עבודת גמר בC#

**נושא הפרויקט:** פלאפי בירד למידת מכונה

**מגיש:** יונתן קרני

**ת"ז:** 214411571

**מנחי הפרויקט:** מריו סולאי

תוכן עניינים

[רקע קצר על המשחק פלאפי בירד: 3](#_Toc103460028)

[תיאור המשחק שאני עומד ליצור: 4](#_Toc103460029)

[תיאור הבעיה: 5](#_Toc103460030)

[תיאור פתרון לבעיה: 5](#_Toc103460031)

[סקירת אלגוריתמים בתחום הבעיה: 6](#_Toc103460032)

[הפתרון הפשוט: 6](#_Toc103460033)

[בעיות באלגוריתם: 8](#_Toc103460034)

[פתרון המערב למידת מכונה: 8](#_Toc103460035)

[מילון מושגים: 9](#_Toc103460036)

[אלגוריתם גנטי: 9](#_Toc103460037)

[רשת נוירונים: 9](#_Toc103460038)

[נוירון: 10](#_Toc103460039)

[שכבה: 10](#_Toc103460040)

[דור: 10](#_Toc103460041)

[אסטרטגיה: 11](#_Toc103460042)

[אלגוריתם ראשי: 11](#_Toc103460043)

[רשת הנוירונים: 11](#_Toc103460044)

[חישוב הפלט של רשת הנוירונים (כל המתודות שמוצגות בחלק זה שייכות למחלקה NeuralNetwork ): 14](#_Toc103460045)

[שחקן רשת הנוירונים: 16](#_Toc103460046)

[הקלט של רשת הנוירונים: 17](#_Toc103460047)

[מהלך האימון ואלגוריתם הגנטי: 20](#_Toc103460048)

[מבני נתונים: 23](#_Toc103460049)

[תרשים UML 24](#_Toc103460050)

[ארכיטקטורת Top Down Level : 24](#_Toc103460051)

[סביבת עבודה ושפת תכנות: 24](#_Toc103460052)

[תיאור ממשקים מחלקות ופונקציות ראשיות בפרויקט: 25](#_Toc103460053)

[ממשקים: 25](#_Toc103460054)

[מחלקות: 25](#_Toc103460055)

[רפלקציה: 34](#_Toc103460056)

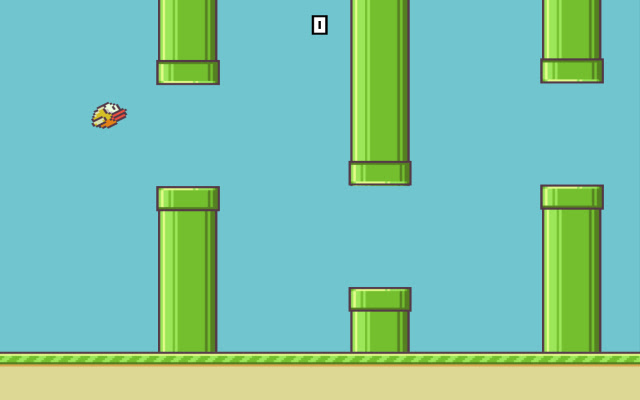
[ביבליוגרפיה: 34](#_Toc103460057)

[נספחים: 35](#_Toc103460058)

# רקע קצר על המשחק פלאפי בירד:

Flappy Bird הינו משחק פשוט לסמארטפון שפותח ב2013 וזכה להצלחה רבה כשנה מאוחר יותר.

המשחק דורש שליטה רק בכפתור אחד אתו ניתן לגרום לדמות הציפור במסך לקפוץ ולעבור את המכשולים. במעבר של כל מכשול השחקן צובר נקודה. מטרת המשחק היא כמובן לצבור כמה שיותר נקודות לפני שנפסלים.



# תיאור המשחק שאני עומד ליצור:

המשחק מיועד רק לשחקן אחד. אני רוצה לשדרג את המשחק וליצור אויב שבו המחשב ישלוט. שני השחקנים יתחרו ביניהם (האחרון ששורד מנצח) ויוכלו להשפיע אחד על השני במהלך המשחק.

השחקן המחשב



# תיאור הבעיה:

בflappy bird מטרת המשחק היא לעבור כמה שיותר מכשולים ולהשיג נקודות. בשביל אנשים המשימה יכולה להידמות פשוטה אך ככל שהמהירות עולה הרמה והופכת לכמעט בלתי אפשרית. ועבור מחשב המשימה אפילו עוד יותר מסובכת והאלגוריתם הדרוש לפתור לרוב מערב למידת מכונה.

\*ישנו פתרון חלקי עבור הבעיה המערב אלגוריתם יותר פשוט אך יש לו חסרונות רבים, על האלגוריתם הזה אני אסביר בהמשך.

# תיאור פתרון לבעיה:

אני רוצה לפתח קודם כל את המשחק ולאחר מכן קוד שיצליח לנצח את המשחק ללא שימוש בספריה שקשורה ללמידת מכונה, כלומר אני אממש את כל הקוד הקשור בלמידת מכונה בעצמי ([רשת נוירונים](#רשת_נוירונים) ואלגוריתם גנטי) ללא ספריות עזר למעט ספריית מנוע המשחקים Unity בו אני אפתח את המשחק.

# סקירת אלגוריתמים בתחום הבעיה:

## הפתרון הפשוט:

תחילה כשנתקלתי בבעיה זאת ניסיתי לחשוב אם קיים אלגוריתם פשוט וקצר שיפתור את הבעיה שלי. לבסוף יישמתי אלגוריתם זה:

-כל פריים נבדוק אם קיים צינור (מכשול) מלפנינו.

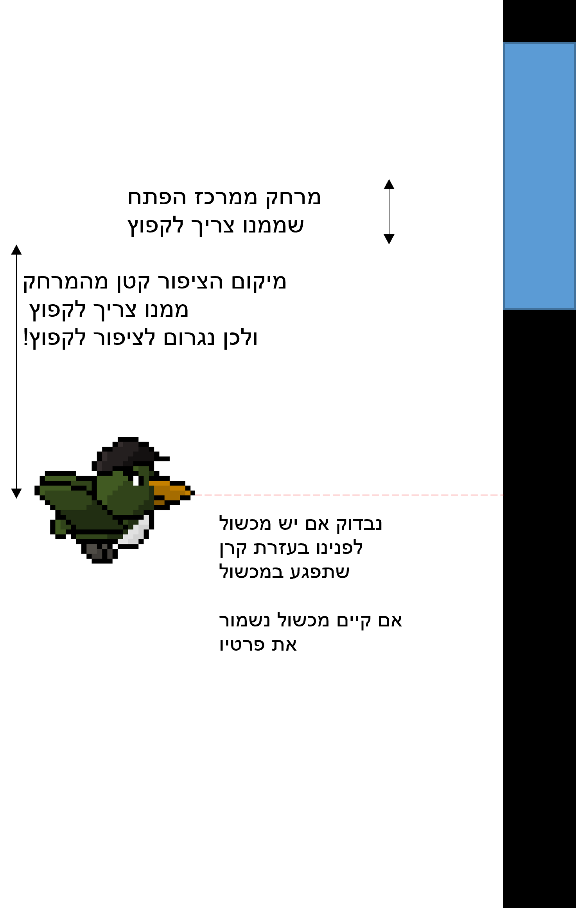
-אם קיים, נשמור את פרטי הפתח של המכשול (מיקום, גודל).

//זאת על מנת לגרום לציפור לקפוץ במקום במקרה שאין מכשולים.

-אם לא קיים, נבדוק את גובה הציפור (מיקום בציר האנכי).

-אם המיקום שלה נמוך ממרכז המסך נגרום לציפור לקפוץ.

-אם מצאנו מכשול מלפנינו.

-נבחר מספר מסוים שמבטא את המרחק (בציר האנכי) ממרכז פתח המכשול הבא נקרא למשתנה זה *offset*.

-נבדוק אם הציפור נמוכה יותר ממיקום הפתח הבא פחות *offset .*

*-*אם מיקומה אכן נמוך יותר נגרום לציפור לקפוץ.

להלן הקוד של הפונקציות המממשות המממשת אלגוריתם זה:

public void CheckForPipe()

    {

        //אורך הקרן שנשלח על מנת לבדוק אם קיים מכשול

        int rayLength = 14;

        //ניצור אובייקט של קרן בעזרת פונקציה שמקבלת את נקודת ההתחלה,

        //כיוון התקדמות, אורך קרן ושכבה של אובייקטים שבהם היא תתנגש

        RaycastHit2D hit1 = Physics2D.Raycast(transform.position, transform.right, rayLength, pipeLayer);

        //נבדוק אם פגענו במכשול

        if (hit1)

        {

            //נעדכן משתנה שיחזיק את האובייקט של המכשול

            if (hit1.transform.gameObject.tag == "pipeFlag")

                nextPipe = hit1.transform.parent;

            else

                nextPipe = hit1.transform;

        }

    }

public virtual bool IsJump()

    {

        //אם קיים מכשול מלפני הציפור

        if (nextPipe != null)

        {

            //אם המיקום של הציפור קטן מהמיקום של הפתח הבא פחות ההיסט

            if (transform.position.y < nextPipe.GetComponent<PipeSetup>().hole.position.y - offset)

                return true;

        }

        //במקרה שלא קיים מכשול נבדוק אם גובה הציפור נמוך ממרכז המסך, אם כן נקפוץ

        else if (transform.position.y < -3 && nextPipe == null)

            return true;

        return false;

    }

## בעיות באלגוריתם:

לאלגוריתם זה יש חיסרון בולט ומשמעותי. עבור פתח שגודלו קטן מגובה הקפיצה של הציפור האלגוריתם לא יצליח לגרום לציפור להתקדם, זאת מאחר שמקרה בו הציפור תקפוץ בתוך הצינור מאוד סביר ואם תעשה זאת בפתח קטן מדי היא תיתקע בתקרת החור.

בנוסף, תנועת הציפור בשימוש אלגוריתם זה תיראה "רובוטית" מדי. עבור מצב בו הציפור מתחת למקום הראוי היא תקפוץ כל פריים עד שתגיע ליעד וזאת תנועה שנראית כמעין רמאות (לחיצות מהירות מידי).

ניתן להוסיף מרווח זמן של המתנה בין כל קפיצה אך לאחר שינוי זה יכול לקרות מצב שהציפור לא תספיק להגיע לפתח.

לא הצלחתי לשפר את האלגוריתם בשביל להתגבר על חסרונות אלו.

\*החלטתי לוותר על שימוש באלגוריתם זה, הצגתי אלגוריתם זה רק על מנת להראות את הצורך באלגוריתם מורכב יותר.

## פתרון המערב למידת מכונה:

לאחר מחקר באינטרנט על הבעיה הבנתי שניתן לפתור את הבעיה בשימוש אלגוריתמים מעולם הבינה המלאכותית.

לפני שאסביר איך אלגוריתם זה עובד אני אתן רקע קצר על מושגים מעולם למידת המכונה בהם אני אשתמש בהמשך.

# מילון מושגים:

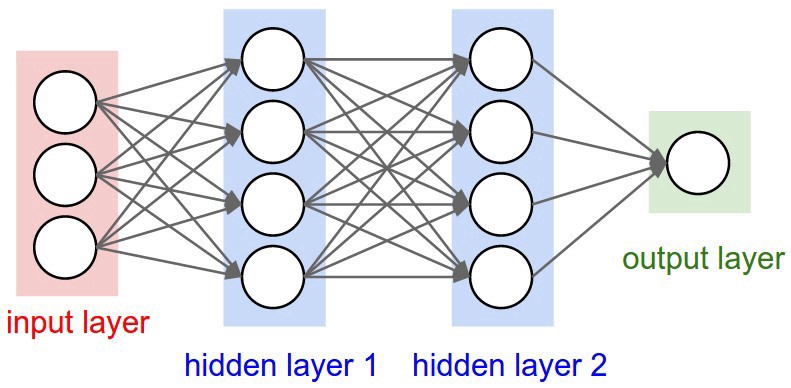
## אלגוריתם גנטי:

"השם אלגוריתמים גנטיים מתאר משפחה של אלגוריתמים לחיפוש, מידול ומיטוב (אופטימיזציה), שבהם משלבים זה בזה אלמנטים של פתרונות אפשריים לבעיה, ומפעילים הליכים של ברירה מלאכותית כדי לבחור את המועמדים שיעברו לשלבים הבאים. רעיון תכנותי בסיסי זה מושפע מההצלחה של האבולוציה בפתרון בעיות אמיתיות." (מתוך ויקיפדיה)

## רשת נוירונים:

"רשת עצבית מלאכותית (ANN – Artificial Neural Network), רשת נוירונים או רשת קשרית הוא מודל מתמטי חישובי, שפותח בהשראת תהליכים מוחיים או קוגניטיביים המתרחשים ברשת עצבית טבעית ומשמש במסגרת למידת מכונה. רשת מסוג זה מכילה בדרך כלל מספר רב של יחידות מידע (קלט ופלט) המקושרות זו לזו, קשרים שלעיתים קרובות עוברים דרך יחידות מידע "חבויות" (Hidden Layer). צורת הקישור בין היחידות, המכילה מידע על חוזק הקשר, מדמה את אופן חיבור הנוירונים במוח."

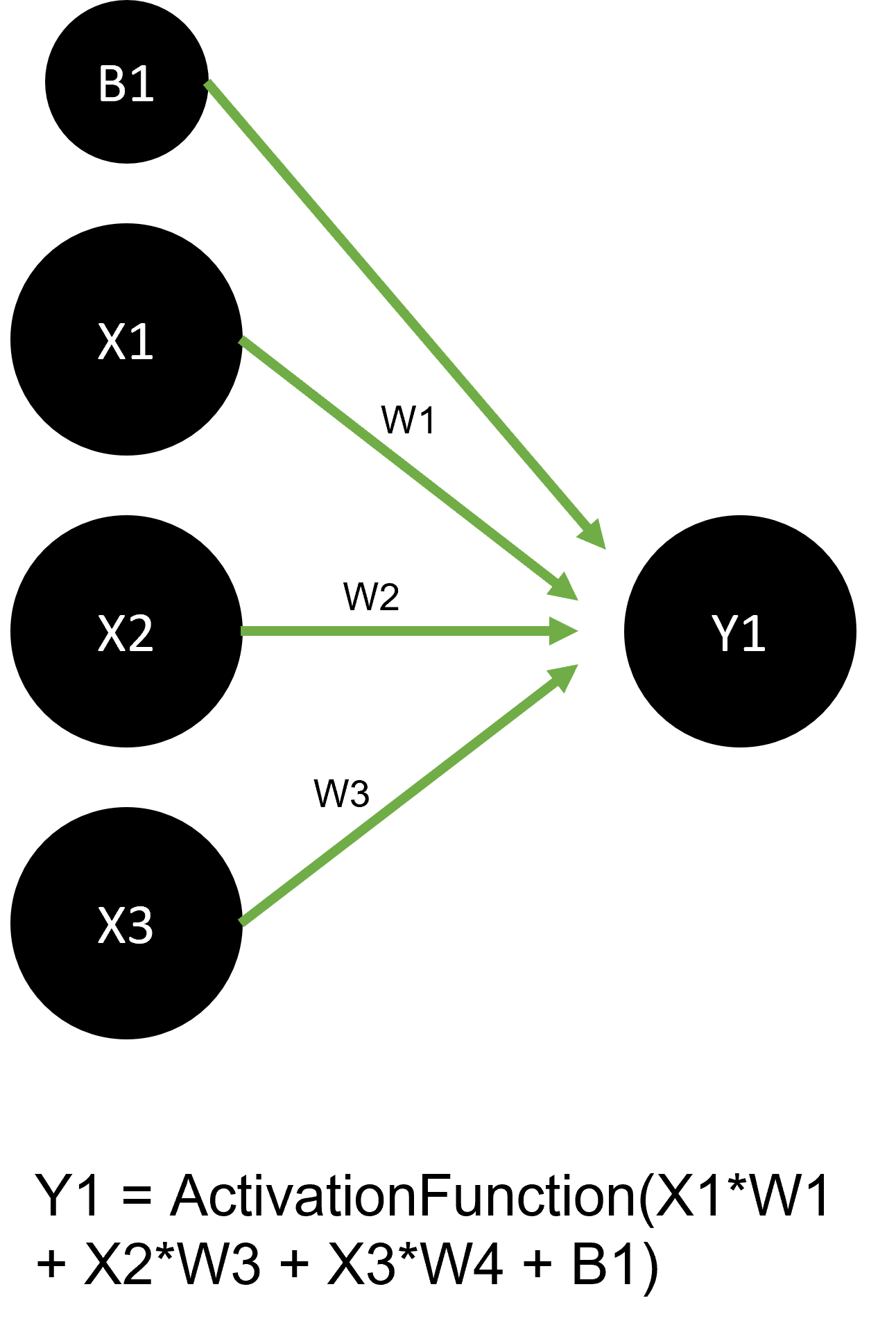
דוגמה לרשת נוירונים:



## נוירון:

בהקשר של [רשת נוירונים](#רשת_נוירונים) (שמכילה מספר נוירונים), מייצג את הקודקודים של הגרף. כל נוירון הוא אובייקט שמכיל מספר כלשהו (מייצג את רמת הגירוי שלו) ומספר קשרים לנוירונים שונים. לכל נוירון יש דרגת כניסה ששווה למספר הנוירונים בשכבה הקודמת ודרגת יציאה שווה למספר הנוירונים בשכבה הבאה.

ערך של כל נוירון מחושב בצורה הפשוטה הבאה (משוואה לינארית):

סכום של כל הערכים של הנוירונים בשכבה הקודמת כפול ערך הקשר שלהם לנוירון זה ועוד ערך ההיסט של השכבה הקודמת.

\*ברשתות נוירונים שונות נהוג לעשות מניפולציה מתמטית על מנת לשים את ערך הפלט בטווח מסוים. פעולה זאת נקראת activation function. בפרויקט זה השתמשתי בפונקציה המתמטית TanH .

## שכבה:

גם בהקשר של [רשת נוירונים](#רשת_נוירונים). שכבה ברשת מכילה מספר של נוירונים. לרוב [רשת נוירונים](#רשת_נוירונים) מכילה שכבת קלט, שכבות נסתרות ושכבת פלט.

## דור:

בהקשר של [אלגוריתם גנטי](#אלגוריתם_גנטי), דור הוא סט של מספר שחקנים, כל שחקן נושא "גנים שונים" כלומר תכונות שונות שתורמות/פוגעות בהישרדותו.

# אסטרטגיה:

נרצה לבנות [רשת נוירונים](#רשת_נוירונים) שבהתאם לקלט מסוים תדע להוציא פלט של אמת או שקר (לקפוץ או לא). אך על מנת לבנות [רשת נוירונים](#רשת_נוירונים) שתצליח במשימה נצטרך למצוא את המשקלים המתאימים.

יש כל מיני שיטות לשפר ביצועים של [רשת נוירונים](#רשת_נוירונים) (למצוא משקלים מתאימים), אני בחרתי בשיטה שמערבת [אלגוריתם גנטי.](#אלגוריתם_גנטי)

כל רשת ביצירתה תתחיל עם ערכי משקלים רנדומליים (בין 0.5 ל-0.5-).

כל ריצה נזמן מספר גדול של שחקנים שכל אחד יכיל רשת שונה.

עבור השחקנים שישרדו הכי פחות זמן נמחק את הרשת שלהם ונחליף אותה בשכפול של הרשת ששייכת לשחקן שהצליח לשרוד הכי הרבה זמן. אך שכפול זה אינו שכפול מדויק, כלומר זהו יהיה שכפול שעבר "מוטציה" - שינוי רנדומלי וקטן של המשקלים ברשת.

# אלגוריתם ראשי:

## רשת הנוירונים:

נבנה [רשת נוירונים](#רשת_נוירונים) בשיטת הפולימורפיזם. תחילה נבנה מחלקה של [נוירון](#נוירון) בודד, לאחר מכן מחלקה שתייצג [שכבה](#שכבה) של נוירונים ולבסוף את הרשת כולה שתכיל מספר שכבות.

חלק ממחלקת Neuron:

public class Neuron

{

    public float val;

    public int size;

    //מערך המכיל את המשקלים של הקשתות שיוצאות מנוירון זה

    public float[] weights;

    public Neuron(int len)

    {

        this.size = len;

        this.val = 0;

        this.weights = new float[len];

        //נגריל ערכים עבור המשקלים

        for (int i = 0; i < len; i++)

        {

            this.weights[i] = Random.Range(-0.5f, 0.5f);

        }

    }

חלק ממחלקת Layer:

public class Layer

{

    //האיבר החופשי - ההיסט

    public float bias;

    //מספר הנוירונים שמכילה

    public int size;

    //מערך המכיל את הנוירונים ברשת

    public Neuron[] neurons;

    //נצטרך את גודל שכבה הבאה על מנת לקבוע כמה משקלים יהיו לכל נוירון ברשת

    public Layer(int size, int nextLayerSize)

    {

        //ערך רנדומלי לאיבר החופשי

        this.bias = Random.Range(-1f, 1f);

        this.size = size;

        //ניצור כל נוירון ברשת

        this.neurons = new Neuron[size];

        for (int i = 0; i < size; i++)

        {

            this.neurons[i] = new Neuron(nextLayerSize);

        }

    }

חלק ממחלקת NeuralNetwork:

public class NeuralNetwork : IComparable<NeuralNetwork>

{

    //משתנה עזר עבור מספר סידורי

    private static int counter = 0;

    //מספר סידורי של הרשת

    public int serialNumber;

    //הכתובת של הרשת בתור קובץ טקסט במערכת קבצים (שדה עזר אופציונלי)

    public string path;

    //כמה זמן הרשת שרדה בשניות

    public float fitness;

    //כמה מכשולים הרשת עברה

    public int score;

    //מערך שמכיל את השכבות של הרשת

    public Layer[] layers;

    //שכבת הקלט - השכבה הראשונה

    private Layer inputLayer;

    //כמות השכבות ברשת

    public int layerNum;

    //layerSet - כל מספר במערך מייצג את מספר הנוירונים בשכבה במיקום האינדקס

    //למשל עבור

    //[2,5,2]

    //שכבת הקלט תכיל שני נוירונים, השכבה הנסתרת חמישה ושכבת הפלט תכיל שניים

    public NeuralNetwork(int[] layerSet)

    {

        //חישוב מספר סידורי ושמירתו

        this.serialNumber = counter;

        counter++;

        //נאפס משתנים המייצגים הישגים באימונים

        this.fitness = 0;

        this.score = 0;

        this.layerNum = layerSet.Length;

        //ניצור כל שכבה ונשמור במערך השכבות

        this.layers = new Layer[layerNum];

        for (int i = 0; i < layerNum -1; i++)

        {

            //למתודה הבונה של השכבה נכניס את מספר הנוירונים בשכבה עצמה ומספר הנוירונים בשכבה הבאה

            this.layers[i] = new Layer(layerSet[i], layerSet[i + 1]);

        }

        //על מנת לא לחרוג ממערך layerset

        //נבצע שורה זאת מחוץ ללולאה עם הערך 0 מאחר

        //ובמקרה בו אין שכבה באה גודל השכבה הבאה הוא כמובן 0

        this.layers[layerNum - 1] = new Layer(layerSet[layerNum - 1], 0);

        this.inputLayer = layers[0];

    }

## חישוב הפלט של רשת הנוירונים (כל המתודות שמוצגות בחלק זה שייכות למחלקה NeuralNetwork ):

תחילה אני אפרט על המתודה *FeedInput* שמקבלת מערך של מספרים ממשיים בתור קלט לרשת ושמה את הערכים בנוירונים של שכבת הקלט.

public void FeedInput(float[] inputArr)

    {

        //נעבור על כל נוירון בשכבת הקלט

        for (int i = 0; i < this.inputLayer.size; i++)

        {

            //כל נוירון מקבל את ערכו ממערך הקלט בהתאמה

            this.inputLayer.neurons[i].val = inputArr[i];

        }

    }

לאחר שהזנו קלט לרשת הנוירונים נרצה לחשב את ערכי הנוירונים (לפי הנוסחה שהצגתי ביחידת המושגים תחת הכותרת [נוירון](#נוירון)).

נעשה זאת עם המתודה *FeedForward* :

public void FeedForward()

    {

        //נעבור על כל שכבה

        for (int layer = 1; layer < this.layerNum; layer++)

        {

            Layer currentLayer = this.layers[layer];

            Layer lastLayer = this.layers[layer - 1];

            //נעבור על כל נוירון

            for (int neuron = 0; neuron < currentLayer.size; neuron++)

            {

                Neuron currentNeuron = currentLayer.neurons[neuron];

                //סכום כל הנוירונים הקודמים כפול המשקלים שלהם ועוד האיבר החופשי

                float value = currentLayer.bias;

                //נעבור על כל משקל בשכבה הקודמת

                for(int weight = 0; weight < lastLayer.size; weight++)

                {

                    //נוסיף לסכום את הערך של הנוירון הקודם הנוכחי כפול המשקל של הקשת שמחברת ביניהם

                    value += lastLayer.neurons[weight].val \* lastLayer.neurons[weight].weights[neuron];

                }

                //פונקציית ההפעלה - מכווצת את ערך הנוירון לטווח מסוים

                currentNeuron.val = (float)Math.Tanh(value);

            }

        }

    }

על מנת לקבל פלט סופי של אמת או שקר לגבי ההחלטה אם לקפוץ או לא נשתמש במתודה *GetOutput* :

public bool GetOutput()

    {

        //ערך הנוירון האחרון

        float result = this.layers[this.layerNum - 1].neurons[0].val;

        return result > 0.6;

    }

\*החלטתי על הערך 0.6 באופן שרירותי של ניסוי וטעיה.

## שחקן רשת הנוירונים:

מחלקה של שחקן שבו המחשב שולט. שחקן זה פועל לפי האלגוריתם של רשת הנוירונים. כל פריים במשחק הוא יכניס את הקלט שהוא אוסף לתוך הרשת שלו ויחליט לקפוץ לפי הפלט שלה. לפני שאסביר על מחלקה זאת אני אפרט על שרשרת הירושה שבאה לפניה.

FlappyInputController מחלקת תנועה בסיסית של הציפור ושליטה בה (מחלקה שבנויה עבור השחקן). כוללת פונקציה של קפיצה, פגיעה במכשול וכולי.

יורש

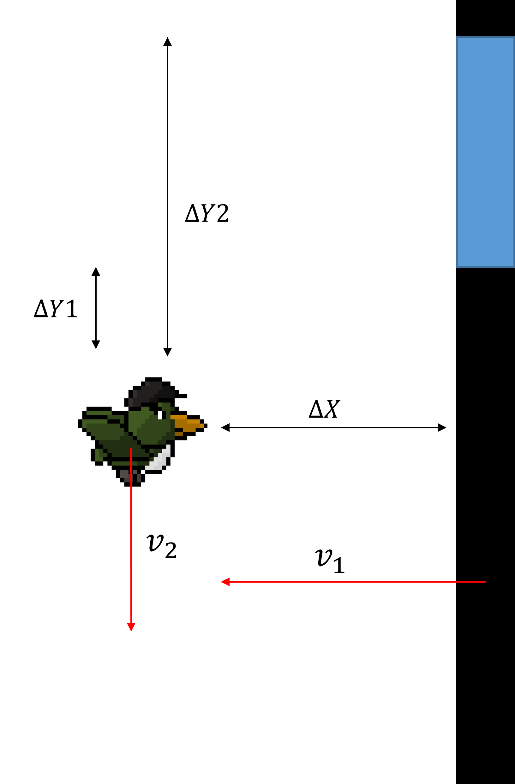
FlappyBot מחלקת תנועה של ציפור הנשלטת על ידי המחשב. ציפור זאת תשחק לפי ה[אלגוריתם הפשוט](#_הפתרון_הפשוט:) שהסברתי קודם לכן. מחלקה זאת מכילה פונקציה למציאת המכשול הבא.

יורש

FlappyNeural מחלקת תנועה עבור ציפור שנשלטת על ידי המחשב. ציפור זאת תשחק לפי הפלט שתביא רשת הנוירונים שלה. במותה היא תכניס את הזמן ששרדה ואת מספר הנקודות שלה לתוך השדות המתאימים ברשת שלה.

מחלקה זאת מכילה [רשת נוירונים](#רשת_נוירונים) וכל פריים היא תעביר לה כקלט פרמטרים שונים לגבי מיקומה במשחק.

## הקלט של רשת הנוירונים:

**לאחר למידה רבה בעזרת ניסוי וטעיה (וכמובן סקירה באינטרנט), בחרתי בפרמטרים הבאים עבור הקלט לרשת הנוירונים.

בציר האופקי

*מרחק מהפתח*

*מהירות המכשול שבא לקראת הציפור*

בציר האנכי

*מרחק מתקרת הפתח*

*מרחק מתחתית הפתח*

*מהירות הציפור בציר האנכי*

*\**הסיבה שנצטרך את המהירות האנכית של הציפור היא שפעולת הקפיצה מאחורי הקלעים רק תוסיף לאובייקט כוח מעלה. מתוך שיקולי תנע פיזיקליים ציפור שתנוע מהר מטה תקפוץ פחות גבוה מציפור שמהירותה אפסית.

\*מהירות המכשול בתור קלט יתרום לרשת הנוירונים את היכולת להתמודד עם כל מיני מהירויות של מכשולים (לפי האימון).

מתודה *IsJump* תחשב האם כדאי לציפור לקפוץ או לא בעזרת רשת הנוירונים. שייכת למחלקה FlappyNeural .

public override bool IsJump()

    {

        //אם אין מכשול נגרום לציפור לקפוץ במקום

        if (transform.position.y < -3 && nextPipe == null)

            return true;

        else if(nextPipe != null)

        {

            //מתוך האובייקט שמכיל את המכשול הבא ניקח את הפרטים הנדרשים

            Transform hole = nextPipe.GetComponent<PipeSetup>().hole.transform;

            Transform upEdge = nextPipe.GetComponent<PipeSetup>().upperEdge.transform;

            Transform lowEdge = nextPipe.GetComponent<PipeSetup>().lowerEdge.transform;

            pipeSpeed = nextPipe.GetComponent<PipeSetup>().speed;

            //חישוב כל הפרמטרים שניתן לרשת הנוירונים

            deltaX = hole.position.x - transform.position.x;

            velocityY = rb.velocity.y;

            disHigh = upEdge.position.y - transform.position.y;

            disLow = transform.position.y - lowEdge.position.y;

            float[] input = { deltaX, velocityY, disHigh, disLow, pipeSpeed};

            //נפעיל את המתודות שיחשבו את פלט הרשת

            net.FeedInput(input);

            net.FeedForward();

            //נחזיר את הפלט של הרשת

            return net.GetOutput();

        }

        //אחרת לא נקפוץ

        return false;

    }

הפונקציה *NeuralUpdate* נקראת כל פריים, היא תקרא ל*CheckForPipe* על מנת לעדכן את אובייקט המכשול. בנוסף, היא תקרא ל*IsJump* בשביל לקבוע אם לקפוץ או לא.

protected void NeuralUpdate()

    {

        //נעדכן את הזמן שהציפור שרדה

        aliveDuration += Time.deltaTime;

        //FlappyBot מתודה של

        CheckForPipe();

        if (IsJump())

        {

            //FlappyInputController מתודה של

            Jump();

        }

    }

## מהלך האימון ואלגוריתם הגנטי:

אם הייתה בידנו [רשת נוירונים](#רשת_נוירונים) מתאימה עם משקלים מתאימים אז האלגוריתם עד כה היה עובד. אך עדיין אין בידנו רשת כזאת, אנחנו נצטרך למצוא את המשקלים שלה.

על מנת למצוא את המשקלים של הרשת ננקוט בגישה של [אלגוריתם גנטי](#אלגוריתם_גנטי). זהו אלגוריתם שמושפע מהדרך בה עובדת האבולוציה – החזק שורד והוא יעביר את הגנים שלו ל[דור](#דור) הבא בעוד שהחלש ימות והגנים שלו ימחקו. ובאופן דומה מאוד לאבולוציה כך נמצא את הרשת עם הערכים המתאימים.

אלגוריתם מילולי:

-נגדיר מספר שמייצג את רמת השינוי כל [דור](#דור) *changeRate*

-נגדיר מספר שמייצג את כמות הציפורים בכל [דור](#דור) *population*

-נגדיר רשימה של רשתות נוירונים ריקה

-עבור ה[דור](#דור) הראשון

-תחזור *population* פעמים

-תזמן ציפור חדשה עם רשת שמכילה משקלים רנדומליים

-תשמור את הרשת של הציפור הנוכחית בתוך הרשימה

-בסוף ה[דור](#דור) תמיין את רשימת הציפורים לפי הזמן שהן שרדו

-עבור כל [דור](#דור)

נגדיר מספר *index = 0*

-עבור *index* קטן מ*changeRate*

-תגדיר רשת חדשה שתכיל העתק של הרשת ששייכת לאיבר במיקום *index* ברשימה

-תבצע מוטציה על הרשת החדשה (שינוי משקלים באופן מקרי)

-תוסיף את הרשת החדשה לרשימה

-תמחק את הרשת במיקום ה *index* מהרשימה

-עבור כל רשת ברשימת הרשתות

-תזמן ציפור חדשה עם רשת מתוך הרשימה

-בסוף ה[דור](#דור) תמיין את רשימת הציפורים לפי הזמן שהן שרדו.

לאחר שנמצא את הרשת הטובה ביותר נשמור את המשקלים שלה ואת הגודל שלה בתוך קובץ טקסט בפורמט מוגדר מראש. עכשיו נוכל לזמן ציפור שתצליח לשרוד לאורך זמן!

**דוגמה של אימון מוצלח** שהרצתי לילה שלם בו הצלחתי לגלות [רשת נוירונים](#רשת_נוירונים) שהגיעה לשיא של כ-10,000 נקודות!

בציר הy: מספר הנקודות הכי גבוה שציפור ב[דור](#דור) זה הצליחה להשיג.

בצירx: מספר ה[דור](#דור).

בציר הy: מספר [דור](#דור).

בצירx: הזמן בו נגמר הדור הנוכחי.

\*ניתן להבחין כי הגרף השני מאוד דומה לlog

\*בנוסף אני לציין שזמן האימון וההצלחה שלו מאוד תלויה בגודל הפתח של המכשולים, בדומה לבנאדם גם לאלגוריתם יהיה יותר קשה ללמוד איך לשחק עם פתח צר יותר

# מבני נתונים:

* רשימת רשתות הנוירונים nets בתוך המחלקה TrainingManager רשימה זו היא מרכז האלגוריתם בגנטי שעליו פירטתי בחלק של [האלגוריתם הראשי](#_מהלך_האימון_ואלגוריתם)
* מערך של שכבות נוירונים Layer בתוך מחלקת רשת הנוירונים NeuralNetwork
* מערך של נוירונים Neuron בתוך מחלקת שכבת הנוירונים Layer
* מערך של מספרים שלמים המייצגים ערכי משקלים של [נוירון](#נוירון) במחלקת Neuron
* מערך netSet מכיל מ ספרים שלמים שמייצגים את כמות הנוירונים בכל [שכבה](#שכבה) ברשת, נמצא במחלקה BirdSpwner

# תרשים UML

# ארכיטקטורת Top Down Level :

פירטתי על הארכיטקטורה בחלק של [האלגוריתם הראשי](#_אלגוריתם_ראשי:).

# סביבת עבודה ושפת תכנות:

מנוע משחקים Unity 2020.30f1

שפת תכנות C#

סביבת פיתוח visual studio community 2017

# תיאור ממשקים מחלקות ופונקציות ראשיות בפרויקט:

## ממשקים:

Icompareable: ממשק שניתן לאובייקטים שניתן להשוות ביניהם. ממשק זה מחייב מימוש של הפונקציה CompareTo .

מימשתי את ממשק זה במחלקת רשת הנוירונים על מנת להיות מסוגל למיין רשימה שמכילה מספר רשתות לפי כמה זמן הרשת שרדה.

\*המתודות בשם Update נקראות על ידי אובייקט מנוע המשחקים כל פריים

\*MonoBehavior זוהי מחלקת הבסיס של כל אובייקט בסצנת המשחק של Unity.

## מחלקות:

|  |  |  |
| --- | --- | --- |
| **מחלקה** | **פונקציה/משתנה** | **הסבר** |
| FlappyInputController : MonoBehavior | public int score; | כמות המכשולים שהציפור עברה |
| private int hitpoints; | מספר נקודות החיים של הציפור. כשיגיעו ל0 הציפור תיפסל. |
| protected Rigidbody2D rb; | אובייקט המכיל את החלקים הפיזיקליים של הציפור (של מנוע המשחקים) |
| public float jumpSpeed; | מהירות הקפיצה |
| public bool isPlayer; | האם הציפור נשלטת בשחקן? עבור מחלקות יורשות משתנה זה יכיל שקר |
| void Update() | מוודאת לאפס את מהירות הציפור בציר האופקי. תקרא לפונקציית הקפיצה אם יתקבל קלט של הכפתור רווח. |
| public void Jump() | תגרום לציפור לקפוץ במהירות הנתונה ותעדכן את האנימציה. |
| public virtual void OnTriggerEnter2D(Collider2D collision) | מתודה זאת תיקרא כשהציפור תתנגש באובייקט מסוים. האובייקט בו היא התנגשה יכנס כפרמטר. במתודה זאת נבדוק במה התנגשנו, אם הציפור התנגשה במכשול או בקצוות המסך נקרא לפונקציית הפגיעה. |
| public void Hit() | תעדכן את מספר החיים של הציפור.  תבדוק כמה נקודות חיים נשאר לה ובהתאם תחליט אם לפסול את השחקן. |
| FlappyBot : FlappyInputController | public Transform nextPipe; | אובייקט המכיל את המכשול הבא. |
| public float offset; | ההיסט ממרכז הפתח שממנו הציפור תקפוץ. |
| public virtual bool IsJump() | המתודה שמכילה את האלגוריתם הפשוט אם לקפוץ או לא. |
| public void CheckForPipe() | תעדכן את משתנה המכשול הבא. היא תמצא את המכשול הבא בעזרת זימון קרן באורך מסוים שתתנגש במכשול. |
| private void Update() | כל פריים המתודה תקרא ל CheckForPipe בשביל לעדכן את המכשול הבא. היא גם תקרא ל IsJump אם היא תחזיר אמת היא תקרא לJump |
| FlappyNeural : FlappyBot | public bool isEnemyPlayer; | יש להבדיל בין ציפור שזומנה לתוך [דור](#דור) מסוים בזמן אימון לציפור האויב הסופי שבו נילחם. למשל לציפור אימונים יש נקודת חיים אחת לעומת האויב הסופי שלא יהיו שלוש נקודות חיים בדומה לשחקן. |
| public NeuralNetwork net; | רשת הנוירונים של הציפור. |
| public TextAsset netTextSource; | במקרה שהציפור היא האויב הסופי יש לטעון את הרשת שלו מתוך קובץ טקסט שמור. |
| float deltaX, velocityY, disHigh, disLow, pipeSpeed; | המשתנים שמכילים את הקלט שנכנס לרשת הנוירונים. פירטתי על כך בחלק [האלגוריתם הראשי](#_שחקן_רשת_הנוירונים:). |
| float aliveDuration; | משתנה שמכיל את כמות הזמן בשניות שהציפור שרדה עד כה. |
| public void InitNet(int[] netSet)  public void InitNet(NeuralNetwork other)  public void InitNet() | שלוש דרכים לאתחול רשת הנוירונים של הציפור.   1. יצירת [רשת נוירונים](#רשת_נוירונים) חדשה עם ערכים רנדומליים בעזרתי מערך המייצג את גודלה של הרשת. 2. העתקה של [רשת נוירונים](#רשת_נוירונים) קיימת. 3. טעינה של [רשת נוירונים](#רשת_נוירונים) מתוך קובץ הטקסט. |
| public override bool IsJump() | מתודה שדורסת את מתודת הבסיס במחלקה ממנה היא יורשת (דורסת את האלגוריתם הפשוט). לפי הפלט של רשת הנוירונים הפונקציה תחזיר אמת או שקר לגבי אם כדאי לציפור לקפוץ או לא. |
| void Update() | תפעל באותו אופן כמו במתודת האב רק שהפעם היא תקרא לIsJump ששייך לה. בנוסף היא תעדכן את הזמן שהציפור שרדה. |
| public override void OnTriggerEnter2D(Collider2D collision) | אם זהו שחקן האויב הסופי נקראה לאותה הפונקציה במחלקת האב. ואם זאת ציפור אימונים נבצע מספר צעדים בשביל לעדכן את מצב ה[דור](#דור) ושדות מסוימים ברשת הנוירונים (זמן הישרדות למשל). |
| Neuron | public float val; | ערך ה[נוירון](#נוירון). |
| public int size; | כמות הקשתות שיוצאות ממנו. |
| public float[] weights; | מערך המכיל את המשקל של כל קשת. |
| public Neuron(int len)  public Neuron(Neuron other) | שתי דרכים ליצור [נוירון](#נוירון) חדש:  1. ניתן את כמות הקשתות שיצאו ממנו והמשקלים יקבלו ערכים רנדומליים  2. נבצע העתקה לעומק של [נוירון](#נוירון) אחר. |
| Layer | public float bias; | האיבר החופשי "היסט". |
| public int size; | כמות הנוירונים שהרשת מכילה. גודל מערך הנוירונים. |
| public Neuron[] neurons; | מערך המכיל את הנוירונים שהרשת מכילה. |
| public Layer(int size, int nextLayerSize)  public Layer(float bias, int size, int nextLayerSize)  public Layer(Layer other) | שלוש שרכים ליצור [שכבה](#שכבה) חדשה:   1. ניתן את כמות הנוירונים שיהיו בה ואת כמות הנוירונים ב[שכבה](#שכבה) הבאה. ניצור כל [נוירון](#נוירון) בעזרת נתונים אלה.   האיבר החופשי יקבל ערך רנדומלי.   1. ניתן בנוסף גם את ערך האיבר החופשי. 2. נבצע העתקה לעומק מתוך [שכבה](#שכבה) שונה. |
| NeuralNetwork :  Icompareable  <NeuralNetwork> | public int serialNumber; | מספר סידורי |
| public string path; | הכתובת של קובץ הטקסט המכיל את הרשת (שדה אופציונלי) |
| public float fitness; | כמה זמן הרשת שרדה בשניות ב[דור](#דור) האחרון. |
| public int score; | כמה מכשולים הרשת עברה ב[דור](#דור) האחרון. |
| public Layer[] layers; | מערך המכיל את השכבות של הרשת. |
| public int layerNum; | כמות השכבות ברשת. |
| public NeuralNetwork(int[] layerSet)  public NeuralNetwork(NeuralNetwork other) | שתי דרכים ליצור [רשת נוירונים](#רשת_נוירונים):   1. ניתן מערך שמכיל מספרים שלמים המייצגים את גדלי השכבות. כל משקל וכל איבר חופשי יקבלו ערכים רנדומליים. 2. נבצע העתקה לעומק מתוך רשת שונה. |
| public void FeedInput(float[] inputArr) | נכניס ערכים לנוירונים בשכבת הקלט מתוך המערך שנקבל כפרמטר. |
| public void FeedForward() | נחשב בעזרת הנוסחה שהסברתי במושג [נוירון](#נוירון) את ערכי כל הנוירונים ברשת. |
| public void Mutate() | נבצע מוטציה על הרשת: נשנה באופן מקרי חלק מהמשקלים או חלק מהאיברים החופשיים. |
| public bool GetOutput() | המתודה תחזיר אמת אם ערך ה[נוירון](#נוירון) בשכבת הפלט מכיל ערך רצוי. |
| public int CompareTo(NeuralNetwork other) | מתודה שמממשת את ממשק ההשוואה. המתודה תחזיר את ההפרש בין זמן ההישרדות של רשת זאת לזמן ההישרדות של הרשת שהתקבלה. |
| PipeSetup : MonoBehaviour | public float speed; | מהירות המכשול בציר האופקי. |
| public float lowR, highR; | הטווח שבו נגריל את מיקום הפתח בצינור. |
| public Collider2D col1, col2; | שני אובייקטים ההתנגשות אחד בשביל החלק העליון של הצינור והשני בשביל החלק התחתון ובאמצע יש פתח. |
| public Transform hole, trigger, upperEdge, lowerEdge; | משתנים פומביים שמצביעים על מיקומים של חלקים מהמכשול. פרטים אלו יילקחו על ידי השחקן. |
| private void Setup() | נזיז את אובייקט החור למקום רנדומלי בטווח הנתון ואת מלבני ההתנגשות לשוליים של המכשול מסביב לפתח |
| private void Update() | כל פריים נזיז את המכשול בשביל ליצור את תנועתו ונבדוק אם הוא יצא מגבולות המסך, אם כן נשמיד את המכשול. |
| PipeSpawner : MonoBehaviour | public float speed; | המהירות שיקבלו המכשולים שיזומנו על ידי מחלקה זו. |
| public float increaseSpeedRate, increaseSpeedDelay; | יש אופציה להעלות את המהירות של המכשולים עם הזמן. יש פה שני משתנים הראשון מגדיר בכמה נעלה את מהירות המכשולים והשני כל כמה זמן נעשה זאת. |
| public GameObject pipe; | תבנית של אובייקט של מכשול שיש לזמן. |
| public float delay; | כל כמה זמן נזמן מכשול חדש. |
| public void IncreaseSpeed(float amount) | הפונקציה תעלה את המהירות שיקבלו המכשולים כשיזומנו. יש לציין כי זאת פונקציה חכמה שיודעת לשנות את מרווח הזמן בין הזימונים כך שהמרחק בין המכשולים יישאר קבוע. |
| public void SpawnPipe() | פונקציה לזימון מכשול חדש. |
| private void Update() | הפונקציה תזמן את spawnPipe() כל פרק זמן מוגדר שניתן כקלט. |
| BirdSpawner :  MonoBehaviour | public GameObject bird; | תבנית של אובייקט של ציפור שיש לזמן. |
| [Range(0,500)]  public int population; | כמות הציפורים שיש לזמן כל [דור](#דור) הטווח הוא מ0 ל500 |
| public int[] netSet; | מערך גדלי השכבות, רצוי בשביל יצירת רשתות חדשות עבור כל ציפור. |
| public void Spawn()  public void Spawn(List<NeuralNetwork> nets) | שתי דרכים לזמן [דור](#דור) חדש של ציפורים:   1. ה[דור](#דור) הראשון יכיל רשתות חדשות עם משקלים רנדומליים ולכן לא נצטרך שום פרמטר. 2. ב[דור](#דור)ות הבאים נזמן את הציפורים לפי הרשתות שעברו את התהליך של המיון הגנטי. |
| TrainingManager :  MonoBehaviour | public static int genNum; | מספר ה[דור](#דור) הנוכחי. |
| public int changeRate; | רמת השינוי של האלגוריתם הגנטי (פירטתי על כך בחלק [האלגוריתם הראשי](#_מהלך_האימון_ואלגוריתם)) |
| public int targetScore; | מספר הנקודות שאם ציפור כלשהי תגיע אליו נסיים את האימון בהצלחה. |
| public int highScore; | מספר הנקודות הכי גבוה שאליו הצליחה להגיע ציפור מ[דור](#דור) כלשהו באימון זה . |
| public static List<NeuralNetwork> nets; | רשימת רשתות הנוירונים של אימון זה. |
| public NeuralNetwork bestNet; | הרשת הכי טובה לרגע זה. |
| public void StartGeneration() | מתודה שתזמן את ה[דור](#דור) הראשון/הבא של הציפורים ותשמור את הרשתות שלהם. |
| private void GenerateNewNets() | יממש את האלגוריתם הגנטי שעליו הסברתי בחלק [האלגוריתם הראשי](#_מהלך_האימון_ואלגוריתם). ימחק את הרשתות הגרועות ויוסיף רשתות חדשות במקומן. |
| public void GenEnd() | המתודה תזומן בכל סוף של דור. היא תמיין את רשימת הרשתות ותקרא לפונקציה generateNewNets() על מנת לחדש את הרשימה לפי תוצאות הדור. הפונקציה גם תעדכן את השיא ואת הרשת הטובה ביותר. בנוסף היא תשמור רשתות ששברו שיא בתוך קובץ טקסט וגם תתעד כל סוף דור בתוך קובץ אקסל. |
| Collector :  MonoBehaviour.  מחלקה לטיפול בלקיחת פרסים במהלך המשחק נגד היריב.  מחלקה זאת לא מתייחסת לאימון הציפורים | public PipeSpawner myPipSpwn, enemyPipSpwn; | מחלקות המזמנות מכשולים של השחקן הנוכחי ושל היריב שלו. |
| public GameObject laser, shootPoint; | תבנית של אובייקט של קליע שניתן לפלוט מהציפור על מנת לפגוע ביריב.  וגם אובייקט בסצנה שמייצג את המיקום ממנו צריך לפלוט את הקליע. |
| public FlappyInputController flappyControl; | מחלקת השחקן/יריב. מאחר ומחלקת הציפור הרשתית יורשת ממחלקת השחקן ניתן לשים גם אותה בשדה זה. |
| private void OnTriggerEnter2D(Collider2D collision) | כשהשחקן/יריב מתנגשים בפרס כלשהו הפונקציה הזאת נקראת עם אובייקט הפרס בתור פרמטר.  פונקציה זאת תסווג איזה פרס קיבל השחקן ויקרא לפונקציות המתאימות כדי לממש את הפרס. בנוסף הפונקציה תשמיד את הפרס. |
| public void ShootLaser() | פונקציה לפליטת קליע. היא תיקרא על ידי פונקציית ההתנגשות אם היא תסווג את הפרס המתאים. |
| IEnumerator FastSpeed() | מתודה שיכולה להחזיר זמן המתנה ולכן קידומת הממשק.  המתודה תשנה את מהירות מכשולי היריב לזמן קצר.  גם היא תיקרא על ידי פונקציית ההתנגשות אם יסווג הפרס המתאים. |
| IEnumerator SlowSpeed() | אותו הסבר כמו מקודם רק שהפעם המתודה תאט את מהירות המכשולים של שחקן זה. |
| public void DestroyAllSpeedCollectibles() | ברגע שהשחקן/יריב לקח פרס שמשפיע על מהירות המכשולים נהרוס כל פרס אחר שקשור למהירות בסצנה על מנת לא ליצור סיבוך מיותר. |
| CollectorSpawner :  MonoBehaviour  מחלקה שנמצאת על המכשול. כלומר הפרסים ינועו ביחד עם המכשולים. | public Collider2D spawnRange; | מלבן התנגשות שמרחביו יגדירו את טווח הנקודות בהן ניתן לזמן פרסים מלפני המכשול. |
| public GameObject[] collectibles; | מערך של תבניות של פרסים שניתן לזמן. |
| public int spawnChance; | הסיכויים שנזמן פרס. למשל עבור הערך 3 נזמן פרס חדש בסיכוי של אחד לשלוש. |
| private void OnEnable() | הפונקציה תיקרא ברגע יצירת האובייקט והפעלתו בסצנה. בפונקציה זאת נזמן או לא (תלוי בהגרלה) פרס רנדומלי מתוך המערך. |

# רפלקציה:

את הפרויקט הזה התחלתי לפני כחצי שנה ועבדתי עליו הרבה זמן אבל אני מאוד מרוצה מהתוצאה. התחלתי אותו מתוך רצון אישי והתעניינות בלמידת מכונה והמשכתי אותו במסגרת היג'.

היו לי המון קשיים במהלך הפרויקט, למשל אחד מהם הוא שבמשך זמן רב לא הבנתי למה לא הצלחתי לגרום לציפורים להגיע רחוק יותר מ20 מכשולים. ואז נפל לי האסימון שהפתח פשוט קטן מדי!

בנוסף אני מאוד מרוצה שבחרתי לוותר על שימוש בספרייה של למידת מכונה כי רכשתי המון ידע על איך [רשת נוירונים](#רשת_נוירונים) עובדת מאוחרי הקלעים, מה המטרה בשימוש בה ואיך לשפר אותה.

אני גם מרוצה שבחרתי בסביבת עבודה קלה יחסית לפיתוח של אנימציה ותפעול בסיסי (מנוע משחקים). מאחר וזה נתן לי זמן רב לעבוד על האלגוריתמים המעניינים באמת בפרויקט.

# ביבליוגרפיה:

סדרה ביוטיוב על דרך הפעולה של רשתות נוירונים:

<https://www.youtube.com/watch?v=aircAruvnKk&t=4s&ab_channel=3Blue1Brown>

פרויקט גמור של אימון ציפורים בעזרת [אלגוריתם גנטי](#אלגוריתם_גנטי):

<https://xviniette.github.io/FlappyLearning/>

מאמר על איך ללמד מכונה לשחק פלאפי בירד בפייתון:

<https://towardsdatascience.com/reinforcement-learning-in-python-with-flappy-bird-37eb01a4e786>

# נספחים:

סרטון של אימון מוצלח שהקלטתי:

<https://www.youtube.com/watch?v=wDGTRChLry8&ab_channel=YonatanKarni>

קוד הפרויקט:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

[System.Serializable]

public class Neuron

{

    public float val;

    public int size;

    public float[] weights;

    /// <summary>

    /// neuron holds a value and several weights (edges to next layer neurons)

    /// </summary>

    /// <param name="len"> the number of weights </param>

    public Neuron(int len)

    {

        this.size = len;

        this.val = 0;

        this.weights = new float[len];

        for (int i = 0; i < len; i++)

        {

            this.weights[i] = Random.Range(-0.5f, 0.5f);

        }

    }

    /// <summary>

    /// deep copy

    /// </summary>

    /// <param name="other"></param>

    public Neuron(Neuron other)

    {

        this.size = other.size;

        this.val = 0;

        this.weights = new float[this.size];

        for (int i = 0; i < size; i++)

        {

            this.weights[i] = other.weights[i];

        }

    }

    public override string ToString()

    {

        string output = "[\*" + this.val + "\* ";

        for (int i = 0; i < this.weights.Length; i++)

        {

            output += ", " + this.weights[i].ToString("0.00");

        }

        output += "]";

        return output;

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// represent a single layer in the neural network

/// contains an array of neurons

/// </summary>

[System.Serializable]

public class Layer

{

    //the bias is the offset added to the formula

    //y = w1\*x1 ... + b (bias)

    public float bias;

    //how many neurons in the array

    public int size;

    public Neuron[] neurons;

    /// <summary>

    /// constructor to a new layer

    /// </summary>

    /// <param name="bias"> b variable </param>

    /// <param name="size"> how many neurons in the layer </param>

    /// <param name="nextLayerSize"> the next layer size is necessary for determining

    /// how many weights for each neuron in layer</param>

    public Layer(float bias, int size, int nextLayerSize)

    {

        this.bias = bias;

        this.size = size;

        this.neurons = new Neuron[size];

        for (int i = 0; i < size; i++)

        {

            this.neurons[i] = new Neuron(nextLayerSize);

        }

    }

    /// <summary>

    /// constructor to a new layer the bias is set randomly

    /// </summary>

    /// <param name="size"> how many neurons in the layer </param>

    /// <param name="nextLayerSize"> the next layer size is necessary for determining

    /// how many weights for each neuron in layer</param>

    public Layer(int size, int nextLayerSize)

    {

        this.bias = Random.Range(-1f, 1f);

        this.size = size;

        this.neurons = new Neuron[size];

        for (int i = 0; i < size; i++)

        {

            this.neurons[i] = new Neuron(nextLayerSize);

        }

    }

    /// <summary>

    /// deep copy of new layer

    /// </summary>

    /// <param name="other"> other layer </param>

    public Layer(Layer other)

    {

        this.bias = other.bias;

        this.size = other.size;

        this.neurons = new Neuron[size];

        for (int i = 0; i < size; i++)

        {

            this.neurons[i] = new Neuron(other.neurons[i]);

        }

    }

    public override string ToString()

    {

        string output = "{ ";

        for (int i = 0; i < size; i++)

        {

            output += neurons[i] + ", ";

        }

        output += "[" + this.bias + "]" + " }";

        return output;

    }

}

using System;

using System.Collections.Generic;

using UnityEngine;

[System.Serializable]

public class NeuralNetwork : IComparable<NeuralNetwork>

{

    //serial number helper

    private static int counter = 0;

    public int serialNumber;

    //if not null is set to the net text serialized file

    public string path;

    //the total survivel time

    public float fitness;

    //how many pipes passed

    public int score;

    public Layer[] layers;

    private Layer inputLayer;

    //how many layers the net containes

    public int layerNum;

    /// <summary>

    /// build new neural network with random weight values

    /// </summary>

    /// <param name="layerSet"> representing each layer size </param>

    public NeuralNetwork(int[] layerSet)

    {

        this.serialNumber = counter;

        counter++;

        this.fitness = 0;

        this.score = 0;

        this.layerNum = layerSet.Length;

        this.layers = new Layer[layerNum];

        for (int i = 0; i < layerNum -1; i++)

        {

            this.layers[i] = new Layer(layerSet[i], layerSet[i + 1]);

        }

        this.layers[layerNum - 1] = new Layer(layerSet[layerNum - 1], 0);

        this.inputLayer = layers[0];

    }

    /// <summary>

    /// deep copy of neural network

    /// </summary>

    /// <param name="other"></param>

    public NeuralNetwork(NeuralNetwork other)

    {

        this.serialNumber = other.serialNumber;

        this.path = other.path;

        this.fitness = other.fitness;

        this.score = other.score;

        this.layerNum = other.layerNum;

        this.layers = new Layer[layerNum];

        for (int i = 0; i < layerNum; i++)

        {

            this.layers[i] = new Layer(other.layers[i]);

        }

        this.inputLayer = layers[0];

    }

    public override string ToString()

    {

        string output = "\*Neural Net\* " + this.serialNumber +"\nsize - " + this.layerNum + "\nfitness - " + this.fitness + "\n";

        for (int i = 0; i < this.layerNum; i++)

        {

            output += this.layers[i] + ",\n";

        }

        return output;

    }

    /// <summary>

    /// give the neural network initial input values

    /// </summary>

    /// <param name="inputArr"> the input of the net: deltaX, velocityY... </param>

    public void FeedInput(float[] inputArr)

    {

        for (int i=0; i<layers[0].size; i++)

        {

            this.inputLayer.neurons[i].val = inputArr[i];

        }

    }

    /// <summary>

    /// calculate each neuron value using the formula: y = w1\*x1 + w2\*x2... + b

    /// </summary>

    public void FeedForward()

    {

        //cycling through each layer

        for (int layer = 1; layer < this.layerNum; layer++)

        {

            Layer currentLayer = this.layers[layer];

            Layer lastLayer = this.layers[layer - 1];

            //cycling through each neuron

            for (int neuron = 0; neuron < currentLayer.size; neuron++)

            {

                Neuron currentNeuron = currentLayer.neurons[neuron];

                //sum of weights and inputs

                float value = currentLayer.bias;

                for(int weight = 0; weight < lastLayer.size; weight++)

                {

                    value += lastLayer.neurons[weight].val \* lastLayer.neurons[weight].weights[neuron];

                }

                currentNeuron.val = (float)Math.Tanh(value);

            }

        }

    }

    /// <summary>

    /// based on chance the function will change or not each weight in the net

    /// </summary>

    public void Mutate()

    {

        float randomNumber;

        //cycling through each layer

        for (int layer = 0; layer < this.layerNum; layer++)

        {

            Layer currentLayer = this.layers[layer];

            //mutate bias value

            randomNumber = UnityEngine.Random.Range(0f, 50f);

            if (randomNumber <= 2f)

            { //if 1

              //flip sign of bias

                currentLayer.bias \*= -1f;

            }

            else if (randomNumber <= 4f)

            { //if 2

              //pick random weight between -10 and 10

                currentLayer.bias = UnityEngine.Random.Range(-1f, 1f);

            }

            else if (randomNumber <= 6f)

            { //if 3

              //randomly increase by 0% to 100%

                float factor = UnityEngine.Random.Range(0f, 1f) + 1f;

                currentLayer.bias \*= factor;

            }

            else if (randomNumber <= 8f)

            { //if 4

              //randomly decrease by 0% to 100%

                float factor = UnityEngine.Random.Range(0f, 1f);

                currentLayer.bias \*= factor;

            }

            //cycling through each neuron

            for (int neuron = 0; neuron < currentLayer.size; neuron++)

            {

                Neuron currentNeuron = currentLayer.neurons[neuron];

                //cycling through each weight

                for(int weight = 0; weight < currentNeuron.size; weight++)

                {

                    //mutate weight value

                    randomNumber = UnityEngine.Random.Range(0f, 50f);

                    if (randomNumber <= 2f)

                    { //if 1

                      //flip sign of weight

                        currentNeuron.weights[weight] \*= -1f;

                    }

                    else if (randomNumber <= 4f)

                    { //if 2

                      //pick random weight between -1 and 1

                        currentNeuron.weights[weight] = UnityEngine.Random.Range(-0.5f, 0.5f);

                    }

                    else if (randomNumber <= 6f)

                    { //if 3

                      //randomly increase by 0% to 100%

                        float factor = UnityEngine.Random.Range(0f, 1f) + 1f;

                        currentNeuron.weights[weight] \*= factor;

                    }

                    else if (randomNumber <= 8f)

                    { //if 4

                      //randomly decrease by 0% to 100%

                        float factor = UnityEngine.Random.Range(0f, 1f);

                        currentNeuron.weights[weight] \*= factor;

                    }

                }

            }

        }

    }

    /// <summary>

    /// get the boolean output of the net jump or not

    /// </summary>

    /// <returns> return true if the last neuron value is greater then 0.6</returns>

    public bool GetOutput()

    {

        float result = this.layers[this.layerNum - 1].neurons[0].val;

        return result > 0.6;

    }

    /// <summary>

    /// Compare two neural networks and sort based on fitness

    /// </summary>

    /// <param name="other">Network to be compared to</param>

    /// <returns></returns>

    public int CompareTo(NeuralNetwork other)

    {

        if (other == null) return 1;

        if (fitness > other.fitness)

            return 1;

        else if (fitness < other.fitness)

            return -1;

        else

            return 0;

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// movement script for a bird controlled by the player

/// </summary>

public class FlappyInputController : MonoBehaviour

{

    //how many pipes passed

    public int score;

    //reference to physics component

    protected Rigidbody2D rb;

    public float jumpSpeed;

    //set to false in children

    public bool isPlayer;

    public bool canDie;

    //does it have animation

    public bool isAnimate;

    //reference animation componenet

    public FlapAnim fa;

    //reference to hearts UI

    public Image[] hearts;

    //used for respawning after a hit was taken

    public Transform startPosition;

    //invinsibility for a short amount of time after a hit was taken

    protected int hitpoints;

    protected float hitCoolDown, maxHitCoolDown = 1f;

    void Awake()

    {

        rb = gameObject.GetComponent<Rigidbody2D>();

        score = 0;

        //initialize health to 3 hitpoints (3 hearts on screen)

        hitpoints = 3;

    }

    void Update()

    {

        //the bird will never move on the x axis

        rb.velocity = new Vector2(0, rb.velocity.y);

        //input for jumping

        if(Input.GetKeyDown(KeyCode.Space) && isPlayer)

            Jump();

        //decerementing from the invinsibility duration

        hitCoolDown -= Time.deltaTime;

    }

    public void Jump()

    {

        //add speed on the y axis

        rb.velocity = new Vector2(rb.velocity.x, jumpSpeed);

        if (isAnimate)

            fa.Flap();

    }

    public virtual void OnTriggerEnter2D(Collider2D collision)

    {

        //passed a pipe's hole

        if (collision.gameObject.tag == "pipeFlag")

            score += 1;

        //hit a pipe

        if (collision.gameObject.tag == "Pipe")

            Hit();

        //hit the floor/ceiling

        if (collision.gameObject.tag == "Finish")

        {

            Hit();

            transform.position = startPosition.position;

            rb.velocity = new Vector2(0, 0);

        }

    }

    public void Hit()

    {

        //set hit animation

        if (isAnimate)

            fa.Blip();

        //prevent double trigger

        if (hitCoolDown > 0)

            return;

        hitCoolDown = maxHitCoolDown;

        //decrement from bird health

        hitpoints--;

        //update UI

        hearts[hitpoints].enabled = false;

        //check if dead

        if (hitpoints == 0)

        {

            //end of the game

            gameObject.SetActive(false);

            if (gameObject.tag == "Player")

                FindObjectOfType<UIManager>().SetPanel("YOU LOST");

            else

                FindObjectOfType<UIManager>().SetPanel("YOU WON!");

        }

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// a movement script for a bird controlled by a simple algorithm

/// </summary>

public class FlappyBot : FlappyInputController

{

    //the next pipe object

    public Transform nextPipe;

    //delta y offset from the pipe's hole from which the bird should jump

    public float offset;

    //the pipes layer

    public LayerMask pipeLayer;

    //set a delay between jumps

    //the bird should not jump too many times in a row it seems unreal

    private float maxDelay = 0.6f, delay;

    /// <summary>

    /// check whether to jump or not, uses next pipe propeties

    /// </summary>

    /// <returns> true if the bird should jump </returns>

    public virtual bool IsJump()

    {

        if (nextPipe != null)

        {

            if (transform.position.y < nextPipe.GetComponent<PipeSetup>().hole.position.y - offset)

                return true;

        }

        else if (transform.position.y < -3 && nextPipe == null)

            return true;

        return false;

    }

    ///<summary>

    /// update nextPipe variable by shooting a ray and discovering

    /// each time a new pipe in front of the bird

    ///</summary>

    public void CheckForPipe()

    {

        //the length of the ray

        //the length should be exectly the space between pipes

        //in order to discover new pipe each passage

        int length = 14;

        //shoot a ray forward

        RaycastHit2D hit1 = Physics2D.Raycast(transform.position, transform.right, length, pipeLayer);

        //show the ray on screen

        Debug.DrawRay(transform.position, transform.right \* length, Color.green, 0.01f);

        //if the ray found a pipe object

        if (hit1)

        {

            if (hit1.transform.gameObject.tag == "pipeFlag")

                nextPipe = hit1.transform.parent;

            else

                nextPipe = hit1.transform;

        }

    }

    private void Update()

    {

        //update next pipe

        CheckForPipe();

        //check if should jump and allowed with delay

        if (IsJump() && delay < 0)

        {

            Jump();

            //restore delay its initial value

            delay = maxDelay;

        }

        delay -= Time.deltaTime;

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// movement script for a bird controlled by a neural network

/// </summary>

public class FlappyNeural : FlappyBot

{

    //used for game scene to check if this is the enemy bird

    public bool isEnemyPlayer;

    //the neural network of the bird

    public NeuralNetwork net;

    //used for loading a neural network from a saved txt file

    public TextAsset netTextSource;

    //net input

    float deltaX, velocityY, disHigh, disLow, pipeSpeed;

    //time spent alive during this scene

    float aliveDuration;

    //how many pipes to pass to increase speed

    //used for training purposes

    private int pipesToSwitch;

    private void Start()

    {

        //if this is the enemy bird playing in the game scene

        //we should load its own neural network

        if (isEnemyPlayer)

            InitNet();

        aliveDuration = 0;

    }

    /// <summary>

    /// initialize a new neural network for this bird using a netset

    /// </summary>

    /// <param name="netSet"> an array that represents how many neurons in each layer</param>

    public void InitNet(int[] netSet)

    {

        net = new NeuralNetwork(netSet);

    }

    /// <summary>

    /// initialize a new neural network for this bird using another net

    /// </summary>

    /// <param name="other"> nother neural network that we want to copy </param>

    public void InitNet(NeuralNetwork other)

    {

        net = other;

    }

    /// <summary>

    /// initialize a new neural network for this bird using another net that is saved on a txt file

    /// </summary>

    public void InitNet()

    {

        net = NetSerialize.Deserialize(netTextSource);

    }

    /// <summary>

    /// check whether to jump by neural network output

    /// </summary>

    /// <returns> true if the bird should jump given the inputs</returns>

    public override bool IsJump()

    {

        //when pipe is not found jump in place

        if (transform.position.y < -3 && nextPipe == null)

            return true;

        else if(nextPipe != null)

        {

            //the middle of the next hole

            Transform hole = nextPipe.GetComponent<PipeSetup>().hole.transform;

            //the upper edge of the next hole

            Transform upEdge = nextPipe.GetComponent<PipeSetup>().upperEdge.transform;

            //the lower edge of the next hole

            Transform lowEdge = nextPipe.GetComponent<PipeSetup>().lowerEdge.transform;

            //the next pipe current speed

            pipeSpeed = nextPipe.GetComponent<PipeSetup>().speed;

            //distance in horizontal exis from the hole

            deltaX = hole.position.x - transform.position.x;

            //speed in the y axis (falling speed)

            velocityY = rb.velocity.y;

            //distance in the y axis from upper edge of the hole

            disHigh = upEdge.position.y - transform.position.y;

            //distance in the y axis from lower edge of the hole

            disLow = transform.position.y - lowEdge.position.y;

            //the input array that will feed the network

            float[] input = { deltaX, velocityY, disHigh, disLow, pipeSpeed};

            if (input != null)

            {

                //input the network the input array

                net.FeedInput(input);

                //calculate all neurons

                net.FeedForward();

            }

            //return the neural net output

            return net.GetOutput();

        }

        return false;

    }

    void Update()

    {

        //play until reached to train target

        if (FindObjectOfType<TrainingManager>() != null)

            if (FindObjectOfType<TrainingManager>().targetScore < score)

                return;

        //add to time spent alive during this generation

        aliveDuration += Time.deltaTime;

        //update next pipe

        CheckForPipe();

        if (IsJump())

            Jump();

        //decerementing from the invinsibility duration

        hitCoolDown -= Time.deltaTime;

    }

    /// <summary>

    /// the death of a bird game object in the training scene

    /// </summary>

    public void TrainedBirdDeath()

    {

        print("TrainedBirdDeath");

        net.fitness = aliveDuration;

        net.score = score;

        gameObject.SetActive(false);

        FindObjectOfType<TrainingManager>().GenEnd();

    }

    public override void OnTriggerEnter2D(Collider2D collision)

    {

        //check if occurs in the training scene

        if (FindObjectOfType<TrainingManager>() &&

            (collision.tag == "Pipe" ||  collision.tag == "Finish"))

            TrainedBirdDeath();

        //if occurs in game scene the death is the same as player

        else

            base.OnTriggerEnter2D(collision);

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEditor;

using UnityEngine.SceneManagement;

using TMPro;

/// <summary>

/// management script for the training process

/// </summary>

public class TrainingManager : MonoBehaviour

{

    //several training options

    public bool generateFromBest, isStartOnPlay;

    //the generation number

    public static int genNum;

    //change rate of the genetic algorithm how many birds to erase each iteration

    public int changeRate;

    //the target score

    public int targetScore;

    //iterations limit

    public int iterations;

    //the current highest score

    public int highScore;

    //put the high score on screen

    public TextMeshProUGUI scoreUI;

    //reference to the bird spawner script

    public BirdSpawner bs;

    public PipeSpawner ps;

    //the current birds list

    //public static List<GameObject> birds;

    //the current birds net list

    public static List<NeuralNetwork> nets;

    //the cuurent most succesful neural network

    public NeuralNetwork bestNet;

    public GameObject panel;

    //called each new training scene iteration

    private void Awake()

    {

        if (genNum == 0)

        {

            panel.SetActive(true);

        }

        else

        {

            StartGeneration();

        }

    }

    public void StartGeneration()

    {

        //increment generation

        genNum++;

        //open a new excel file if this is the first generation

        if (genNum == 1 && ExelFormat.isPathSet)

            ExelFormat.OpenCSVFile();

        //birds = new List<GameObject>();

        //check if we have already got a nets list

        //if we have nets we will spawn the new ones based on previous

        if (nets != null)

        {

            bs.Spawn(nets);

            print("spawn last nets");

        }

        else

        {

            nets = new List<NeuralNetwork>();

            if (generateFromBest)

                bs.Spawn(NetSerialize.Deserialize());

            else

                bs.Spawn();

        }

        //enable pipe spawner

        ps.run = true;

    }

    /// <summary>

    /// after a spawning a new generation this function is called

    /// to collect the new neural nets

    /// </summary>

    public void InitNetsList()

    {

        foreach (FlappyNeural fp in FindObjectsOfType<FlappyNeural>())

        {

            nets.Add(fp.net);

        }

    }

    /// <summary>

    /// generate new nets list for next generation

    /// </summary>

    private void GenerateNewNets()

    {

        for(int i = 0; i < changeRate; i++)

        {

            //delete the worst nets

            nets.RemoveAt(i);

            //duplicate the best nets

            NeuralNetwork mutatedNet = new NeuralNetwork(nets[nets.Count - 1]);

            //and mutate them to discover even more succesful nets

            mutatedNet.Mutate();

            nets.Add(mutatedNet);

        }

    }

    /// <summary>

    /// run all generation ending functions

    /// </summary>

    public void GenEnd()

    {

        foreach(FlappyNeural bird in FindObjectsOfType<FlappyNeural>())

        {

            if (bird.gameObject.activeInHierarchy)

                return;

        }

        //sort nets by fitness

        nets.Sort();

        //get current generation best net

        NeuralNetwork best = nets[nets.Count - 1];

        print(nets.Count);

        bestNet = best;

        //if we get a new high score we will save a serialized text

        //file representing the current net

        if (best.score > highScore)

        {

            //update the high score and the best net

            highScore = best.score;

            bestNet = best;

            //NetSerialize.Serialize(bestNet, genNum);

        }

        //update high score UI accordingly

        scoreUI.text = "High Score:" + highScore;

        //add new line to the excel file

        if (ExelFormat.isPathSet)

            ExelFormat.AddLine(genNum, best.score);

        //the genetic algorithm to generate new nets to next generation

        GenerateNewNets();

        //should wait a while to let the rest of the training process run

        StartCoroutine("Wait");

    }

    /// <summary>

    /// wait 1 sec before reloading

    /// </summary>

    /// <returns></returns>

    IEnumerator Wait()

    {

        yield return new WaitForSeconds(1);

        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEditor;

using UnityEngine.SceneManagement;

using TMPro;

/// <summary>

/// management script for the training process

/// </summary>

public class TrainingManager : MonoBehaviour

{

    //several training options

    public bool generateFromBest, isStartOnPlay;

    //the generation number

    public static int genNum;

    //change rate of the genetic algorithm how many birds to erase each iteration

    public int changeRate;

    //the target score

    public int targetScore;

    //iterations limit

    public int iterations;

    //the current highest score

    public int highScore;

    //put the high score on screen

    public TextMeshProUGUI scoreUI;

    //reference to the bird spawner script

    public BirdSpawner bs;

    public PipeSpawner ps;

    //the current birds list

    //public static List<GameObject> birds;

    //the current birds net list

    public static List<NeuralNetwork> nets;

    //the cuurent most succesful neural network

    public NeuralNetwork bestNet;

    public GameObject panel;

    //called each new training scene iteration

    private void Awake()

    {

        if (genNum == 0)

        {

            panel.SetActive(true);

        }

        else

        {

            StartGeneration();

        }

    }

    public void StartGeneration()

    {

        //increment generation

        genNum++;

        //open a new excel file if this is the first generation

        if (genNum == 1 && ExelFormat.isPathSet)

            ExelFormat.OpenCSVFile();

        //birds = new List<GameObject>();

        //check if we have already got a nets list

        //if we have nets we will spawn the new ones based on previous

        if (nets != null)

        {

            bs.Spawn(nets);

            print("spawn last nets");

        }

        else

        {

            nets = new List<NeuralNetwork>();

            if (generateFromBest)

                bs.Spawn(NetSerialize.Deserialize());

            else

                bs.Spawn();

        }

        //enable pipe spawner

        ps.run = true;

    }

    /// <summary>

    /// after a spawning a new generation this function is called

    /// to collect the new neural nets

    /// </summary>

    public void InitNetsList()

    {

        foreach (FlappyNeural fp in FindObjectsOfType<FlappyNeural>())

        {

            nets.Add(fp.net);

        }

    }

    /// <summary>

    /// generate new nets list for next generation

    /// </summary>

    private void GenerateNewNets()

    {

        for(int i = 0; i < changeRate; i++)

        {

            //delete the worst nets

            nets.RemoveAt(i);

            //duplicate the best nets

            NeuralNetwork mutatedNet = new NeuralNetwork(nets[nets.Count - 1]);

            //and mutate them to discover even more succesful nets

            mutatedNet.Mutate();

            nets.Add(mutatedNet);

        }

    }

    /// <summary>

    /// run all generation ending functions

    /// </summary>

    public void GenEnd()

    {

        foreach(FlappyNeural bird in FindObjectsOfType<FlappyNeural>())

        {

            if (bird.gameObject.activeInHierarchy)

                return;

        }

        //sort nets by fitness

        nets.Sort();

        //get current generation best net

        NeuralNetwork best = nets[nets.Count - 1];

        print(nets.Count);

        bestNet = best;

        //if we get a new high score we will save a serialized text

        //file representing the current net

        if (best.score > highScore)

        {

            //update the high score and the best net

            highScore = best.score;

            bestNet = best;

            //NetSerialize.Serialize(bestNet, genNum);

        }

        //update high score UI accordingly

        scoreUI.text = "High Score:" + highScore;

        //add new line to the excel file

        if (ExelFormat.isPathSet)

            ExelFormat.AddLine(genNum, best.score);

        //the genetic algorithm to generate new nets to next generation

        GenerateNewNets();

        //should wait a while to let the rest of the training process run

        StartCoroutine("Wait");

    }

    /// <summary>

    /// wait 1 sec before reloading

    /// </summary>

    /// <returns></returns>

    IEnumerator Wait()

    {

        yield return new WaitForSeconds(1);

        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// pipe (the main obstacle) setup and movement script

/// holds public references to it's hole importent positions

/// </summary>

public class PipeSetup : MonoBehaviour

{

    //the script that spawned this gameobject

    public PipeSpawner spawner;

    public float speed;

    //the random range of the hole places

    public float lowR, highR;

    //the pipe's bottom and top colliders

    public Collider2D col1, col2;

    //several importent positions of the hole

    //used by other scripts

    public Transform hole, trigger, upperEdge, lowerEdge;

    private void Awake()

    {

        Setup();

    }

    /// <summary>

    /// chooses a random place to put the pipe's hole

    /// moves the collider accordingly

    /// </summary>

    private void Setup()

    {

        //moving the hole object

        hole.localPosition = new Vector2(0 ,Random.Range(lowR, highR));

        trigger.localPosition = hole.localPosition;

        //moving the colliders

        col1.offset = new Vector2(0, (-0.5f + hole.localPosition.y) - (hole.localScale.y / 2));

        col2.offset = new Vector2(0, (0.5f + hole.localPosition.y) + (hole.localScale.y / 2));

    }

    private void Update()

    {

        //moving the pipe at given speed

        transform.Translate(new Vector3(-speed \* Time.deltaTime, 0, 0));

        //end of screen -> self destruct

        if (transform.position.x < -25)

            Destroy(gameObject);

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PipeSpawner : MonoBehaviour

{

    public bool run;

    public float speed;

    public float increaseSpeedRate, increaseSpeedDelay;

    private float speedTimer, delayTimer;

    public GameObject pipe;

    private float maxSwitchDelay = 1, switchDelay = 1;

    public float delay;

    void Start()

    {

        speedTimer = increaseSpeedDelay;

        //run = true;

        delayTimer = delay;

    }

    private void Update()

    {

        delayTimer -= Time.deltaTime;

        if (delayTimer < 0)

        {

            delayTimer = delay;

            SpawnPipe();

        }

        speedTimer -= Time.deltaTime;

        if (speedTimer < 0)

        {

            IncreaseSpeed(increaseSpeedRate);

            speedTimer = increaseSpeedDelay;

        }

        switchDelay -= Time.deltaTime;

    }

    public void IncreaseSpeed(float amount)

    {

        if (switchDelay > 0)

            return;

        switchDelay = maxSwitchDelay;

        delay = (speed \* delay) / (speed + amount);

        speed += amount;

        //update already instantitaed pipes

        PipeSetup[] pipes = FindObjectsOfType<PipeSetup>();

        foreach (PipeSetup pipe in pipes)

        {

            //this class spawned this pipe

            if (pipe.spawner == this)

                pipe.speed = speed;

        }

    }

    public void SpawnPipe()

    {

        pipe.GetComponent<PipeSetup>().speed = speed;

        pipe.GetComponent<PipeSetup>().spawner = this;

        if (run)

            Instantiate(pipe, transform.position, transform.rotation);

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine.SceneManagement;

using UnityEngine;

/// <summary>

/// scales time, also includs some scene management

/// </summary>

public class TimeScaler : MonoBehaviour

{

    public float timeScale;

    //used for game starting and -> set time scale to 3

    private bool started;

    private void Start()

    {

        started = false;

    }

    private void Update()

    {

        if (Input.GetKeyDown(KeyCode.R))

            SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);

        if (Input.GetKeyDown(KeyCode.Escape))

        {

            SceneManager.LoadScene(0);

            //destroy all training options related scripts

            //reset the training

            FindObjectOfType<DDOL>().DestroyOnNextLoad();

        }

        if (Input.GetKeyDown(KeyCode.Space) && !started)

        {

            timeScale = 3f;

            started = true;

        }

        if (Input.GetKeyDown(KeyCode.X))

            timeScale = (timeScale + 1) % 10;

        if (Input.GetKeyDown(KeyCode.Alpha0) || Input.GetKeyDown(KeyCode.Keypad0))

            timeScale = 0;

        if (Input.GetKeyDown(KeyCode.Alpha1) || Input.GetKeyDown(KeyCode.Keypad1))

            timeScale = 3;

        if (Input.GetKeyDown(KeyCode.Alpha2) || Input.GetKeyDown(KeyCode.Keypad2))

            timeScale = 4;

        if (Input.GetKeyDown(KeyCode.Alpha3) || Input.GetKeyDown(KeyCode.Keypad3))

            timeScale = 5;

        if (Input.GetKeyDown(KeyCode.Alpha4) || Input.GetKeyDown(KeyCode.Keypad4))

            timeScale = 6;

        if (Input.GetKeyDown(KeyCode.Alpha5) || Input.GetKeyDown(KeyCode.Keypad5))

            timeScale = 7;

        if (Input.GetKeyDown(KeyCode.Alpha6) || Input.GetKeyDown(KeyCode.Keypad6))

            timeScale = 8;

        if (Input.GetKeyDown(KeyCode.Alpha7) || Input.GetKeyDown(KeyCode.Keypad7))

            timeScale = 9;

        if (Input.GetKeyDown(KeyCode.Alpha8) || Input.GetKeyDown(KeyCode.Keypad8))

            timeScale = 10;

        if (Input.GetKeyDown(KeyCode.Alpha9) || Input.GetKeyDown(KeyCode.Keypad9))

            timeScale = 12;

        if (Input.GetKeyDown(KeyCode.F))

            timeScale = 5;

        if (Input.GetKeyDown(KeyCode.U))

            timeScale = 10;

        Time.timeScale = timeScale;

    }

    public void LoadScene(int index)

    {

        SceneManager.LoadScene(index);

    }

    public void ExitGame()

    {

        Application.Quit();

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

using TMPro;

/// <summary>

/// UI manager for game scene

/// </summary>

public class UIManager : MonoBehaviour

{

    public TimeScaler ts;

    public GameObject panel;

    public TextMeshProUGUI EndTitle, startTitle;

    public void SetPanel(string massage)

    {

        panel.SetActive(true);

        EndTitle.text = massage;

        ts.timeScale = 0;

    }

    public void LoadScene(int sceneIndex)

    {

        SceneManager.LoadScene(sceneIndex);

    }

    private void Update()

    {

        if (Input.GetKeyDown(KeyCode.Space))

            startTitle.enabled = false;

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// movement script for bullet (laser or poop prefabs)

/// </summary>

public class BulletMovement : MonoBehaviour

{

    public Rigidbody2D rb;

    public int direction;

    //determine if the player shot it or the bot

    public bool isPlayer;

    //both players cameras

    Camera playerCam, botCam;

    [HideInInspector]

    public float speed;

    void Start()

    {

        rb.velocity = new Vector2(direction \* speed, 0);

        playerCam = PublicReferences.playerCam;

        botCam = PublicReferences.botCam;

        //destroy bullet when out of screen

        Destroy(gameObject, 13);

    }

    private void Update()

    {

        rb.velocity = new Vector2(direction \* speed, 0);

        //check if passed this player's screen

        //and teleport to enemy screen

        if (transform.position.x < (playerCam.transform.position.x - playerCam.orthographicSize) && isPlayer)

        {

            transform.position = new Vector2(botCam.transform.position.x + (botCam.orthographicSize),

                botCam.transform.position.y + (transform.position.y - playerCam.transform.position.y));

        }

        if (transform.position.x > (botCam.transform.position.x + botCam.orthographicSize) && !isPlayer)

        {

            transform.position = new Vector2(playerCam.transform.position.x - (playerCam.orthographicSize),

                playerCam.transform.position.y + (transform.position.y - botCam.transform.position.y));

        }

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// spawnes collectibles

/// </summary>

public class CollectibleSpawner : MonoBehaviour

{

    //each pipe has its own spawn range in front of it

    public Collider2D spawnRange;

    //all the collectible prefabs

    public GameObject[] collectibles;

    //the chances to spawn

    //if 3 then 1 out of 3 attempts will succeed

    public int spawnChance;

    private void OnEnable()

    {

        int randNum = Random.Range(1, spawnChance);

        if (randNum == 1 && !FindObjectOfType<TrainingManager>())

        {

            Vector2 spwnPos = new Vector2(Random.Range(spawnRange.bounds.min.x, spawnRange.bounds.max.x),

                Random.Range(spawnRange.bounds.min.y, spawnRange.bounds.max.y));

            int randIndex = Random.Range(0, collectibles.Length);

            //check if allowed to spawn this type of collectible

            if (randIndex <= 1 && !PublicReferences.speedCollectiblesAllowed)

                return;

            Instantiate(collectibles[randIndex], spwnPos, transform.rotation, transform);

        }

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// script for collecting the game collectibles

/// </summary>

public class Collector : MonoBehaviour

{

    public PipeSpawner myPipSpwn, enemyPipSpwn;

    public GameObject laser, shootPoint;

    public FlappyInputController flappyControl;

    public Image fastIcon, slowIcon;

    //called when colliding with a collectible

    private void OnTriggerEnter2D(Collider2D collision)

    {

        //hit from laser/poop

        if (this.tag == "Player" && collision.tag == "Laser")

            flappyControl.Hit();

        if (this.tag == "Bot" && collision.tag == "Poop")

            flappyControl.Hit();

        //collect

        char[] sep = { '\_' };

        string[] args = collision.tag.Split(sep);

        //not a collectible, return

        if (!args[0].Equals("collect"))

            return;

        //collectible number 0-2

        int colNum = int.Parse(args[1]);

        switch (colNum)

        {

            case 0:

                StartCoroutine("FastSpeed");

                break;

            case 1:

                StartCoroutine("SlowSpeed");

                break;

            case 2:

                ShootLaser();

                break;

        }

        Destroy(collision.gameObject);

    }

    //shoot laser or poop after collecting laser reward

    public void ShootLaser()

    {

        Instantiate(laser.transform, shootPoint.transform.position, laser.transform.rotation).

            GetComponent<BulletMovement>().speed = myPipSpwn.speed;

    }

    public void DestroyAllSpeedCollectibles()

    {

        GameObject[] arr = GameObject.FindGameObjectsWithTag("collect\_0");

        foreach (GameObject gm in arr)

            Destroy(gm);

        arr = GameObject.FindGameObjectsWithTag("collect\_1");

        foreach (GameObject gm in arr)

            Destroy(gm);

    }

    IEnumerator FastSpeed()

    {

        DestroyAllSpeedCollectibles();

        fastIcon.enabled = true;

        PublicReferences.speedCollectiblesAllowed = false;

        enemyPipSpwn.IncreaseSpeed(1f);

        yield return new WaitForSeconds(7f);

        enemyPipSpwn.IncreaseSpeed(1f);

        yield return new WaitForSeconds(10f);

        enemyPipSpwn.IncreaseSpeed(-1f);

        yield return new WaitForSeconds(7f);

        enemyPipSpwn.IncreaseSpeed(-1f);

        PublicReferences.speedCollectiblesAllowed = true;

        fastIcon.enabled = false;

    }

    IEnumerator SlowSpeed()

    {

        DestroyAllSpeedCollectibles();

        slowIcon.enabled = true;

        PublicReferences.speedCollectiblesAllowed = false;

        myPipSpwn.IncreaseSpeed(-1f);

        yield return new WaitForSeconds(15f);

        myPipSpwn.IncreaseSpeed(1f);

        PublicReferences.speedCollectiblesAllowed = true;

        slowIcon.enabled = false;

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// animation script for background

/// </summary>

public class BackgroundAnimation : MonoBehaviour

{

    public float speed;

    public float waitToTerminate;

    public Transform spawnPoint;

    public Transform trigger;

    private bool alreadySpawned = false;

    void Update()

    {

        transform.Translate(new Vector3(-speed \* Time.deltaTime, 0, 0));

        if (transform.position.x < trigger.position.x && !alreadySpawned)

        {

            Destroy(gameObject, waitToTerminate);

            Instantiate(this, spawnPoint.position, spawnPoint.rotation);

            alreadySpawned = true;

        }

    }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// a class that will preserve certein gameobject over scene loading

/// </summary>

public class DDOL : MonoBehaviour

{

    void Awake()

    {

        //find the specified game objects with tag

        GameObject[] objs = GameObject.FindGameObjectsWithTag("DDOS");

        //delete new spawned game objects that already exist

        if (objs.Length > 1)

            Destroy(this.gameObject);

        DontDestroyOnLoad(this.gameObject);

    }

    //reset all training gameobjects

    //called by game manager when pressed esc

    public void DestroyOnNextLoad()

    {

        //reset generations

        TrainingManager.genNum = 0;

        //destroy this game object

        Destroy(gameObject);

    }

}