

תקציר מנהלים:

סקירה כללית:

פרויקט זה מציג שימוש מלא של רשות ניורונים רב שכבותית שנבנתה באמצעות שפת **Python** וספריית **NumPy** בלבד, ללא הסתמכות על ספריות למידה عمוקה חיצונית כמו PyTorch. מטרת הפרויקט היא למדוד בצורה מעמיקה את המתמטיקה שמאחוריה למדית מכונה ויישום עקרונות הנדסת תוכנה מתקדמים.

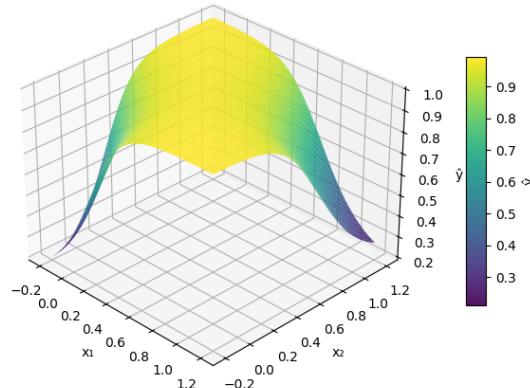
dagשים טכניים:

- ארQUITטורה מודולרית:** הקוד בנוי בצורה מונחית עצמים עם הפרדה מלאה בין שכבות, פונקציות אקטיבציה, וניהול הרשת.
- שימוש דני של אלגוריתם הפצה לאחרו:** תוך כדי שימוש בכלל השרשראת לחישוב גרדיאנטים ועדכון המשקלות והרטויות.
- גימות תפוצולית:** בשל המודולריות ניתן להשתמש במגוון פונקציות אקטיבציה ו-Loss למטרות למידה שונות של הרשת.
- פתרון בעיה שאינה ליניארית:** השתמשתי ברשת כדי לפתור את בעית ה-XOR, הבחירה בעיה זו היא כדי להדגים את יכולת הרשת לפתור בעיה שאינה ליניארית באמצעות שכבות נסתרות.

תוצאות:

באימון על בעית ה-XOR המandal הגיע לרשת דיקון גבואה עם ביטחון של 99.2% בחיזיו ושגיאה מזערית של 0.0002.

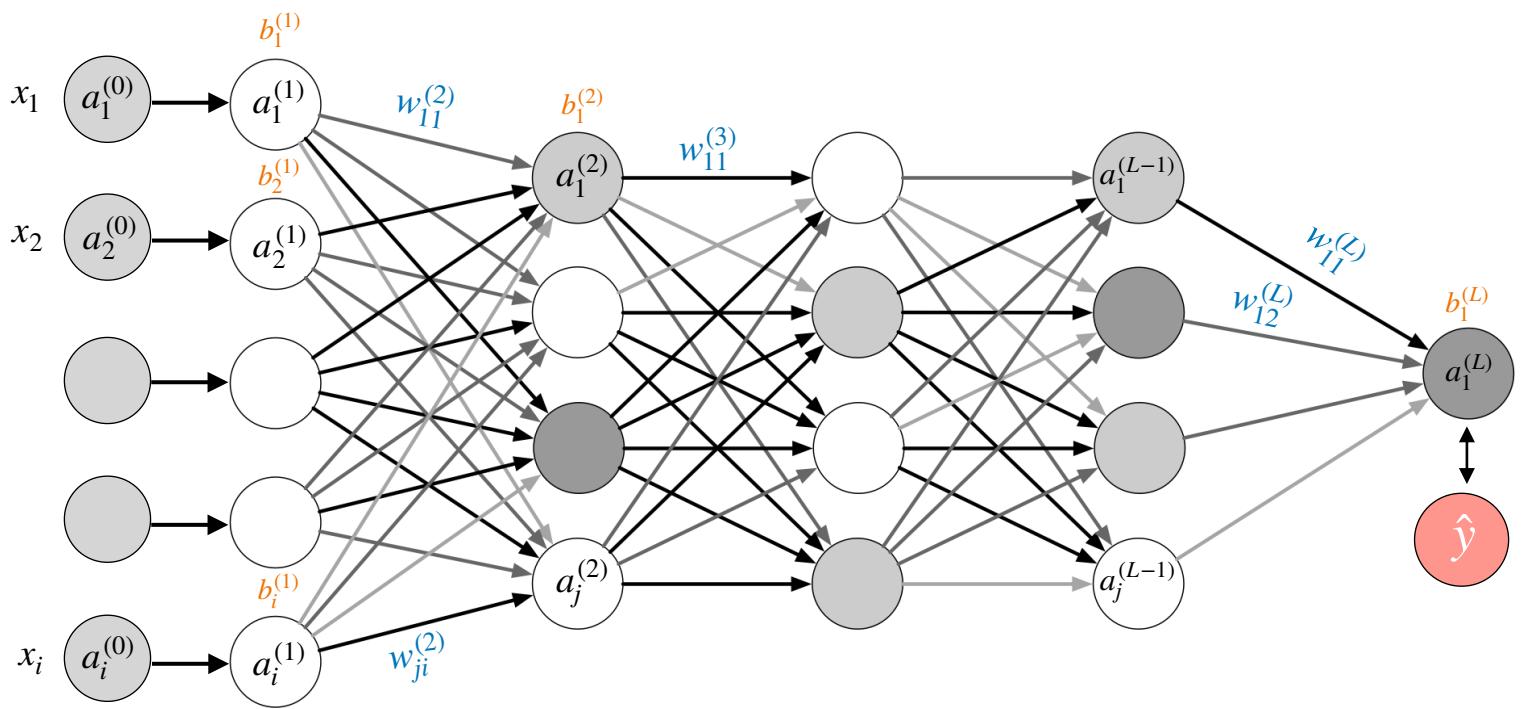
Decision Surface



- דוגמת הרצה

תוכן עניינים:

1. יסודות וכתיב מתמטי
2. תהליכי החיזוי: הפצה קדימה
3. מדידת ביצועים ואופטימיזציה
4. מנגןון הלמידה: הפצה לאחרו
5. ארQUITטורות התוכנה ומימוש
6. ניסוי הרכבה, דוגמאות הרצה ופתרון בעית ה-XOR.



כתב

אציג את הכתיבה הנהוג שבו אشتמש לאורך המבנה.
נסמן ב- L את מספר השכבות בראשת, בדוגמה שלנו יש בראשת 5 שכבות, שכבה קלט, 3 שכבות נסתרות ושכבה פלט.

אם נרצה להפנות לשכבה ספציפית נסמן $\{1, 2, 3, 4, \dots, L\} \in l$.

עבור כל שכבה - $a^{(l)}$ הוא וקטור הערכים של הנוירונים בשכבה ה- l (אחרי הפעוקצית האקטיבציה) ו- $a_i^{(l)}$ הוא הערך של נוירון ספציפי i בשכבה ה- l .

בහינתן מטריצת קלט עם n עמודות, נסמן ב- x את הערך ה- i של הקלט. ככלומר מתקיים $.a_i^{(0)} = x_i$

הסימן $W^{(l+1)}$ מסמן את מטריצת המשקלים של השכבה ה- l , והוא מטריצה בגודל $n^{(l+1)} \times n^{(l)}$ כאשר $n^{(l)}$ מייצג את מספר הנוירונים בשכבה ה- l .

אם נרצה להתייחס למשקל ספציפי המחבר בין הנוירון ה- i בשכבה ה- l לבין הנוירון ה- j בשכבה ה- $(l+1)$ נסמן בתוור $w_{ji}^{(l+1)}$.

באופן זהה נסמן $b_i^{(l+1)}$ עבור סימון ההטייה (bias) של הנוירון ה- i בשכבה ה- l או של וקטור כל השכבה ה- l בהתאם.

בדוק כפי ש- $a^{(l)}$ מסמן את הערכים של הנוירונים בשכבה ה- l , הסימון $z^{(l+1)}$ ו- $z_i^{(l+1)}$ יסמנו את הערך לפני פונקציית האקטיבציה - ככלומר המשקל המשוכפל של הנוירונים, המשקלים וההטיות מהשכבה הקודמת.

נסמן ב- \hat{y} את הוקטור ערכים של שהמודל סיק בסוף כל איטרציה - ככלומר $\hat{y} = a^{(L)}$ כאשר L היא השכבה الأخيرة. בנוסף נסמן ב- y את הוקטור של הערכים האמתיים (הרצויים).

שכבה נוירוניים - הפעלה קדימה

כל שכבה $i+1$ נחשבת שכבה צפופה, כלומר כל נוירון בשכבה מחובר לכל נוירון אחר בשכבה $i+1$ בעורף "משקל". נסתכל למשל על הנוירון $a_1^{(2)}$ מהדיאגרמה:

הסכום של הנוירון (וכל נוירון בכלל) הוא סכום שתלו依 ליניארית ב:

$$1. \text{ ערכי הנוירונים שמחוברים אליו, במקרה שלנו} - a_1^{(1)} \dots a_i^{(1)}$$

$$2. \text{ ערכי המשקלות שמחוברים אותו לנוירון} - w_{11}^{(2)} \dots w_{ji}^{(2)}$$

$$3. \text{ ערך הנטיה של הנוירון: } b_1^{(2)}$$

הנוסחה לסכום הערך היא:

$$a_1^{(1)} * w_{11}^{(2)} + a_2^{(1)} * w_{12}^{(2)} + \dots + a_i^{(1)} * w_{1i}^{(2)} + b_1^{(2)}$$

$$\text{או בסכום: } b_1^{(2)} + \sum_{n=1}^i a_n^{(1)} * w_{1n}^{(2)}$$

בפועל כל ערך של נוירון בערך המשקל המתאים, נסuum אותו ולבסוף נוסיף את ערך הנטיה, הסכום שמתתקבל הוא בעצם $\sigma^{(2)}$, הסכום זהה הוא הערך אותו כל שכבה מפעילה קדימה לשכבה הבאה, והוא בעצם הבסיס שאיתו נחשב את ערכי הנוירונים הבאים.

לבסוף, השתמש בשכבת אקטיבציה כדי להפעיל פונקציה לא ליניארית על הערך שמתתקבל, נסמן אותה ב- σ ונסביר עליה בהמשך. ככלור לאחר הפעלת הפונקציה σ על $\sigma^{(2)}$ נקבל: $\sigma^{(2)} = \sigma$ - קיבלנו את ערך הנוירון.

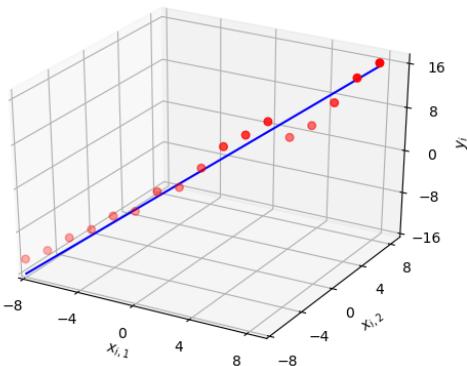
פונקציית האקטיבציה + מוטיבציה

אפשר להסתכל על כל הרשות נוירונים כמודול הסטברוטי מורכב שמטרתו להסיק על סיטואציה שנראית אקראית לחלוון על ידי מציאת קשרים שלרבות אינם נראים לעין. הכל בו משתמשים כדי להתאים את ההסקות של המודל הוא שינוי המשקלים והנטיות בצורה כאשת שתקרב אותנו לאומדן הרצויים בכל איטרציה של לימוד (נרחיב על כך בהמשך).

בוגמה יש 3 שכבות נסתרות בעלות 4 נוירונים, שכבת קלט בעלת 5 נוירונים ו scavbat פלט בעלת נוירון אחד. זאת אומרת שיש לנו **56 משקלים ו-13 הטיות**, בסך הכל הרשות היא פונקציה עם 69 פרמטרים שונים.

זכור שכל הפעולות שדרושות לחישוב ערכי הנוירונים מtabבשות על פעולות ליניאריות, אם לא נכנס למודול פונקציית אקטיבציה, כל החיזיו של המודל יהיה ליניארי. כיצד בעית XOR כמו הרובה בעיות נפוצות הן אינן ליניאריות, ונרצה שהפונקציה שהמודל "מנחש" לא תהיה ליניארית.

במקרה שלנו אפשר לחוש על הרשות בתור פונקציה במילדי 69, כך שאנו יכולים שעבור כל קלט הפונקציה תחזיר פלט שקרוב כמה שיותר לתוכאה הרצואה. אם הפונקציה הסופית שלנו היא ליניארית אפשר להקיים את זה למציאת ישר שעובר כמו שיטור "קרוב" לנקודות הרצויות, בדומה לוגריסיה ליניארית אבל במילדי 75.75. כМОון שפונקציה כזו לא תהיה מדויקת במקרה שהבעה שלנו אינה ליניארית.



(התמונה מתארת פונקציה במרחב 3 שהפתרון שלה כן ליניארי, אם נחשב על פונקציה במרחב 3 שהפתרון שלה אינו ליניארי, לא נוכל למצוא ישר שעובר כמו שיטור "קרוב" לנקודות הרצויות, ולכן ההסקות לא יהיו מדויקות)

חישוב השגיאה (פונקציית ה-cost)

הזכירנו שכל שיש למודל כדי להתאים את ההסקות של המודל הוא שינוי הפרמטרים של הרשות. ולפני שנדבר איך זה קורה בפועל נבין איך אפשר למדוד כמה קרוב הייתה ההסקה של המודל מערך המטרה.

כפי שראינו בדיאגרמה, שכבת הפלט של הרשות מחזירה במקרה שלנו ערך ייחיד, הוא החיזוי של הרשות. באופן זה במקרה שיש לנו כמה נוירונים בשכבת הפלט, החיזוי יוצג על ידי וקטור שמכיל את כל התוצאות של הרשות בשכבת הפלט.

פונקציית cost מקבלת את הפלטים של הרשות (y - \hat{y}) ואת הפלטים האמתיים (שסימנו ב- y) ומחזירה מספר ייחיד שהמשמעות שלו היא כמה החיזוי היה קרוב לערך המטרה. ככל שהערך של הפונקציה נמוך יותר - כך החיזוי של הרשות יותר מדויק, אם הערך הוא 0 זאת אומרת שקיבלו בדיקות את ערך המטרה.

בשביל התוכנית שلنנו נשתמש בפונקציית MSE שנסמנו אותה ב-C:

$$C(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$$

הפונקציה תקבל את הווקטור התוצאות \hat{y} (במקרה של הדוגמה שלנו זה יהיה מספר יחיד) ואת וקטור הערכים המתאימים שציפינו לקבל ע. לכל i החישוב $(\hat{y}^{(i)} - y^{(i)})^2$ מציג את הסטייה של השערך הרצוי מהערך שהרשעת נתנה. והפלט של ה-C הוא בעצם ממוצע השגיאות של כל הפלטים.

הסיבה שלוקחים את ריבוע הפרש היא:

1. נרצה לעבוד עם ערכים חיוביים
2. הריבוע מודיא שפלט הפונקציה הוא פונקציה רציפה וגזירה (נראה בהמשך למה זה שימושי)
3. גורם לשגיאות גדולות להיות יותר משמעותית מאשר שגיאות קטנות יותר, לומר השגיאה תהיה יחסית.

הגרדיינט השליili - איך הרשות לומדת?

כפי שראינו, המטרה הכללית של הרשות היא **למזרע את ערך ה-C** כמו שיתור קרוב ל-0. אם $0 = (\hat{y}, y)$ זה סימן שככל הפלטים שהתקבלו זהים לערכים שציפינו לקבל.

ນחוור לדוגמה שלנו ונסתכל על הווקטור \hat{y} - הוא ערך שמתබול מהפונקציה $\sigma(z_1^{(L)}) * w_{11}^{(L)} + a_i^{(L-1)} * w_{12}^{(L)} + b_1^{(L)} + \dots + a_j^{(L-1)} * w_{1j}^{(L)}$ וזכור ערך \hat{y} הוא לא אחר מהפונקציה:

אפשר להסביר שהערך של \hat{y} תלוי 3 פרמטרים:
1. ערך **ההתיה** של \hat{y}

2. ערכי **המשקלים** שמתבחברים אליו (כתלות בערך הנוירונים שמתבחברים אליו)
3. ערכי הנוירונים שמתבחברים אליו (כתלות בערכי **המשקלים**)

באופן פשוטי המטרה שלנו היא לגלוות כמה (איזה) שינוי של כל פרמטר כזה מוביל לשינוי בערך ה-C. בambilים אחרות - איזה פרמטרים נצטרך לשנות כדי להוביל לשינוי (וירידה) בערך ה-C ובכך הראות האפקטיבית ביותר.

צחה בעיה, במודל שלנו יש 69 פרמטרים כפי שציינו, אבל ברשותות סטנדרטיות יכולים להיות عشرות אלפי משקלים והטויות! בנוסף לזה, כל שינוי יוביל לשינויים במקומות אחרים: אם נסתכל על 3 האפשרויות שציינו קודם לשינוי הפרמטר \hat{y} - ערכי הנוירונים שמתבחברים אליו (3) תלויים גם הם בהטיות, המשקלים שמתבחברים בכל אחד מהם וגם בערכי הנוירונים שמתבחברים אליהם.

כלומר לכל נוירון יש תלות בכל המשקלים וההטיות של כל הרשות בשכבות שלפניהם.
את הפתרון לבעה אראה בחלק הבא. המטרה היא למצוא וקטור באורך כל המשקלים והטויות, שمراה את היחס בין שינוי של כל אחד מהם לבין השינוי שנוצר בפונקציית ה-C, כדי למזרע אותה בצורה אפקטיבית.
וקטור זה יקרא **הגרדיינט** נסמן אותו ב- ∇C ונשתמש בו על מנת לאמן את המודל.

הപצה לאחר - מציאת הגרדיינט

האלגוריתם שנשתמש למציאת הגרדיינט נקרא "הപצה לאחרו" והוא בנייתו בצורה רקורסיבית ומתרבש על כל השרשרת של משוואות דיפרנציאליות. האלגוריתם מחשב את הגרדיינט של כל משקל והטיה ברשות - ככלומר מחשב את ה"רגישות" של שינוי בכל רכיב צזה לשינוי בערך ה- C אותו נרצה להוריד.

אם שוב נסתכל על הרשות כפונקציה בעלת מאות או אלפי משתני משקלות והטיות כאשר הפלט שלה הוא התוצאה של פונקציית ה- C , אנחנו בעצם נרצה למצוא את **נקודת המינימום** של הפונקציה. למרות שימושה כזאת היא כמעט בהחלטה אפשרית בפני עצמה, נוכל בכל איטרציה לבחור את השינויים שיקדמו **לבירור** המינימום של הפונקציה.

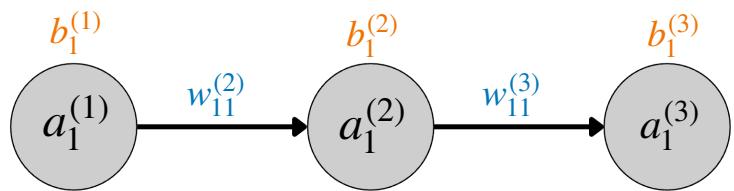
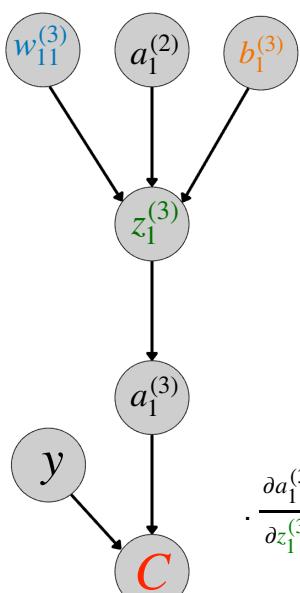
בדומה לבחודה שבפונקציה זו מימדיות אם הנגזרת של נקודה a מסוימת היא שלילית, נוכל לדעת שקיים טעם שבו שהתקדמות לצידה הימני (כלומר $a+1$) טוביל לOIDה בערך הפונקציה $(a)f$, נוכל לחוש על זה ככח שווה למספר הפרמטרים, במקורה שלנו הוא מיד 69.

אם ננסה לדמיין את אותו התהליך בתלת מימד, נוכל להסתכל על האирור הבא: אם נסתכל על הנקודה האדומה, שהיא הערך של פונקציית ה- C , נרצה לבחור את הצד הבא ככח שהירידה בערך תהייה כמו שיווט תלולה ביחס לגדול הצעד, זה יהיה הגרדיינט השלילי - כמו נרצה לשנות את ערכי z,y,x כדי להוביל לשינוי הגדול ביותר כלפי מטה.

*למרות שהוא עוזר להבין את האינטואיציה האמת היא שיש הרבה מגבלות ופרטים טכניים שונים שמקשים על חישוב שכח.

איך ההפצה לאחרו עובדת?

שם הסביר נסתכל על רשת נוירונים פשוטה בהרבה, המכילה 3 שכבות בנוסף לשכבה הקלט, ונורוון בודד בכל שכבה:



תחילה נרצה לדעת מה ההשפעה של שינוי ב- $a_1^{(3)}$ (הלא הוא \hat{y}) על התמונה של C .
בגלל שתי הפרמטרים הללו הון פונקציות, נוכל לחשב את הנגזרת החלקית: $\frac{\partial C}{\partial a_1^{(3)}}$.

אם נסתכל על הדיאגרמה מצד שמאל (כפי שהזכרנו קודם), הערך $a_1^{(3)}$ עצמו מושפע מ- $\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}$.
ככלומר ש כדי למצוא את $a_1^{(3)}$ נרצה למצוא את היחס שלו כלפי $z_1^{(3)}$, או במילים אחרות, את: $\frac{\partial z_1^{(3)}}{\partial w_{11}^{(3)}}$.
באופן לא מפתיע נגלה כי $w_{11}^{(3)}$ תלוי ב- $z_1^{(3)}$ ובאופן זהה, נרצה לבדוק את היחס של המשקל ליחס של $z_1^{(3)}$ בעקבות המשוואת החלקית הבאה: $\frac{\partial z_1^{(3)}}{\partial w_{11}^{(3)}}$.

כלומר בשלב ראשון למציאת היחס בין $\frac{\partial C}{\partial w_{11}^{(3)}}$, נרצה למצוא את היחס בין $a_1^{(3)}$ ו- $z_1^{(3)}$ ואת היחס בין $a_1^{(3)}$ ל-

$$\frac{\partial C}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C}{\partial a^{(L)}}$$

הפתרון של הנגזרות הוא:

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}, \quad \frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}), \quad \frac{\partial C}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

נгла שהיחס בין $\frac{\partial C}{\partial w_{11}^{(3)}}$ יוצא $a_1^{(2)}$, היחס בין תנודה קטנה לערך המשקל - הוא פרופורציוני לערך הנוירון ש לפניו. על שאר המסקנות - אפרט בחלק של מבנה הרשת.

בחלק זהה סיימנו את החלק הראשון של הרקורסיה, מצאנו את $\frac{\partial C}{\partial a_1^{(3)}}$, אבל לא נוכל לעזור כאן - בגלל שהיחס בין $\frac{\partial z^{(3)}}{\partial w_{11}^{(3)}}$ הוא $a_1^{(2)}$, נדרש המשיך ולחקרו איך הנוירונים, המשקלים וההטיות בשכבות שלפני משפיעות על $a_1^{(2)}$, כדי שנוכל לדעת איך זה משפייע על כל שרשרת התלויות שמצאו לפניו, ולבסוף על C .

המשך הרקורסיה

ນמשיך להתקדם בצורה רקורסיבית, בכל שכבה i שנעבור נחשב ונוסף לגרדייאנט הכללי את הנגזרות: $\frac{\partial C}{\partial W^{(l+1)}}$, נחשב את $\frac{\partial C}{\partial a^{(l)}}$ - כדי שנוכל לחשב את $\frac{\partial C}{\partial W^{(l)}}$ בשכבה הבאה. בסופו של התהליך נוכל לדעת מהה כל משקولات והטייה ברשות החל מהשכבה הראשונה ועד האخונה, משפיעה על פונקציית $-C$.

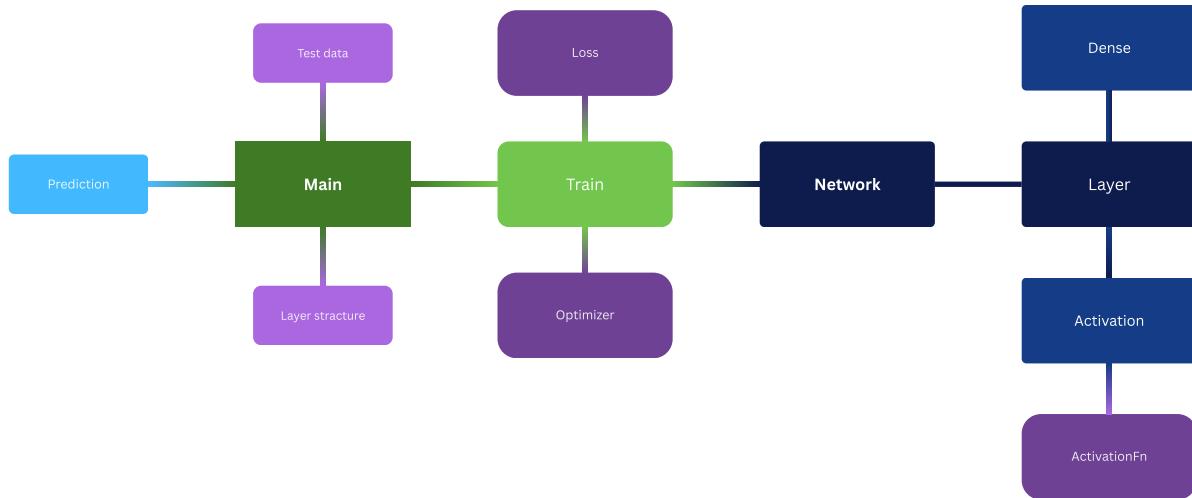
כפי שציינתי לכל שכבה i יוחשב וקטור גרדיאנט שמתאים למשקל השכבה, נסמן אותו ב- ∇C_{l+1} \iff $\frac{\partial C}{\partial W^{(l+1)}}$ בכל איטרציה אימון, נמצא את C לכל שכבה ולאחר מכן נעדכן את המשקלים וההטיות (תלו במודול ובסוג האימון). ונחזור חלילה.

(*) ברשותנו נוירונים עם יותר מנוירון אחד בכל שכבה, החישובים כמעט זהים. חשוב להבין כי ברשות גודלה שינוי של ערך נוירון בודד (בעזרת המשקלים מהשכבות שלפני) יגרור שינוי **במספר** של נוירונים בשכבה הבאה. לכן היחס $\frac{\partial C}{\partial a^{(l)}}$ יהיה **סכום** ההשפעות על C (כਮובן שבל איבר בסכום בפרופורציה למשקל שלו ולכל שאר הנוירונים שבאים אחריו).

המבנה בשפת פיתון

כדי ליבא את הרעיון ששהוצגו בעמודים הקודמים, אציג את הרעיון המרכזים והמבנה של הקוד.

מבנה כללי:



*בקוד שלי ה-Optimizer הוא חלק ממבנה השכבה לשם הפשטות

אובייקט השכבה:

כדי שנוכל להתייחס לסוגי השכבות השונות בצורה נוחה, נשתמש במחלקה Layer כללית שמנה נוכל ליצור מחלקות יורשות לסוגי השכבות השונות.

מחלקה Dense

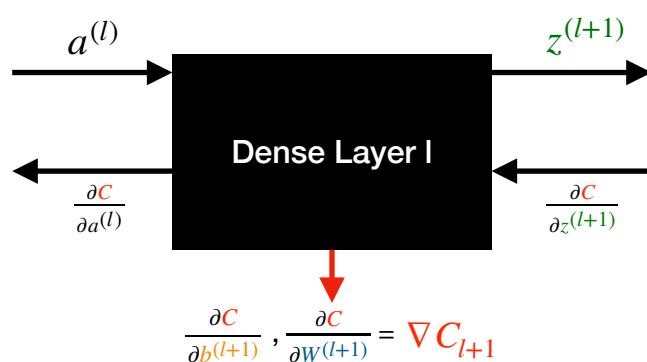
1. אחסון המידע - נרצה שכבה תאהסן את הערכים הבאים:

ערכי הנוירונים שלה ($a^{(l)}$) - וקטור עמده 1 $\times n^{(l)}$
 ערכי המשקלות של השכבה ($W^{(l+1)}$) - מטריצת בגודל $n^{(l)} \times n^{(l+1)}$
 ערכי ההטויות ($b^{(l+1)}$) - וקטור עמده 1 $\times n^{(l+1)}$

2. העברה קדימה - כל שכבה צריכה להחזיר וקטור עמده שמכיל את כל ערכי הבסיס לשכבה הבאה ($z^{(l+1)}$)
 שיוחשב בהתאם למטריצת המשקלות ולוקטור ערבי הנוירונים של השכבה הנוכחית.

3. העברה אחורית - השכבה קיבל כקלט את $\frac{\partial C}{\partial z^{(l+1)}}$ כלומר היחס בין השגיאה לפט שהשכבה החזירה במהלך ההערכה קדימה, ותחזיר את $\frac{\partial C}{\partial a^{(l)}}$ - היחס בין הקלטים שהשכבה קיבלה לשגיאה כדי להמשיך את חישוב יתרה-

▼. במהלך התהילה הפונקציה תחשב גם את $\frac{\partial C}{\partial W^{(l+1)}}$ לעדכון המשקלות של השכבה (העדכון עצמו יקרה ב-Optimizer).



מחלקה ה-Accivation

1. אחסון המידע - שכבה האקטיבציה תארחן את:

שכבת האקטיבציה תארחן 3 משתנים:

ערכי הנירונים של השכבה הבאה ($\zeta^{(l+1)}$)

סוג פונקציית האקטיבציה הנשמר במשתנה `fn_activation` ויסומן ב-`s`

הנגורות של פונקציית האקטיבציה שנשמר במשתנה `fn_prime_activation_fn` ויסומן ב-`s'` (fn_prime_activation_fn). (ActivationFn)

2. העברת קדימה - שכבה האקטיבציה אינה שכבה צפופה, כלומר כל נוירון בשכבה "מחובר" לנויירון אחד בלבד בשכבה הבאה. שכבת האקטיבציה נמצאת בין כל 2 שכבות Dense, תפקידה לקחת את הערכי הפלט של השכבה

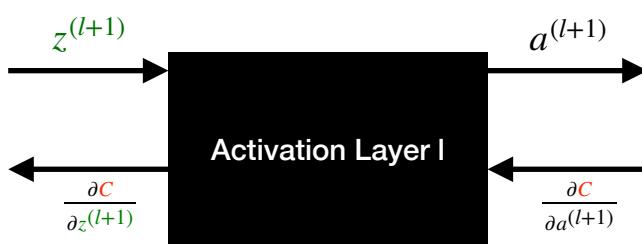
שלפניהם ($\zeta^{(l+1)}$) ולהעביר אותם לפונקציית האקטיבציה, זה יהיה הפלט שלה.

ולומר: $a^{(l+1)} = s(\zeta^{(l+1)})$, הפלט של שכבת האקטיבציה תהיה הקלט של שכבת Dense הבאה.

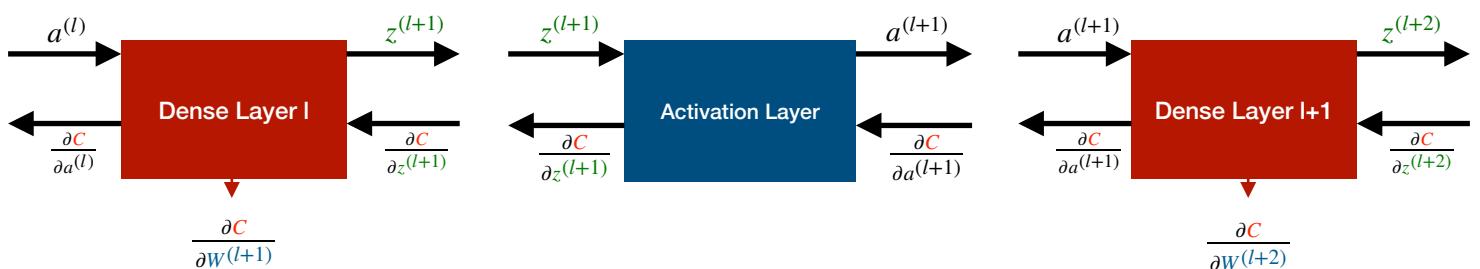
3. העברת אחרת - בדומה להעברה קדימה, בהעברה אחרת נבצע תהליך דומה שבסופו נעביר אחרת את היחס

בין C לנגורת של $\zeta^{(l+1)}$. היא תקבל קלט את היחס בין $a^{(l+1)}$, ובעזרת הנגורות של פונקציית האקטיבציה

תחזיר אחרת את המבוקש. ולומר $\frac{\partial C}{\partial a^{(l+1)}} = \frac{\partial C}{\partial \zeta^{(l+1)}}$, הפלט יהיה הקלט של העברת אחרת של שכבת Dense- $\zeta^{(l+1)}$.



מבנה הרשות:



ניתן לראות איך הפלט של כל שכבה הוא פלט של השכבה שלפניהם.

כעת נראה איך פעולות החישוב שצינתי בחלק התאורטי מתבצעות על ידי כפל מטריצות פשוט.

העברה קדימה:

Dense: נזכיר שהחישוב הוא: $W^{(l+1)} \cdot a^{(l)} + b^{(l+1)} -$ הופך לכפל וחיבור מטריצות: $b_1^{(l+1)} + \sum_{n=1}^i a_i^{(l)} \cdot w_{1i}^{(l+1)}$

акטיבציה: תפעיל את פונקציית האקטיבציה על כל פלט של שכבת Dense: $\sigma(z^{(l+1)}) = a^{(l+1)}$

$$\text{העבירה אחורית: } \frac{\partial C}{\partial a^{(l)}}$$

זהי הנוסחה לחישוב הערות מול השכבה ה- i :

$$\boxed{\frac{\partial C}{\partial a^{(l)}} = \left(W^{(l+1)}\right)^T \frac{\partial C}{\partial z^{(l+1)}}}$$

היחס בין השגיאה $-l^{(l+1)}z$ מתקיים בקלט, מטריצת המשקלות של השכבה שMOVEDה באופן מקומי בה. על ידי מכפלה במטריצת המשקלות המשוחלפת, אנו "מעבירים" את השפעה על הערות לנירונים הקודמים, וממשיכים להפיץ לאחור (כמוון אחריו שהיחסנו את הגרדיינט של השכבה).

בפועל נבצע כפל מטריצות בין הפלט מהשכבה הקודמת, למטריצת המשקלות המשוחלפת.

$$\text{חישוב הגרדיינט: } \frac{\partial C}{\partial W^{(l+1)}}$$

זהי הנוסחה לחישוב הגרדיינט עבור השכבה ה- i :

$$\boxed{\nabla C_{l+1} = \frac{\partial C}{\partial w^{(l+1)}} = \frac{\partial z^{(l+1)}}{\partial w^{(l+1)}} \frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} \frac{\partial C}{\partial a^{(l+1)}}}$$

אציג את המסקנות של הנזירות:

$$\boxed{\frac{\partial z^{(l+1)}}{\partial w^{(l+1)}} = a^{(l)}, \quad \frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} = \sigma'(z^{(l+1)}), \quad \frac{\partial C}{\partial a^{(l+1)}} = \left(W^{(l+2)}\right)^T \frac{\partial C}{\partial z^{(l+2)}}}$$

(ניתן לראות שהנגזרת של השגיאה ביחס לערכי השכבה שונה מאשר מה שהציגו קודם, הסיבה לכך היא שבשכבה ה- L (השכבה האחורית) בה התעסקנו קודם, חישוב $\frac{\partial C}{\partial a^{(l+1)}}$ הוא ישיר, ופה הוא מועבר ותלי בשכבה ה- $(L-1)$)

כדי לחשב את יחס השגיאה למשקלות עבור שכבה i , לפי כלל השרשרת והרכבה:

$$\boxed{\frac{\partial C}{\partial w^{(l+1)}} = \frac{\partial z^{(l+1)}}{\partial w^{(l+1)}} \frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} \frac{\partial C}{\partial a^{(l+1)}} = a^{(l+1)} \sigma'(z^{(l+1)}) \odot \left(W^{(l+2)}\right)^T \frac{\partial C}{\partial z^{(l+2)}}}$$

1. את הרכיב $\frac{\partial C}{\partial a^{(l+1)}}$ כבר ראיינו בחישוב של העבירה אחורית. **חישוב זה יבוצע בשכבה ה- $i+1$ ויתקבל בקלט בשכבת האקטיבציה.**

2. לאחר מכן, מכפלה של כל איבר ב- $\frac{\partial C}{\partial a^{(l+1)}}$ בנגזרת של פונקציית האקטיבציה (σ') תתן את: **חישוב זה יבוצע בשכבת האקטיבציה ותקבל בקלט בשכבה ה- i .**

3. לבסוף, נכפול ב- $a^{(l)}$ ונקבל את: $\frac{\partial C}{\partial W^{(l+1)}}$ כפי שרצינו.

כלומר, החישוב היחיד שכבת Dense נדרש לחישוב הגרדיינט הוא כפל של המטריצה $\frac{\partial C}{\partial W^{(l+1)}}$ ב- $a^{(l)}$.
(לא ארchip בעל חישוב הגרדיינט של ההטיות אך ההליך דומה)

שאר המחלקות ו” הפרדת הרשות ”:

הרעיוון הכללי הוא לחלק את תחומי האחריות השונים על הרשות למחלקות נפרדות ומוחדרות, כדי לאפשר ניסוי ויצירת שיטות אופטימיזציה, אימון, וחישוב שוננות, לשפר את הקרויאות של הקוד, ותת יכולת טוביה יותר יותר לארור ולטפל בתקלות למידה ברשת.

יש הרבה גישות לחלוקת הרשות, בהתאם לחלוקת שאני בחרתי:

מחלקה ה- Network:

בעלת שני תפקידים עיקריים:

הראשון הוא לאחסן במקום מסודר את כל האובייקטים של שכבות הרשות, את פונקציית העלות שנבחרה (פונקציות העלות השונות נמצאות במחלקה `py.loss`), ואת סוג ה-Optimizer שנבחר. הרעיון הוא שרק המחלקה הזו תוכל לבצע שינויים בשכבות הרשות.

התפקיד השני הוא מימוש השיטות הבסיסיות של הרשות:
השיטה **foward** של הרשות תבצע מחזור העברה שלם של הרשות, כלומר בהינתן set ייחד, השיטה תבצע foward לכל שכבה ושכבה עד לשכבת הפלט.
באופן דומה השיטה **backward** תבצע מחזור העברה אחורה לרשות.

השיטה **compute loss** תחשב את העלות בעזרת פונקציית העלות שנבחרה.

והשיטה **Optimize** שבעזרתה יותאמו ערכי המשקלות וההטיות של כלל הרשות בהתאם לשיטת השיטה `set` שנבחרה.

מחלקה ה- Train:

בעלת 2 שיטות מורכזיות:

השיטה **predict** שתבצע העברה קדימה בעזרת השיטה של `network` `data` אמיתית אותו נראה לבדוק לאחר האימון.
השיטה **train** המרכזית שתנהל את הליך הלמידה של הרשות. השיטה מקבלת כל המשתנים שרלוונטיים לאימון כגון: `data`, `true`, `learning rate`, `epoches`, `training data`.

במהלך היריצה השיטה “תכנית” בכל פעם שט אימון בודד מתוך ה-`data` לרשות, ובעזרת השיטות של `network` תבצע לה `forward-pass`, תחשב את העלות, תבצע `backward-pass`, ולבסוף `Optimize`. כולל שיטות של `network`.

השיטה תחזיר את השגיאה של הסט אימון האחרון.

יש הרבה דרכים לבצע את רוטינת האימון, אני הצגתי את הבסיסית ביותר אך ניתן להחליף בכלות לרוטיניות אחרות כמו `mini-batch`.

מחלקה ה- Optimize:

כפי שצייתי בקוד שלי האופטימיזציה של המשקלות בהתאם לגרדיינט מתבצעת ברמת השכבה, אך ברשותות מתקדמות יש צורך בלקראן למחלקה נפרדת שתבצע את עדכון המשקלות.

הקובץ Main:

הקובץ `main` הוא בעצם הקובץ הפעלה של הרשות, שם ניתן ליצור את מערך השכבות, לבחור פונקציות אקטיבציה, ליצר אובייקט `network`, ליבא את `data` ולהפעיל את שיטות האימון וה-`predict`.

*המחלקה `visual` יוצרת גרף החלטה לרשות ולא נכתבת על ידי.

הרצה לדוגמה:

נשתמש ברשת כדי לאמן מודל XOR:

```
1 #configure network's structure, choose {func}
2 layers = [
3     Dense(2, 3),
4     Activation(tanh, tanh_prime),
5     Dense(3, 1),
6     Activation(tanh, tanh_prime)
7 ]
8 net = Network(layers, mse, mse_prime) #create network object, choose {loss}
```

নיצור רשימה של אובייקטים מסוג Layer ונשתמש בה כדי לאותל את הרשת: במקורה שלנו לרשת 3 שכבות, פונקציית האקטיבציה היא tanh, ופונקציית חישוב העלות היא MSE.

```
1 #training
2 X = np.reshape([[0, 0], [0, 1], [1, 0], [1, 1]], (-1, 2, 1))
3 Y = np.reshape([[0], [1], [1], [0]], (-1, 1, 1))
4 train(network=net, train_data=X, true_data=Y, epochs=1000, learning_rate=0.1)
```

ניבא את ה-training data, נשמרו אותו כמטריצה כאשר כל עמודה .data set בה היא

בנוסף ניבא את ה-data true, במקורה שלנו כערבים בוחדים (או קטור 1×1) כאשר כל $[i][j]$ הוא הערך האמיתי של הסט $[i][j]$.

לאחר מכן נקרה לפונקציית ה-train עם הרשת שלנו וכל הערכים הרלוונטיים.

```
1 # testing
2 T = np.reshape([1, 0], (2, 1))
3 pred = predict(network=net, test_data=T) #predict test_data
```

נשמר את הערך אותו נרצה להכניס לרשת לבדיקה, גם הוא בתור וקטור עמודה.

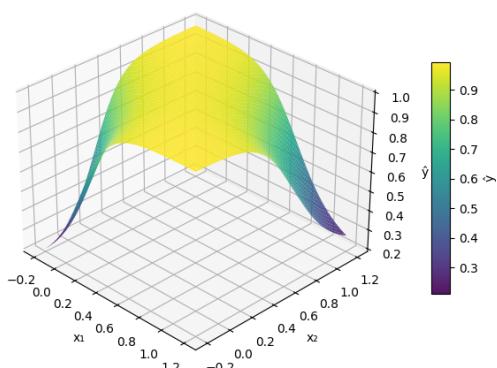
נפעיל את השיטה predict שתעשה לו העברת הקדימה ותחזר את הערך שהרשת חצתה.

ניתן לראות שעבור $[1,0]$, זה הפלט של הרשת:

המערכת חצתה בהצלחה את המספר 1, עם אחוז ביטחון של 99.2%, ופונקציית השגיאה באיטרציה الأخيرة ירדה ל-0.0002 - נתונים מעולים.

```
Run main x
Run | Stop | :
/usr/bin/python3 /Users/tomoplll/Documents/JetBrains/XOR/main.py
Prediction: 1
Confidence: 99.2%
Error: 0.0002
Process finished with exit code 0
```

Decision Surface



ניתן גם לראות את התפלגות הנתונים של ההרצה:

שער הקוד:

מחלקה (כרגע יותר כמו פונקציית ה-Train):

```
1 def train(network, train_data, true_data, epochs=1000, learning_rate=0.1):
2     """
3         Train the network by performing forward and backward passes for each epoch.
4         Updates layer parameters using training data to reduce prediction error.
5
6         :param network: Network object
7         :param train_data: Inputs for the first layer's values
8         :param true_data: Corresponding expected outputs
9         :param epochs: Number of training iterations
10        :param learning_rate: Step size for updating parameters
11        :return: Error of the last epoch
12    """
13    error = 0
14    for epoch in range(epochs):
15        error = 0
16        for training_set, y_true in zip(train_data, true_data):
17            y_pred = network.forward(training_set) # forward-pass
18            error += network.compute_loss(y_true, y_pred) # compute error
19            network.backward(y_true, y_pred, learning_rate) # backward-pass
20    return error / len(train_data)
21
22 def predict(network, test_data):
23     """Perform forward pass and return the network's output for the given input."""
24     return network.forward(test_data).item()
```

מחלקה ה-Network:

```
1 from loss import mse, mse_prime
2
3 class Network:
4     def __init__(self, layers, loss=mse, loss_prime=mse_prime):
5         """
6             Initialize the network with the chosen structure and loss function
7             :param layers: Layers list of the network
8             :param loss: Chosen loss function from loss.py
9             :param loss_prime: Corresponding prime function
10        """
11        self.layers = layers
12        self.loss = loss
13        self.loss_prime = loss_prime
14
15    def forward(self, input_data):
16        """Perform forward pass through all the layers given single data-set"""
17        x = input_data
18        for layer in self.layers:
19            x = layer.forward(x)
20        return x
21
22    def backward(self, y_true, y_pred, learning_rate):
23        """Perform backpropagation through all layers and update the weights"""
24        gradient = self.loss_prime(y_true, y_pred)
25        for layer in reversed(self.layers):
26            gradient = layer.backward(gradient, learning_rate)
27
28    def compute_loss(self, y_true, y_pred):
29        """Compute the error using {loss} func"""
30        return self.loss(y_true, y_pred)
```

מחלקה Layer

```
1 class Layer:
2     def __init__(self):
3         self.input = None
4         self.output = None
5
6     def forward(self, layer_input):
7         pass
8
9     def backward(self, output_gradient, learning_rate):
10        pass
```

מחלקה Activation

```
1 from layer import Layer
2 from activationFn import sigmoid, sigmoid_prime
3 import numpy as np
4
5 class Activation(Layer):
6     def __init__(self, func=sigmoid, func_prime=sigmoid_prime):
7         """
8             Initialize the activation layer with {func} as the activation function
9             to include non-linearity in the model.
10            :param func: Chosen activation function from activationFn.py
11            :param func_prime: Corresponding prime function
12            """
13        super().__init__()
14        self.activation_fn = func
15        self.activation_fn_prime = func_prime
16
17    def forward(self, layer_input):
18        self.input = layer_input
19        return self.activation_fn(self.input)
20
21    def backward(self, output_gradient, learning_rate):
22        return output_gradient * self.activation_fn_prime(self.input)
```

מחלקה Dense

```
1 from layer import Layer
2 import numpy as np
3
4 class Dense(Layer):
5     def __init__(self, input_size, output_size):
6         """
7             Initialize the layer with random weights and biases, sets inputs to None
8             :param input_size: Number of neurons in the layer (input)
9             :param output_size: Number of neurons in the next layer (output)
10            """
11        super().__init__()
12        self.weights = np.random.randn(output_size, input_size)
13        self.bias = np.random.randn(output_size, 1)
14
15    def forward(self, layer_input):
16        self.input = layer_input
17        return np.dot(self.weights, self.input) + self.bias
18
19    def backward(self, output_gradient, learning_rate):
20        weight_gradient = np.dot(output_gradient, self.input.T)
21        input_gradient = np.dot(self.weights.T, output_gradient)
22
23        self.weights -= learning_rate * weight_gradient
24        self.bias -= learning_rate * output_gradient
25        return input_gradient
```

קובץ פונקציות ה-Loss

```
1 import numpy as np
2
3 def mse(y_true, y_pred):
4     return np.mean(np.power(y_true - y_pred, 2))
5
6 def mse_prime(y_true, y_pred):
7     return 2 * (y_pred - y_true) / np.size(y_true)
8
9 def binary_cross(y_true, y_pred):
10    p = np.clip(y_pred, 1e-15, 1 - 1e-15)
11    return -np.mean(y_true * np.log(p) + (1 - y_true) * np.log(1 - p))
12
13 def binary_cross_prime(y_true, y_pred): # can only be used if final activation is sigmoid
14    p = np.clip(y_pred, 1e-15, 1 - 1e-15)
15    return (-y_true / p + (1 - y_true) / (1 - p)) / y_true.size
16
17 def categorical_cross(y_true, y_pred):
18    p = np.clip(y_pred, 1e-15, 1 - 1e-15)
19    return -np.mean(np.sum(y_true * np.log(p), axis=1))
20
21 def categorical_cross_prime(y_true, y_pred):
22    p = np.clip(y_pred, 1e-15, 1 - 1e-15)
23    return - (y_true / p) / y_true.shape[0]
```

קובץ פונקציות האקטיבציה:

```
1 import numpy as np
2
3 def tanh(x):
4     return np.tanh(x)
5
6 def tanh_prime(x):
7     return 1 - np.tanh(x)**2
8
9 def sigmoid(x):
10    return 1 / (1 + np.exp(-x))
11
12 def sigmoid_prime(x):
13    s = sigmoid(x)
14    return s * (1 - s)
15
16 def relu(x):
17    return np.maximum(0, x)
18
19 def relu_prime(x):
20    return (x > 0).astype(float)
```

קובץ הרצה :main

```
1 import numpy as np
2 from dense import Dense
3 from activation import Activation
4 from network import Network
5 from train import train, predict
6 from activationFn import *
7 from loss import *
8 from visual import plot_surface
9
10 def main():
11     """
12         Build, train and evaluate an MLP.
13         Variables:
14             Network structure
15                 Activation function from activationFn.py (default: sigmoid)
16                 Loss function from loss.py (default: MSE)
17                 Epochs and learning_rate (default: 1000, 0.1)
18     """
19
20     # configure network's structure, choose {func}
21     layers = [
22         Dense(2, 3)
23         , Activation(tanh, tanh_prime)
24         , Dense(3, 1)
25         , Activation(tanh, tanh_prime)
26     ]
27
28     net = Network(layers, mse, mse_prime) # create network object, choose {loss}
29     # training
30     X = np.reshape([[0, 0], [0, 1], [1, 0], [1, 1]], (-1, 2, 1))
31     Y = np.reshape([[0], [1], [1], [0]], (-1, 1, 1))
32     error = train(network=net, train_data=X, true_data=Y, epochs=1000, learning_rate=0.1)
33
34     # testing
35
36     T = np.reshape([1, 0], (2, 1))
37     pred = predict(network=net, test_data=T) # predict test_data
38
39     # confidence computation
40     pred = (pred+1)/2 # this line only needed when using tanh
41     result = 1 if pred >= 0.5 else 0
42     confidence = max(pred, 1 - pred) * 100
43
44     #visualize
45     plot_surface(net)
46
47     # print results
48     print(f"Prediction: {result}")
49     print(f"Confidence: {confidence:.1f}%")
50     print(f"Error: {error:.4f}")
51
52
53 if __name__ == "__main__":
54     main()
```