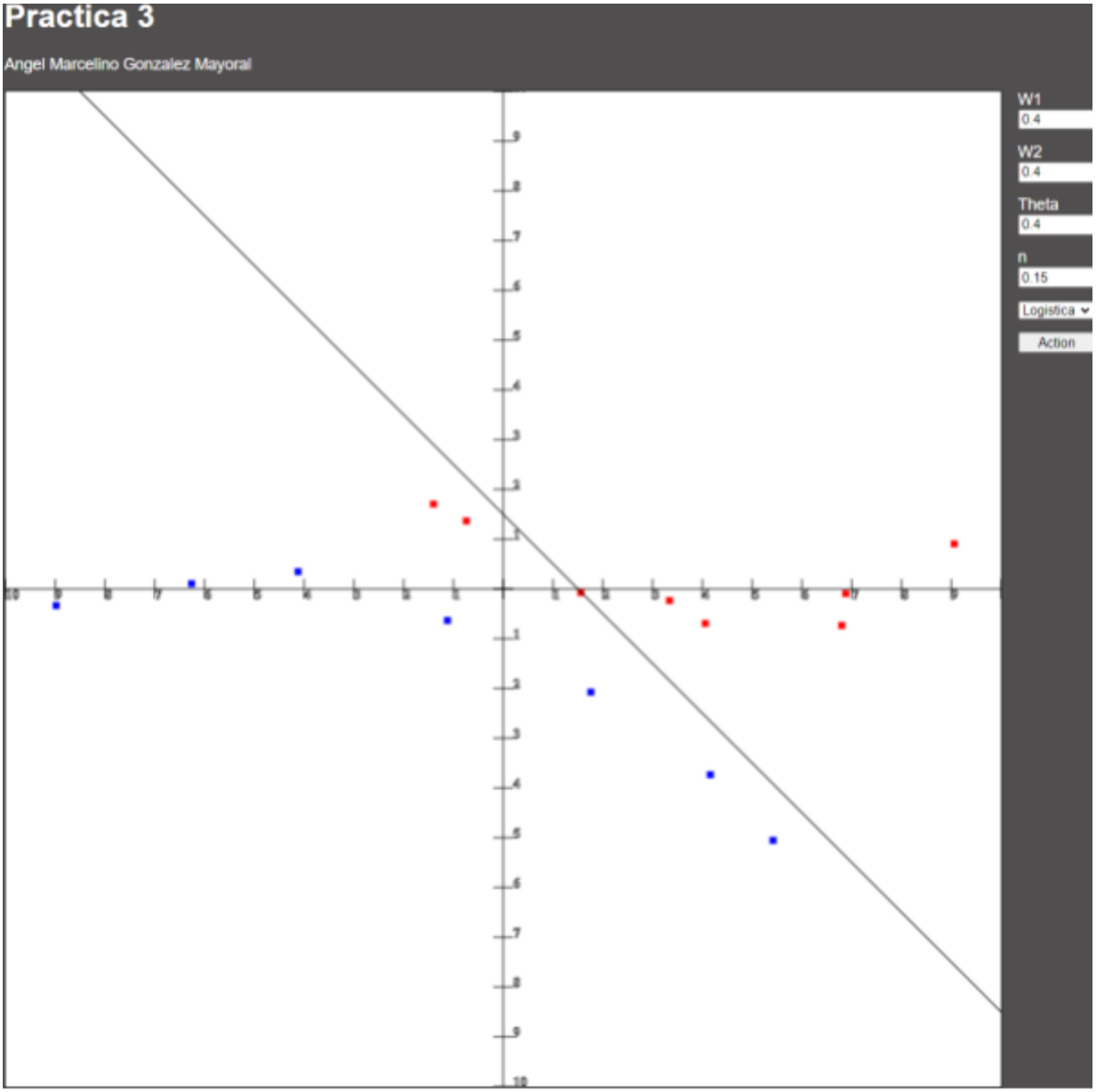
Introducción

En esta práctica realizaremos un cambio al perceptrón. En vez de utilizar una función de activación de escalón. Se utilizarán diferentes funciones de activación las cuales son la logística, la tangente hiperbólica y la linear. Esto con el propósito de mejorar la clasificación debido a que de esta forma la neurona particiona los datos de tal manera que el promedio de los errores cuadrados sea lo menor posible. Por lo tanto no se presentarán casos que se presentaban en el perceptrón que por ejemplo la línea de clasificación quedaba más cerca de cierta clase.

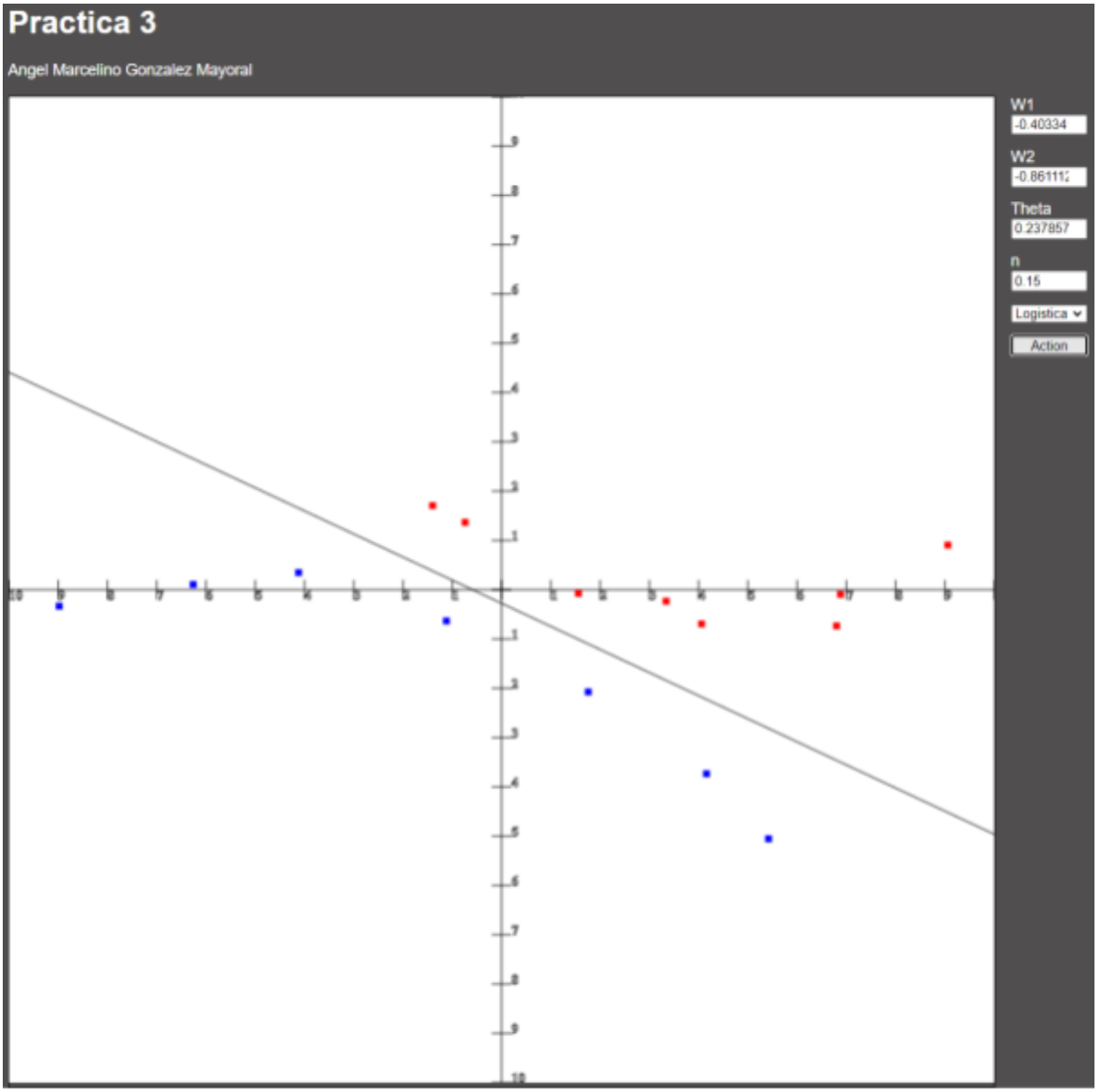
Capturas de pantalla



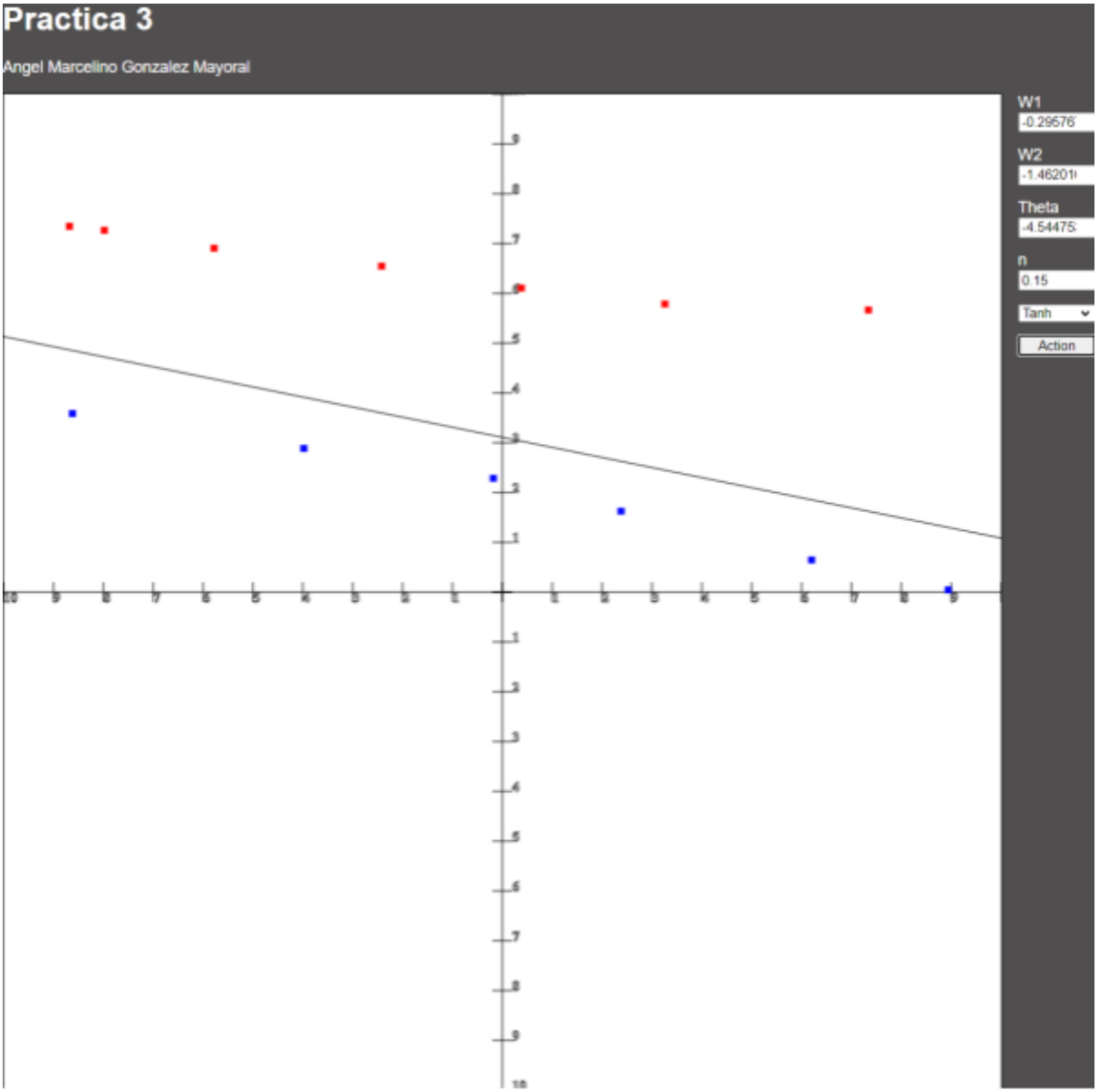
Estado inicial



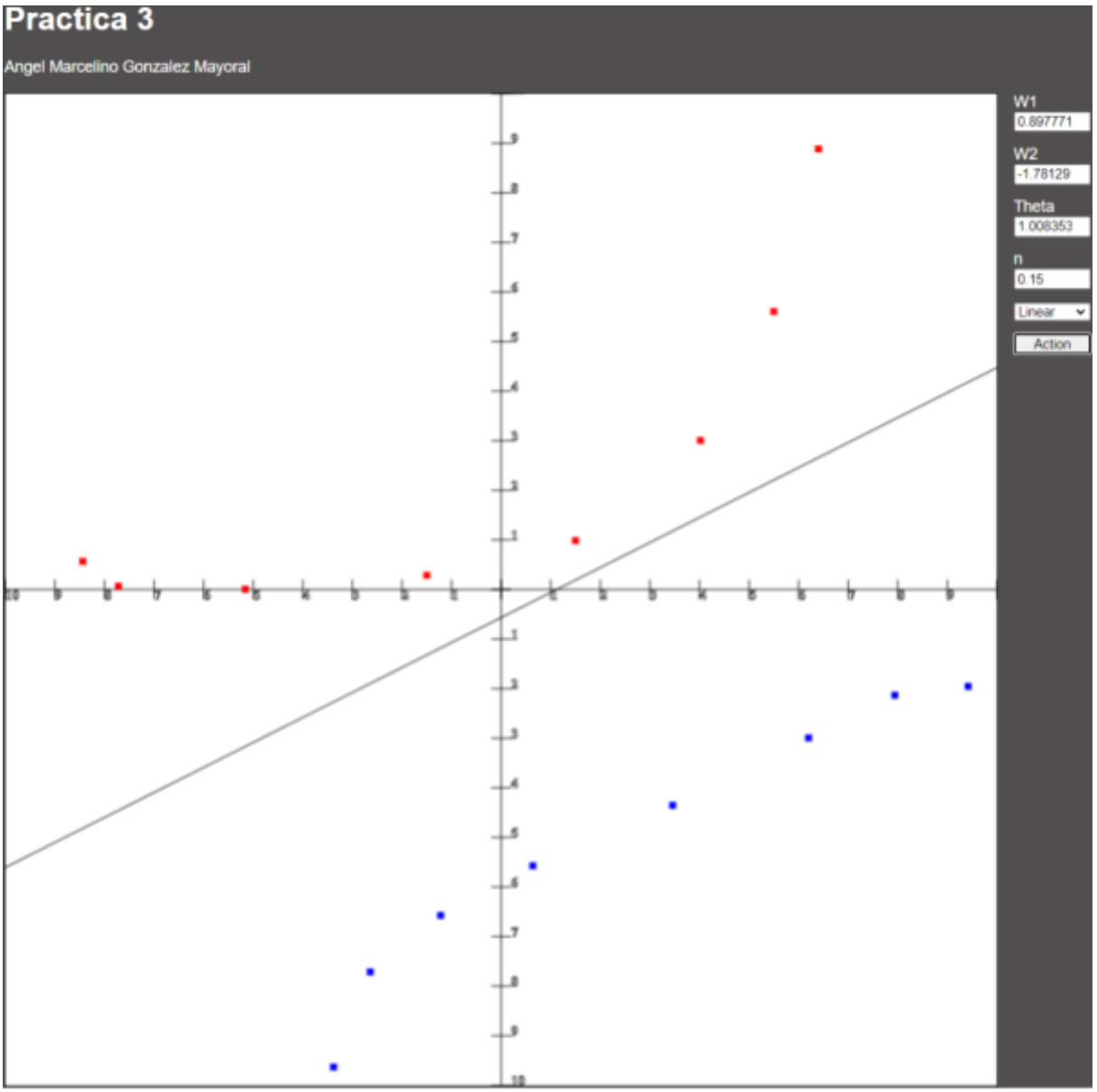
Se pueden agregar puntos de diferentes clases



Ejecución del entrenamiento con función de activación logística



Ejecución del entrenamiento con función de activación tanh



Ejecución del entrenamiento con función de activación linear

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 = 7;
    let pointHeight = 7;
    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));
  }
}

<!-- index.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>Practica 3</h1>
    <p>Angel Marcelino Gonzalez Mayoral</p>
    <div class="container">
      <div class="canvas-container">
        <canvas id="canvas" height="1000px" width="1000px"> </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" />
        <label for="n">n</label>
        <input id="n" type="number" value="0.15" />
        <select id="activationFn">
          <option value="1">Logistica</option>
```

```
          <option value="2">Tanh</option>
          <option value="3">Linear</option>
        </select>
        <button type="button" id="actionButton">Action</button>
      </div>
    </div>
    <script src="coordinates-translator.js"></script>
    <script src="neuron-calculator.js"></script>
    <script src="trainer.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>

//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
const LOGISTIC = 1;
const TANH = 2;
const LINEAR = 3;
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];
    }
  }

  calculateLogistic(point, weights, a) {
    let v = this.#calculateV(point, weights);
```

```
  let result = this.logisticFunction(a, v);
  if (result < 0.5) {
    return [0, v, result];
  } else {
    return [1, v, result];
  }
}

calculateTanh(point, weights, a) {
  let v = this.#calculateV(point, weights);
  let result = this.tanh(v);
  if (result < 0.5) {
    return [0, v, result];
  } else {
    return [1, v, result];
  }
}

calculateLinear(point, weights, a) {
  let v = this.#calculateV(point, weights);
  let result = this.linear(a, v);
  if (result < 0) {
    return [0, v, result];
  } else {
    return [1, v, result];
  }
}

linearFunction(a, v) {
  let result = a * v;
  return result;
}

logisticFunction(a, v) {
  let numerator = 1;
  let d_e = Math.exp(-a * v);
  let denominator = 1 + d_e;
  return numerator / denominator;
}

logisticFunctionDerivative(a, v) {
  let result =
```

```
      a * this.logisticFunction(a, v) * (1 - this.logisticFunction(a,
v));
    return result;
  }

  tanh(v) {
    let result = Math.tanh(v);
    return result;
  }
  tanhDerivative(v) {
    let result = 1 - this.tanh(v) * this.tanh(v);
    return result;
  }

  linear(a, v) {
    let result = a * v;
    return result;
  }

  linearDerivative(a) {
    return a;
  }

  /**
   *
   * @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;
  #result;
  /**
   *
   * @param {number} x
   * @param {number} y
   * @param {string} clas
   * @param {() => void} reRenderCallback
   */
  constructor(x, y, clas, reRenderCallback, result = null) {
    super(reRenderCallback);
    this.x = x;
    this.y = y;
    this.clas = clas;
    this.showClas = clas;
    this.result = result;
  }
  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;
  }
  set result(value) {
    this.#result = value;
  }

  get result() {
    return this.#result;
  }
}

//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,
 * HTMLSelectElement,
 * HTMLCanvasElement]} */
let [
  button,
  w1Input,
  w2Input,
  thresholdInput,
  nInput,
  activationFnSelect,
  canvas,
] = [
  "actionButton",
  "w1",
  "w2",
  "threshold",
  "n",
  "activationFn",
  "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,
      +activationFnSelect.value,
      activationFnSelect.value == 3 ? 0.15 : 0.05
    );
    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);
}
```

```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: 70px;
}

.input-container > select {
  margin-bottom: 1em;
}
```

```javascript
//trainer.js
const interval = 20;
let a = 0.1;
class Trainer {
  #n;
  #epoch;
  #trainerCallback;
  #endCallback;
  #neuronCalculator;
  #scale;
  #activationFunction;
  #errorPrecision;
  /**
   *
   * @param {number} n
   * @param {number} epoch
```

```
 */
constructor(
  n,
  epoch,
  trainerCallback,
  endCallback,
  scale,
  activationFunction,
  errorPrecision
) {
  this.#n = n;
  this.#epoch = 10000000;
  this.#trainerCallback = trainerCallback;
  this.#endCallback = endCallback;
  this.#neuronCalculator = new NeuronCalculator();
  this.#scale = scale;
  this.#activationFunction = activationFunction;
  this.#errorPrecision = errorPrecision;
  if (activationFunction == LINEAR) {
    a = 0.1;
  } else {
    a = 1;
  }
}

#getClassNumberMap() {
  let classNumberMap = {
    [CLASS_A]: 0,
    [CLASS_B]: 1,
  };
  if (this.#activationFunction == TANH) {
    classNumberMap = {
      [CLASS_A]: -1,
      [CLASS_B]: 1,
    };
  }
  if (this.#activationFunction == LINEAR) {
    classNumberMap = {
      [CLASS_A]: -1,
      [CLASS_B]: 1,
    };
  }
  return classNumberMap;
```

```
}

/**
 *
 * @param {Point[]} dataSet
 * @param {number[]} startWeights
 * @param {number} startBias
 */
async train(dataSet, startWeights) {
  let currentWeights = startWeights;
  let classNumberMap = this.#getClassNumberMap();

  let sumOffErrors = 0;
  let epoch = 0;
  do {
    sumOffErrors = 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, result] = this.#calculateNeuronOuput(
        current,
        currentWeights
      );
      let error = classNumberMap[current.showClas] - result;
      if (error !== 0) {
        let newWeights = this.#modifyWeight(
          [1, current.x, current.y],
          currentWeights,
          error,
          v
        );
        currentWeights = newWeights;
        this.#trainerCallback(newWeights);
        sumOffErrors += error * error;
        console.log("Error actual: ", error * error);
      }
    }, dataSet.length);
    epoch++;
    console.log("Error promedio: ", sumOffErrors / dataSet.length);
  } while (sumOffErrors / dataSet.length > this.#errorPrecision);
```

```
    this.#endCallback();
    console.log("Pesos resultantes" + currentWeights);
    console.log("Entradas resultantes" + currentWeights);
    for (let i = 0; i < dataSet.length; i++) {
      let current = dataSet[i];
      console.log(this.#calculateNeuronOuput(current, currentWeights));
      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);
      }
    });
  }

  #calculateNeuronOuput(current, currentWeights) {
    switch (this.#activationFunction) {
      case LOGISTIC:
        return this.#neuronCalculator.calculateLogistic(
          current,
          currentWeights,
          a
        );
      case TANH:
        return this.#neuronCalculator.calculateTanh(current,
currentWeights, a);
      case LINEAR:
        return this.#neuronCalculator.calculateLinear(
          current,
          currentWeights,
```

```
          a
        );
    }
  }

  #evaluateDerivative(v) {
    switch (this.#activationFunction) {
      case LOGISTIC:
        return this.#neuronCalculator.logisticFunctionDerivative(a, v);
      case TANH:
        return this.#neuronCalculator.tanhDerivative(v);
      case LINEAR:
        return this.#neuronCalculator.linearDerivative(a);
    }
  }

  /**
   *
   * @param {number[]} inputs
   * @param {number[]} currentWeights
   * @param {number} bias
   * @param {number} error
   */
  #modifyWeight(inputs, currentWeights, error, v = 1) {
    let newWeights = currentWeights.map((weight, index) =>
      this.#getNewValue.bind(this)(weight, inputs[index], error, v)
    );
    return newWeights;
  }

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

```
    }
}
```

Conclusiones

Adaline ayuda a realizar una clasificación más inteligente diría yo, ya que la línea clasificadora es aquella que tenga el menor promedio de errores esto hace que parta por el medio de los dos grupos de puntos. Esto no sucedía con el perceptrón. También noté que la línea corrige su posición más suavemente sin dar steps tan grandes como lo hace el perceptrón.