

# למידה عمוקה

מדריך ללמידה



### **תנאי שימוש בקובץ הדיגיטלי:**

1. הקובץ הוא לשימוש **האישי** בלבד. פרטים מזהים שלך מוטבעים בקובץ בגלוי ובצורה סטטיסטית.
2. השימוש בקובץ הוא אך ורק למטרות לימוד, עיון ומחקר אישי.
3. העתקה או שימוש בתכנים נבחרים מותרת בהיקף העומד בכלל השימוש ההוגן, המפורטים בסעיף 19 לחוק זכויות יוצרים 2007. במקרה של שימוש כאמור חלה חובה לציין את מקור הפרסום.
4. הנר רשאי/ת להדפיס דפים מחומר הלימוד לצורכי לימוד, מחקר ועיון אישיים. אין להפיץ או למכור תדים כleshם מתוך חומר הלימוד.



# למידה عمוקה

## מדריך למידה

מחודשת פנימית

לא להפצה ולא למכירה

מק"ט 0000-22961

**דף צוות**

**כתובת:** עידן אלתר

**אחראי אקדמי:** אורן ברכאן

**עריכה לשונית:** דפי בר אילן

**הדפסה דיגיטלית מעודכנת – ינואר 2024**

© תשפ"ד – 2024. כל הזכויות שמורות לאוניברסיטה הפתוחה.

בית ההוצאה לאור של האוניברסיטה הפתוחה, הקרייה ע"ש דורותי דה ROTHSCHILD, דרך האוניברסיטה 1, ת"ד 808, רעננה 4353701.  
The Open University of Israel, The Dorothy de Rothschild Campus, 1 University Road, P.O.Box 808, Raanana 4353701.  
Printed in Israel.

אין לשכפל, להעתיק, לצלם, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או בכל אמצעי אלקטרוני, אופטי, מכני או אחר כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת ובכתב ממדור זכויות יוצרים של האוניברסיטה הפתוחה.

אין לשכפל, להעתיק, לצלם, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או בכל אמצעי אלקטרוני, אופטי, מכני או אחר כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת ובכתב ממדור זכויות יוצרים של האוניברסיטה הפתוחה.

## **ראשית דבר**

הקורס למידה عمוקה נלמד במתכונת "רצפים", ובאתר הקורס נמצא כל חומר הלימוד הדרושים. בקובץ זה מאוגדים פרקי מדריך הלמידה, והם רק חלק מהחומר הנלמד. קובץ זה מוגש עבורהם, הסטודנטים והסטודנטיות, ככלי עזר – למשל כדי לחפש מונח כזה או אחר במדריך כולם. עם זאת, זכרו שתהליך הלמידה נעשה באתר הקורס, שבו משולב מדריך הלמידה, כל פרק בהקשרו המתאים.

א

File #0005850 belongs to Tiran Spierer- do not distribute

# תוכן העניינים

א

ראשית דבר

## חידה 1: מבוא טכני

7	אתחול טנזור
9	אופרטורים טנזוריים
13	גישה לנוטונים השמורים בטנזור
16	חישוכן בזיכרון
19	פעולות חשבון על טנזוריים השונים זה מזה בגודלם
23	גירה אוטומטית

## חידה 2: צעדים ראשוניים

27	דגימת נתונים סינטטיים והגדרת מודל הסיווג
32	פונקציית המחיר ואלגוריתם האימון של הנירון
36	הAIMON הנירון – מימוש מהיסוד
42	הAIMON הנירון – מימוש עם PyTorch
45	אוסף הנתונים
50	הגדרת המודל ופונקציית המחיר
54	אלגוריתם האימון – מورد הגרדיאנט האקראי
56	הAIMON הרשות
59	רשתות עמוקות

## חידה 3: אופטימיזציה

63	התפשטות לאחר
70	קצב הלמידה
74	זמן קצב הלמידה
78	שיפורים לאלגוריתם מورد הגרדיאנט
84	שכבות נורמליזציה
91	אתחול הפרמטרים

## חידה 4: רגולרייזציה

94	רשתות נירוניים לרגרסיה
98	התאמת יתר
102	דעיכת המשקלים
108	הAIMON רשות בעזרת מאיצ' גרפי
111	שכבות Dropout

## חידה 5: רשתות קונבולוציה (CNN)

114	מוטיבציה לשימוש ברשתות קונבולוציה
117	פעולות הקונבולוציה עם גרעין
122	מבנה שכבת קונבולוציה ברשתות נירוניים

126	מבנה שכבת קונבולוציה : ריבוי ערווצים
131	רכיבים נוספים של שכבות קונבולוציה
138	רשתות קונבולוציה מודרניות
143	רשתות שיוריות
 <b>חידה 6 : רשתות נשנות (RNN)</b>	
149	שימוש מקדים של נתוני טקסט
154	סיווג נתוני טקסט
157	רשתות נשנות
163	הטפסות לאחרך דרך הזמן
 <b>חידה 7 : ארכיטקטורות מקודד-מפרש</b>	
167	מקודד עצמי
170	שימושים נוספים של מקודדים עצמיים
175	קונבולוציה משוחלפות ו שימושה
178	תרגום מכונה באמצעות רשת מקודד-מפרש נוספת
185	שכבות תשומת לב ו שימושה במפרש הנשנה

# ICHIDAH 1: מבוא לתוכנותי

## ATCHOL TANZOR

כדי לעבוד עם טנзорים (tensor), ראשית علينا לאתחל אותם בזיכרון, ולשם כך קיימים בנאים (קונסטראקטוריים) רבים.

הדרך הבסיסית ביותר לאתחל טנזור היא באמצעות העברת מערך פיתוני או ndarray של כפרמטר לבניי tensor, כפי שניתן לראות בהרצת קטע הקוד שלחן במחברת ג'ופיטר:

```
import torch
import numpy as np
x = torch.tensor([[1, 2], [3, 4]])
```

לאחר הריצה, נדפיס את המשתנה `x` וכמה מאפיינים שלו כך:

```
print(x, x.shape, x.dtype, sep='\\n')
tensor([1, 2,
        [3, 4]])
torch.Size([2, 2])
torch.int64
```

פלט:

מפלט זה ניתן ללמוד כי `x` הוא אובייקט tensor המכיל מטריצת נתונים מסדר 2X2, וכי כל אחד מאיבריה הוא משתנה מסוג טיפוס .int64.

אם נשנה מעט את הקלט של הבניי, למשל את האיבר הראשון נשנה לטיפוס ממשי, נקבל את התוצאה זו:

```
x = torch.tensor([1.0, 2, 3, 4])
print(x, x.shape, x.dtype, sep='\\n')
tensor([1., 2., 3., 4.])
torch.Size([4])
torch.float32
```

פלט:

ומכך אנו למדים כי כל איברי המערך הומרו לטיפוס ממשי, כולל הטיפוס של כל האיברים בטנזור חייב להיות זהה. מבנה הנתונים תומך בטיפוסי הנתונים המספריים הסטנדרטיים של מספרים שלמים, ממשיים, מרכבים וכן בטיפוס בוליани.



לרוב נרצה לאתחול טזוריים לפי חוקיות מסוימת, למשל בתור מטריצת היחידה או באמצעות דגימות ערכאים אקראיים מהתפלגות נתונה, ולמטרות אלו קיימים בנאים אחרים. דוגמאות אחדות לכך מופיעות בקטע הקוד שלහן:

```
x = torch.arange(9)
y = torch.eye(3,dtype=torch.bool)
z = torch.randn(size=[2, 3, 3])
print(x, y, z, sep='\n')

tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
tensor([[ True, False, False],
        [False,  True, False],
        [False, False,  True]])
tensor([[[[-0.8441,  0.5493,  0.2198],
          [-1.4702, -1.2846,  0.1339],
          [ 1.3861,  1.7858,  0.5074]],

         [[-1.0042, -0.9809,  0.4707],
          [-0.1001, -2.6261,  0.8152],
          [-1.1585,  0.8810, -0.2189]]])
```

**פלט:**

שימוש לב Ci הטנזור z – תלת-ממדי – הוא מורכב משתי מטריצות בגודל 3X3. למעשה, אין מגבלה על מספר הממדים של אובייקט tensor, ולוთים קרובות הברירה הנוחה ביותר לשימושינו תהיה טזוריים רב-יממדים. ישנו בנאים רבים נוספים, ואין טעם בסקירה ממצה של כלם – כל שיש לדעת הוא שלרוב הצרכים אשר יעלו בעת מימוש רשתות נוירונים קיימים בנאי מתאים, ושיש למצוא אותו בתיעוד של הספרייה PyTorch.

## שאלות לתרגול

1. צרו רשימה סטנדרטית של פירטן המכיל משהנים מסוגים שונים (float, int, char וכו') ונסו להמיר להטנזור PyTorch האם הצלחתם? אם לא, האם תוכלו להסביר מדוע?
2. צרו טנзор 4-ממדי של ביתים אקראיים לחלוין (0 או 1 בהסתברות שווה). היוזרו בקישור לתיעוד של PyTorch המופיע באתר הקורס.

## אופרטורים טנзорיים

אחד היכולות של שימוש בטנзорים הוא האפשרות לבצע פעולות איבר-איבר על כל הטנзор באופן חסכוני ומהיר, לדוגמה :

```
x = torch.tensor([1, 2, 3, 4])
y = torch.tensor([5, 6, 7, 8])
print(x + y)

tensor([ 6,  8, 10, 12])
```

פלט:

ניתן לראות כי הפעולה  $y + x$  הניבת טנזור אשר כל איבר שלו הוא סכום האיברים בעלי אותו אינדקס בטנзорים המקוריים. על כך נאמר כי האופרטור `"+"` עבר הרמה לפועל איבר-איבר על טנзорים. רוב הפעולות המתמטיות הורמו בצורה דומה, ראו למשל (זכרו כי `*` היא פעולה החזקה) :

```
print(x - y, x * y, x / y, x**y, sep='\n')
tensor([-4, -4, -4, -4])
tensor([ 5, 12, 21, 32])
tensor([0.2000, 0.3333, 0.4286, 0.5000])
tensor([ 1, 64, 2187, 65536])
```

פלט:

מלבד היתרונו של תמציתיות הכתיבה והקריאות של השימוש באופרטורים מורמים, לרוב השימוש בהם מהיר בסדרי גודל לעומת שימוש נאיבי של הפעולה בעזרות לולאות. ראו למשל את ההשוואה שלහן, המשמשת בפקודת `timeit %timeit`magic. לאחר יצירת שני טנзорים חד-ממדים בעלי 10,000 איברים,

```
x = torch.ones(size=(10**5,))
y = torch.randint(low=0, high=10, size=(10**5,))
```

נמדד את הזמן שאורכת פעולה הכפל איבר-איבר בשתי דרכים שונות :

1. פעולה הכפל איבר-איבר הטנзорית :

```
%%timeit
x*y
```

ומן טיפוסי של הרצת תא זה הוא כ-200 מיקרו-שניות.

2. פעולה הכפל בעזרת לולאה :

```
%%timeit
for i in range(x.numel()):
    x[i]*y[i]
```

זמן טיפוסי של הרצת תא זה הוא כ-600 מילישניות, איטי פי 3,000 מההרצת הקודמת!

ההבדל בזמן הריצה אינו יהודי כתוצאה הפעלת המכפלת, ויש פעולות מורכבות אחרות אשר הופיעו בין המימושים השונים של חישוב. הסיבות לכך חמורות מתחום הדין הנוכחי, אך עליינו לזכור שלמען מהירות זמן הריצה של הקוד, כדאי להשתמש בפעולות טזוריות ככל האפשר. למשל, לא רק הפעולות האריתמטיות הבסיסיות הורמו לטזוריים, אלא גם אופרטורים בוליאניים וכן אופרטורים המוכרים לנו מאלgebra לינארית, כגון כפל מטריצות וחישוב מכפלות פנימיות וחיצוניות. למשל:

```
x = torch.tensor([1, 2, 3, 4])
y = torch.tensor([5, 6, 7, 8])
print(x@y, x==1, sep='\n')

tensor(70)
tensor([ True, False, False, False])
```

**פלט:**

נשים לב שכאשר שני הטזוריים הם חד-ממדיים, האופרטור `@` מחשב את המכפלת הפנימית שלהם, אך אם נשנה אותן לוקטור عمودה ווקטור שורה דו-ממדיים, יוכל לקבל את תוצאה כפל המטריצות באמצעות שימוש באותו אופרטור:

```
x = x.reshape(1, 4)
y = y.reshape(4, 1)
print(x, y, x@y, y@x, sep='\n')

tensor([[1, 2, 3, 4]])
tensor([[5],
       [6],
       [7],
       [8]])
tensor([[70]])
tensor([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28],
       [ 8, 16, 24, 32]])
```

**פלט:**

סוג שימושי נוסף של אופרטורים הפעילים על טזוריים הוא מתודות הפקחה – מתודות של הטזור המחזירות סקלר, לרוב מזרדי סיכום, כגון ערכי מינימום, מקסימום, ממוצע או נורמה. דוגמאות ספורות למתודות אלו ניתן לראות כאן:

```
x = torch.ones(size=(3,))
y = torch.ones(size=(3, 3))
z = torch.ones(size=(3, 3, 3))
print(x.norm(), y.min(), z.sum(), sep='\n')

tensor(1.7321)
tensor(1.)
tensor(27.)
```

**פלט:**

שימוש לב שמותודות ההפחתה תמיד מוחזירות סקלר, ללא תלות בממדי הטנזור המקורי. עם זאת, ניתן להעביר את אחד הממדים כפרמטר, ההפחתה תתבצע רק לאורך ממץ זה, ויווצר טנзор הקטן בממד אחד מטנזור הקלט. ראו למשל:

```
x = torch.arange(24).reshape(2, 3, 4)
print(x, x.size())

y = x.sum()
print(y, y.size())

z = x.sum(axis=0)
print(z, z.size())

w = x.sum(axis=1)
print(w, w.size())

tensor([[ [ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11]],

         [[12, 13, 14, 15],
          [16, 17, 18, 19],
          [20, 21, 22, 23]]]) torch.Size([2, 3, 4])
tensor(276) torch.Size([])
tensor([[12, 14, 16, 18],
        [20, 22, 24, 26],
        [28, 30, 32, 34]]) torch.Size([3, 4])
tensor([[12, 15, 18, 21],
        [48, 51, 54, 57]]) torch.Size([2, 4])
```

פלט:

בדוגמה זו הסכום ב-`z` התבצע לאורך המטריצות (ממד 0) והתקבלו מטריצה אחת מאותו ממד כמו השתיים בטנזור המקורי, ואילו ב-`w` הסכום התבצע על עמודות המטריצות (ממד 1 של `x`) המוקריות, וכל שורה במטריצת הפלט מייצגת את סכום העמודות של אחת מהן.

בדומה לקונסטרוקטורים, אין צורך בעל פה את כל האופרטורים שניתנו להפעיל על טנзорים, אך לפני מימוש נאיבי של פעולה מתמטית או לוגית סטנדרטית בעוררת לולאות, חונייני לבדוק בתיעוד האם קיימים עבורה אופרטור מובנה יעיל. לרוב התשובה תהיה חיובית ותחסוך זמן תכנון וזמן ריצה.

## שאלות לתרגול

- צרו טנזור תלת-מדי בגודל `2X2X100` אשר מכיל מספרים אקראיים מהתפלגות אחידה סטנדרטית.
- חשבו את הערכים העצמיים של כל המטריצות מסדר `2X2` המתקבלות עבור כל ערך של הממד הראשון. מה תוכלו להגיד על הערך המדומה של הערכים העצמיים של המטריצות? האם תוכל להסביר תופעה זו?
- חשבו את נורמת פרובניאס של כל אחת מהמטריצות.
- חשבו את הנורמה האוקלידית של הטנזור המקורי, כאשר כל ערכיו מובאים בחשבון.



- ד. חשבו את נורמת אינסוף של שתי המטריצות מוגדל  $2 \times 100$  המתקבלות עבור כל אחד מערכי הממד האחרון.
- הערה: היערו בתיעוד הספרייה `torch.linalg` למציאת הפונקציות המתאימות, ונסו לבצע כל חישוב בעזרת שורת קוד אחת.
2. השוו את זמן הריצה של פעולות חילוק והאקספוננט אשר הורמו לפעול על טזוריים, בזמן הריצה המקורי בעזרת LOLA. ערכו השוואה זו בעזרת הפקודה `timeit %timeit`. האם תוכלו להסביר את המקביל בעזרת LOLA. האם השוואת הפעולות מושגתה?

## גישה לנוטונים השמורים בטנזור

לנתונים השמורים בטנזור ניתן לגשת כמו שниיגשים לכל מערך פיבוטני רגיל, ובדומה לכך ניתן גם לכתוב נתונים לתוך הטנזור. ראו לדוגמה:

```
x = torch.arange(10)
print(x)
```

**פלט:**

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

זכרו שהאינדקס של האיבר הראשון במערך פיבוטני הוא 0, ושניתן לגשת אל האיבר האחרון במערך בעזרה האינדקס -1.

```
x[-1] = 100
print(x[2], x[-1], sep='\n')
```

**פלט:**

```
tensor(2)
tensor(100)
```

בדומה לבניה הנתונים של הספרייה NumPy, ישנן דרכים מתקדמות לגשת לאיברים ספציפיים בטנזור, היכולות אינדקסים רבימדיים. לצורך הדוגמה נשנה את צורת הטנזור הקודם:

```
x = torch.arange(10).reshape(2, 5)
print(x)
```

**פלט:**

```
tensor([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

בעת x הוא מטריצה, ולאיבריה ניתן לגשת בעזרה אינדקס שורה ועמודה:

```
print(x[1, 2], x[0, -1], sep='\n')
```

**פלט:**

```
tensor(7)
tensor(4)
```

שיטה זו עובדת עבור טנזור מכל גודל, ובעור כל אחד מממדיו, ראו כאן לדוגמה:

```
y = torch.zeros(size=(2, 3, 4))
y[1, 0, 0] = -2
y[-1, -1, -1] = 1
print(y)
```

**פלט:**

```
tensor([[[ 0.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  0.]],

        [[-2.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  1.]]])
```

בדומה לכך, ניתן לקבל חתך (slice) מהמטריצה, באמצעות מתן טווח ערכים באחד האינדקסים. עבור צ הNIL, בדוגמה שלහן נציב 100 בכל האיברים אשר אינדקס הממד הראשון שלהם הוא 1, אינדקס הממד השני הוא 1 עד 2 (לא כולל) ואינדקס הממד השלישי הוא כל המספרים בין אפס (כולל) ל-4 (לא כולל), בקפיצות של 2 :

```
y[1,1:2,0:4:2] = 100
print(y)
```

**פלט:**

```
tensor([[[ 0.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  0.]],

        [[-2.,  0.,  0.,  0.],
         [100.,  0.,  100.,  0.],
         [ 0.,  0.,  0.,  1.]]])
```

ובן שבדרכ זו אפשר גם לאחזר גם את המידע השמור ב-*y* ולא רק ל כתוב לתוכו.

דרך נוספת ושימושית ביותר לגשת לאיברי הטזוזר היא באמצעות **אינדקסים בוליאניים** (masks), באמצעותם ניתן לחוץ מהתזוזר איברים מסוימים תנאי מסויים. לדוגמה :

```
print(y[y>0])
```

**פלט:**

```
tensor([100., 100., 1.])
```

למעשה, הפעולה האלגנטית הניל מתרבצעת בשני שלבים : ראשית מחושב הטזוזר בוליאני בעל ממד זהה ל-*y*. אחרי כן, הטזוזר הבוליאני מועבר כאינדקס ל-*y* ומוחזרים רק הערכים אשר באותו מקום יש ערךאמת בטזוזר הבוליאני. ניתן להמחיש זאת באמצעות פיצול החישוב לשני שלבים :

```

z = y>0
print(z,y[z],sep='\n')
פלט:

tensor([[ [False, False, False, False],
          [False, False, False, False],
          [False, False, False, False]],

         [[False, False, False, False],
          [ True, False,  True, False],
          [False, False, False,  True]]])
tensor([100., 100.,    1.])

```

שימוש לב שב식ת זו הפלט המתתקבל הוא תמיד חד-ממדי.

טזoor האינדקס הבוליאני יכול להיות גם מממד נמוֹך יותר. במקרה זה הפלט יהיה רב-ממדי:

```

print(y[[False,True]])
פלט:

tensor([[[-2.,    0.,    0.,    0.],
        [100.,   0., 100.,    0.],
        [  0.,    0.,    0.,    1.]]])

```

בדוגמה זו ניתן לראות שהאינדקס הבוליאני התייחס לממד הראשון של `y` והחזיר למעשה את כל המידע בעל אינדקס 1 של הממד הראשון: `[:, :, 1]`. זהו הכלל במקרים שבהאינדקס מממד נמוֹך יותר: הוא מתייחס לממדים הראשונים, ואינו משנה על שאר הממדים.

## שאלות לתרגול

1. דגמו טזoor `X` בגודל  $10 \times 10$  של משתנים מקריים נורמליים אחידים, וצרו טזoor `Y` בעל אותם ממדים, המכיל רק ערכי `Inf`. השתמשו בשנייהם להחזיר טזoor מאותו גודל אשר מכיל את האיברים החשובים של `X`, ובמקום האיברים השליליים – `Inf`. רמז: כדאי להשתמש בפונקציה `.torch.where()`.
2. שנו את הטזoor `X` מהשאלה הקודמת לטזoor תלת-ממדי בגודל  $2 \times 5 \times 10$ , וצרו חתך שלו המכיל רק את האיברים אשר שלושת האינדקסים שלהם איזוגיים.

## חיסכון בזיכרון

במהלך הקורס נשתמש בתנוזרים גדולים מאוד, ועל כן פעולות של הקצאת זיכרון וכתיבה אליו ייצרו צוואר בזיכרון שננסה להימנע ממנו ככל האפשר. לרוב ננקוט שתי גישות שונות:

1. כאשר אפשר, למשל כאשר רוצים לשנות את ערכי המשתנים, ניצור משתנים חדשים בעלי **מבט (view)** אל הנתונים של משתנה אחר, **במקום להעתיקם מחדש**. משתנים אלו יקבלו גישה אל המיקום בו זיכרונו שבו שמורים הנתונים של המשתנה המקורי.
2. ככל האפשר ננסה **לבצע פעולות במקום (in-place)** – תוצאות חישוב על טנזור נתון ישמרו באותו מקום בזיכרון שבו שמורים הנתונים המקוריים של הטנзор.

יש בספרייה PyTorch כלים שייעזרו לנו לממש גישות אלו, ולמעשה במקרים מסוימים הן מתחממות אוטומטית. למשל, כאשר לוקחים חתך של טנзор ומציבים אותו בטנזור חדש, מתקבלמבט, שכן אין צורך להעתיק את הנתונים:

```
x = torch.arange(2*4*5).reshape(shape=(2, 4, 5))
y = x[:, 0::2, 2::5]
print(x, y, sep='\\n')

tensor([[[ 0,  1,  2,  3,  4],
         [ 5,  6,  7,  8,  9],
         [10, 11, 12, 13, 14],
         [15, 16, 17, 18, 19]],

        [[20, 21, 22, 23, 24],
         [25, 26, 27, 28, 29],
         [30, 31, 32, 33, 34],
         [35, 36, 37, 38, 39]]])
tensor([[ [ 2,  3,  4],
          [12, 13, 14]],

         [[22, 23, 24],
          [32, 33, 34]]])
```

**פלט:**

בדוגמה זו חתכו את x כדי לקבל את האיברים אשר אינדקס הממד השני שלהם (שורות המטריצה) הוא 0 או 2, ואינדקס הממד השלישי (עמודות המטריצה) הוא 3, 2 או 4. ודאו שאתם מבינים זאת, ואם לא – חזרו על נושא האינדקסים לפני שתמשיכו. את החתך הצבנו במשתנה y אך עליינו להיות מודעים לכך שגם y חולקים את הנתונים זה עם זה, ושלפיכך שינוי אחד מהם יוביל לשינוי בשני:

```

y[-1] = 100
print(y, x, sep='\n')

tensor([[[ 2,   3,   4],
         [ 12,  13,  14]],

        [[100, 100, 100],
         [100, 100, 100]]])
tensor([[[ 0,   1,   2,   3,   4],
         [ 5,   6,   7,   8,   9],
         [10,  11,  12,  13,  14],
         [15,  16,  17,  18,  19]],

        [[ 20,  21, 100, 100, 100],
         [ 25,  26,  27,  28,  29],
         [ 30,  31, 100, 100, 100],
         [ 35,  36,  37,  38,  39]]])

```

פלט:

מצב זה לרוב רצוי, שכן כבירית המודול אנו מעוניינים בחישוכו בזיכרונו. עם זאת, הוא יכול להוביל לשגיאות לא צפויות. כאשר נרצה בכל זאת להעתיק את הנתונים למקום חדש תוך כדי ניתוק הקשר בין המשתנים, נעשה זאת באמצעות המתודה `( ) clone` אשר יכולה לפעול על כל טנזור. בהרצאת שורת הקוד שלහלן קיבל בטנзор `z` עותק חדש של הנתונים, ושינוי ערכיו לא ישפיע על ערכי `x` או `y`.

```
z = x[:, 0::2, 2:5].clone()
```

שימוש לבכך שכאש משתמשים באינדקס בוליאני, התוצאה המתקבלת היא תמיד עותק. ראו בדוגמה המצורפת כיצד שינוי ב-`z` אינו משפיע על הטנзор `x`, אף שהראשון נחתך מהשני.

```

x = torch.arange(5)
index = (x>=2) & (x<4)
y = x[index]
y[-1] = 0
print(x, index, y, sep='\n')

tensor([0, 1, 2, 3, 4])
tensor([False, False, True, True, False])
tensor([2, 0])

```

פלט:

כעת נעבור לדון בגישה השנייה לחישוכו בזיכרונו, ביצוע פעולות החישובן במקום, ודרך זאת מוחזר מקום בזיכרונו שכבר הוקצה בעבר. בדוגמה שלහלן הממחישה את הבעיה, משתמש בפונקציה `( ) id` המחזיר את כתובות הזיכרונו של נתוני הטנзор.

```
x = torch.ones(5)
address = id(x)
x = torch.exp(x)
print(address == id(x))

False
```

**פלט:**

nicer שהפעלת האקספוננט על `x` דרש הקצאה של מקום חדש בזיכרון, ופעולות ההצבה של תוצאה החישוב ב-`x` עצמו היא למעשה שינוי המצביע לנ吐ונים של הטנזור אל המקום החדש בזיכרון. בכך נוצר "בזבוז" של מקום.

הגישה המקובלת להתמודד עם בעיה זו היא לסמוך על האופטימיזציות הקיימות בספרייה PyTorch ולהימנע מפתרונה באופן ישיר, שכן כך עלולה להיווצר התנשאות עם אחת הממערכות היסודיות של הספרייה, הגירה האוטומטית, שנלמד עלייה בהמשך הקורס. למרות זאת, באפשרותנו להכabil את דרך הפעולה ישירות בשתי דרכים שונות. ראשית, אפשר להציב ישירות את תוצאה החישוב לתוך המקום היישן בזיכרון, באמצעות שימוש בחיתוך :

```
address = id(x)
x[:] = torch.exp(x)
print(address == id(x))

True
```

**פלט:**

שנית, אפשר להשתמש בפונקציות של PyTorch המובנות, כדי לבצע את החישוב במקום. ברוב המקרים פונקציות אלו יהיו בעלות שמות זהים לפונקציות המקוריות, עם קו תחתון בסוף השם, למשל הפקודה `(x) torch.exp_` תבצע את פעולה האקספוננט על ערכי הטנзор, ונוסף על כך תציב את הערכים החדשים במקום המקורי בזיכרון, כפי שניתן לראות בדוגמה זו :

```
x = torch.ones(5)
print(x)
torch.exp_(x)
print(x)

tensor([1., 1., 1., 1., 1.])
tensor([2.7183, 2.7183, 2.7183, 2.7183, 2.7183])
```

**פלט:**

## שאלה לתרגול

הפכו את הטנзор `x` הניל לווקטור عمودה וחברו אותו לעצמו, תוך כדי שמירת התוצאה באותו המקום בזיכרון.

## פעולות חישוב על טנзорים שונים זה מזה בגודלים

כאשר עסקנו באופרטורים טנוריים ודנו בכך שהם הורמו לפעול איבר-איבר, הרצנו בין השאר את הדוגמה זו :

```
x = torch.tensor([1, 2, 3, 4])
print(x==1)

tensor([ True, False, False, False])
```

פלט:

על פניו, תוך כדי פירוש נאיבי של פעולה ההשוואה, הפוקודה הייתה צריכה להשוות בין הטנзор  $x$  לבין הסקלר 1 ולהחזיר תשובה שלילית. פונקציונליות זו אינה שימושית במיוחד שכן התשובה להשוואה זו ברורה מלאה, ולכן קודם להשוואה (איבר-איבר), הסקלר 1 עבר **שידור (broadcasting)** : ממדיו הושו לumed הטנзор באמצעות שכפול הערך 1. למעשה, הפעולה שהתבצעה מתחום הקלעים שקופה לשורה הראשונה בקטע הקוד :

```
y = torch.tensor([1]*x.numel())
print(x, y, x==y, sep='\\n')

tensor([1, 2, 3, 4])
tensor([1, 1, 1, 1])
tensor([ True, False, False, False])
```

פלט:

פונקציונליות השידור מרחיקה לכת מעבר לשכפול בסיסי זה, ובאופן כללי כאשר ננסה להפעיל אופרטורים על טנзорים במדדים שונים, קודם לבצע הפעולה אחד מהם (או שניהם יחדיו) יעבור שידור, ממדיו יגדלו והפעולה תבוצע רק לאחר שני הטנзорים יהיו מאותו ממד.

הדוגמאות שלහן מלמדות על הכללים של פעולות השידור, שנדונו בהם אחרי כן.

דוגמאות 1 :

```
A = torch.arange(9).reshape(3,3)
B = torch.arange(3)
print(A, B, A+B, sep='\\n')

tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
tensor([0, 1, 2])
tensor([[ 0,  2,  4],
        [ 3,  5,  7],
        [ 6,  8, 10]])
```

פלט:

בדוגמה זו ניתן לראות שקודם לביצוע פעולה החיבור בין המטריצה A מגודל  $3 \times 3$  לבין הטנזור חד-ממדי B, B עבר שידור לאורך שורות A, ופעולה החיבור בוצעה בין A לבין המטריצה הבאה:

```
B.expand_as(A)
tensor([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

**פלט:**

ראו כיצד בעזרת המתוודה השימושית (`expand_as`) ניתן לבצע את השידור בגלוי, לפרק כל פעולה הכלולת שידור לשכלה השוננים ולבחוון כל שלב בנפרד. כמו כן עלייכם לדעת כי פוקודה זו חסכנות בזיכרונו: המטריצה המתקבלת היא **מבט** לתוך הנתונים של הטנзор B (כאשר כל העמודה הראשונה מחייבת אל אותו המקום בזיכרון שבו שבו ש谋ור הערך 0 וכו'). בהתאם לכך, לאחר שימוש בפקודת שידור `clone` שכזו ולפניה ביצוע פעולות המשנות את ערכיו של הטנзор במקומות – יש להשתמש במתודה () המשכפלת את הערכים למקום חדש בזיכרון.

נשאלת השאלה מדוע שידור B ה被执行 דווקא לאורך השורות ולא לאורך העמודות. על פניו ייתכן היה שפעולות השידור תניב מטריצה אחרת, ובהתאם לכך פעולות החיבור תניב תוצאה אחרת. ואכן, לו היינו מצהירים כי B הוא וקטור עמודה (מטריצה מסדר  $1 \times 3$ ), תוצאה השידור הייתה משתנה בהתאם לכך:

```
B.reshape(3,1).expand_as(A)
tensor([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```

**פלט:**

```
A = torch.arange(3).reshape(3,1)
B = torch.arange(4).reshape(1,4)
print(A, B, A+B, sep='\n')

tensor([0,
       [1],
       [2]])
tensor([[0, 1, 2, 3]])
tensor([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5]])
```

**דוגמה 2:**

בדוגמה זו ניתן לראות שתי המטריצות עברו שידור: העמודה היחידה של A שוכפלה ארבע פעמים. בעוד השורה של B שוכפלה שלוש פעמים, והתוצאה שהתקבלה היא סכום שתי המטריצות, בגודל  $4 \times 3$ .

## דוגמה 3:

```
A = torch.arange(6).reshape(3, 2)
B = torch.arange(5).reshape(1, 5)
print(A, B, sep='\\n')
A+B
tensor([[0, 1],
        [2, 3],
        [4, 5]])
tensor([[0, 1, 2, 3, 4]])
RuntimeError: The size of tensor a (2) must match the size of
tensor b (5) at non-singleton dimension 1
```

פלט:

בדוגמה זו ניתן לראות שלא כל זוג טנзорים מתאימים לשידור ייחדיו. במקרה הנוכחי הסיבה לכך היא חוסר ההסכמה על גודל הממד השני: ל-A שתי עמודות בעוד ל-B חמש עמודות.

ושלוש הדוגמאות שראינו אנו לומדים את חוקי השידור הכלליים:

1. לפני ביצוע פעולה טנורית איבר-איבר על שני טנзорים A ו-B בעלי גודל שונה, יש לבדוק אם הם מתאימים לשידור:
    - a. מתחילה מהממד האחרון (הימני ביותר) של שני הטנзорים ובודקים ש:
      - i. הם שווים זה לזה, או
      - ii. אחד מהם שווה ל-1.
    - אם אף אחד מתנאים אלו אינו מתקיים – התקבל שגיאה, כמו זו שהתקבלה בדוגמה 3 לעיל.
  - b. עבורים ממד אחד שמאליה בכל אחד מהטנзорים וחוזרים על הבדיקה שלו.
  - c. כאשר סיימנו לעבור על כל הממדים של לפחות אחד הטנзорים מימין לשמאל ולא התקבלה שגיאה, אפשר לעבור לשלב השידור.
- בשלב השידור משכפלים את ערכי הטנзорים לפי החוקיות זו:
- a. אם אחד הטנзорים הוא מגודל קטן יותר מהשני (כלומר בעל פחות ממדים), מוסיפים בתחילתו ממדים מנוקנים עד אשר מספר הממדים בשני הטנзорים שווה.
  - b. ככל מקרה שיש אי-הסכמה בערך הממד – בדיקת החתامة לשידור הסטיימה בהצלחה ועל כן אחד הטנзорים הוא בעל ממד מנוקן בגודל 1, אותו נשכפל לאורך ממד זה.

נסיים פרק זה בדוגמה נוספת הממחישה את חוקי השידור.

## דוגמיה 4:

```

A = torch.arange(24).reshape(2,3,4)
B = torch.tensor([0,1,2,3])
C = B.expand_as(A)
D = A+B
print(A.size(), B.size(), C, D, sep='\n')

```

פלט:

```

torch.Size([2, 3, 4])
torch.Size([4])
tensor([[[0, 1, 2, 3],
          [0, 1, 2, 3],
          [0, 1, 2, 3]],

         [[0, 1, 2, 3],
          [0, 1, 2, 3],
          [0, 1, 2, 3]]])
tensor([[[ 0,   2,   4,   6],
          [ 4,   6,   8,  10],
          [ 8,  10,  12,  14]],

         [[12,  14,  16,  18],
          [16,  18,  20,  22],
          [20,  22,  24,  26]])]

```

בדוגמה זו ניתן לראות שהテンзорים A ו-B מותאימים לשידור, שכן הממד האחרון של A שווה לממד היחיד (והאחרון) של B. לכן B עובר שידור לטזוזר תלת-ממדי בגודל 1X1X4, ואחרי כן איבריו של B משוכפלים לאורך הממדים האחרים, כך שפעולות החיבור מתבצעת למעשה בין המטריצה A ו-C. לסיום דוגמיה זו, ודאו כי אתם מבינים שעבור כל הצבה חוקית של אינדקסים, החישוב הבא מניב ערך True.

```
C[i,j,k] == B[k]
```

**שאלה לתרגול**

נתונים טנזוריים כלהלן בעלי הממדים הבאים :

A	$2 \times 1 \times 2$
B	$2 \times 1 \times 1 \times 3$
C	$5 \times 1$
D	$1 \times 5$
E	1
F	$5 \times 3$

- א. קבעו אילו זוגות של טנזוריים מותאימים לשידור זה עם זה.  
 ב. לזוגות המותאימים לשידור – מהו ממד הטnzור המתקבל מהשידור?

## גזרה אוטומטית

בפרק זה נזכיר את אחד הכלים השימושיים ביותר בספרייה PyTorch – הגזרה האוטומטית. כלי זה, המומוש בחבילת autograd של הספרייה, מאפשר לנו להציג מראש על טנזורים כ"משתנים" אשר בהמשך נרצה לגוזר פונקציות שלהם, וזאת באמצעות שינוי המאפיין `requires_grad` לערך אמת (לרוב בעת ייצירת הטנзор), כדלהלן:

```
x = torch.tensor([2.], requires_grad=True)
```

לאחר הצהרה זו, הספרייה תתחילה בעקבות אחרי כל פעולה חשבון שבוצעת על הטנзор, ולבסוף כאשר ברצוננו לחשב את הנגזרת של פונקציה כלשהי לפי  $x$ , כל שעליינו לעשות הוא להרים את המתודה `backward()`. הנגזרת תחשב אוטומטית ותשמר במאפיין `grad` של הטנзор. ראו למשל:

```
y = x**2
z = y + x
z.backward()
print(x.grad)

tensor(5.)
```

**פלט:**

לפנינו שתשיכו לקרוא, ודאו שגם מבניים כי הקשר הפונקציונלי בין  $z$  לבין  $x$  הוא  $z = x^2 + x$ , ולכן הנגזרת היא  $\frac{dz}{dx} = 2x + 1$ , ובמצבתה הסקלר 2 ב- $x$  אכן מקבל הערך 5.

פונקציונליות זו עובדת כמובן גם עבור נגזרות חלקיות, כפי שנitinן לראות בשתי הדוגמאות שלහן:

### דוגמה 1

```
x = torch.tensor([2., 3.], requires_grad=True)
y = x[0]*x[1]
y.backward()
print(x.grad)

tensor([3., 2.])
```

**פלט:**

בדוגמה זו אנו רואים כי הקשר הפונקציונלי בין  $y$  לבין הווקטור  $x$  הוא  $y = x_0 \cdot x_1$  ובמאפיין `grad`

$$\nabla y = \left( \frac{\partial y}{\partial x_0}, \frac{\partial y}{\partial x_1} \right) \text{ נשמר הגרדייאנט}$$

## דוגמה 2

```

x = torch.tensor([2., 3.], requires_grad=True)
y = torch.tensor([4., 5.], requires_grad=True)
w = x.sum() ** 2
z = w + y.sum() ** 3 + x[0] + y[1]
z.backward()
print(x.grad, y.grad, sep='\n')

```

פלט:

```

tensor(11., 10.)
tensor(243., 244.)

```

בדוגמה זו אנו עוקבים אחרי פעולות החישוב שמבצעו טנзорים שונים **בזמן-**  
**המתקדם** () `backward` מחושב בזמן-  
**הGRADEANT** של `z` ביחס לכל אחד מהם. שימושם לבן קשור  
**הפונקציונלי** בין הטנзорים המופיעים בדוגמה,

$$z = (x_0 + x_1)^2 + (y_0 + y_1)^3 + x_0 + y_1$$

ובהתאים לכך וודאו כי אתם מבינים את חישוב הנגזרות:

$$\frac{\partial z}{\partial x} = \left( \frac{\partial z}{\partial x_0}, \frac{\partial z}{\partial x_1} \right) = (2(x_0 + x_1) + 1, 2(x_0 + x_1))$$

$$\frac{\partial z}{\partial y} = \left( \frac{\partial z}{\partial y_0}, \frac{\partial z}{\partial y_1} \right) = (3(y_0 + y_1)^2, 3(y_0 + y_1)^2 + 1)$$

לעתים נרצה לבצע חישוב על טנзорים שאנו עוקבים אחריהם ולהשתמש בתוצאה **קבוע**, ולא  
**כפונקציה** של הטנзорים המקוריים, שאויה נרצה לגוזר בעת הרצת () `backward`. למקרה זה ניתן  
להשתמש במתודה **היווצרת עותק** של הטנзор המקורי, אך ללא מעקב אחריו פעולות החישוב. השוו את  
הדוגמה שלහלו קודם לכן, וודאו כי אתם מבינים שלמעשה כתת מתקיים

$$z = s + (y_0 + y_1)^3 + x_0 + y_1$$

לא התחשבות בעובדה ש-`s` חושב כפונקציה של הטנзор `x`.

```

x = torch.tensor([2., 3.], requires_grad=True)
y = torch.tensor([4., 5.], requires_grad=True)
w = x.sum()**2
s = w.detach()
z = s+y.sum()**3+x[0]+y[1]
z.backward()
print(x.grad, y.grad, sep='\n')

```

פלט:

```

tensor([1., 0.])
tensor([243., 244.])

```

לסיוום, נציג שמערכת הגזירה האוטומטית עוקבת **בעת הריצת** אחרי פעולות החשבון המבוצעות על הטנзорים. בהתאם לכך, אם בכל ריצה מחושבת פונקציה שונה, למשל בעקבות החלטות שונות של בקרת הזרימה, רק הפעולות שבוצעו בפועל הן אלו שיובאו בחישובן בעת הגזרה. ראו בדוגמאות של להלן כיצד אנו גוזרים "דרך" תנאים ולולאות באופן מפורע:

**דוגמה 3**

```

y = torch.zeros(10,dtype=int)
x = torch.tensor([2.], requires_grad=True)
for i in range(10):
    z = 2*x
    while torch.rand(1)<0.7:
        z = 2*z
    z.backward()
    y[i] = x.grad
    x.grad = None
print(y)

```

פלט:

```

tensor([ 8, 32,   2,   4,  64,   2,  64,  16,   2,  32])

```

שיםו לב שבקטע קוד זה הקשר הפונקציוני בין  $z$  לא תלוי בהגרלה מספרים אקרים:  $x=2^{1+k}$  כאשר  $k$  הוא מספר הפעמים שהיא עליינו להגריל מ"מ אחידים סטנדרטיים, עד שהתקבל אחד שערכו גדול מספיק. לפיכך, בכל איטרציה של הלולאה החיצונית מתקבל קשר אחר, וכך גם ערכי הנגזרת שונים לבסוף.

כמו כן, שימו לב לפקודה "backward ()". כל קריאה למתודה `x.grad=None` מוסיפה את הערך שחוושב לזה הקיים במאפיין `grad`. כלומר, מוביל לאפס אותו קודם לכך. לפיכך, פלט טיפוסי של הקוד הנ"ל ללא פקודה זו ייראה כך:

```

tensor([ 2, 18, 20, 52, 84, 148, 150, 406, 414, 422])

```

מן ניכר שבכל איטרציה של הלולאה נוסף ערך הנגורת של החישוב הנוכחי אל ערך כל החישובים הקודמים.

#### דוגמה 4

כעת נראה שאוטו עיקרונו חל גם על תנאי if-else : **בכל איטרציה הגזירה מותבצעת ביחס לענף הנבחר בבדיקה הזרימה.**

```
y = torch.zeros(5)
for i in range(5):
    x = torch.randn(1, requires_grad=True)
    if x>0:
        z = torch.exp(x)
    else:
        z = -x
    z.backward()
    y[i] = x.grad
print(y)

平淡:
tensor([ 1.1121, -1.0000, -1.0000,  2.4863, -1.0000])
```

**פלט:**

### שאלות לתרגול

- הרכזו את הדוגמה הראשונה בפרק זה (לפני דוגמה 1), ומיד לאחר הרכיצה קראו שוב למתחודה ()backward(). האם תוכלו להסביר את מקור השגיאה המתקבלת?
- הרכזו שוב את הדוגמה, אך כעת בעת הקריאה הראשונה למתודה backward() הוסיפו את הפרמטר `create_graph=True`. לאחר מכן הרכזו את הקוד :

```
t = x.grad
x.grad = None
t.backward()
```

נסה להסביר מהי התוצאה המתקבלת אחרי כן במשתנה `x.grad`.

## ICHIDA 2: צעדים ראשוניים

### דגימת נתונים סינטטיים והגדרת מודל הסיווג

רשתות הנוירונים אשר נשתמש בהן בהמשך למטרות מורכבות, כגון תרגום בין שפות או זיהוי אובייקטים בתמונה, דורשות מורכבות חישובית הנובעת בין השאר מכך שהן כוללות מיליוני נוירונים המוחברים זה לזה בדרכים מתחכਮות. למשל, גם היחידה הבסיסית של מרכיבים זו, הנוירון היחיד, מאפשרת בפני עצמה שימוש של מטרות כאלה, אם כי פשוטות יותר. למשל נוירון יחיד יוכל להתמודד עם סיווג ביןרי של קבוצות של נקודות פשוטות במרחב דו-ממדי.

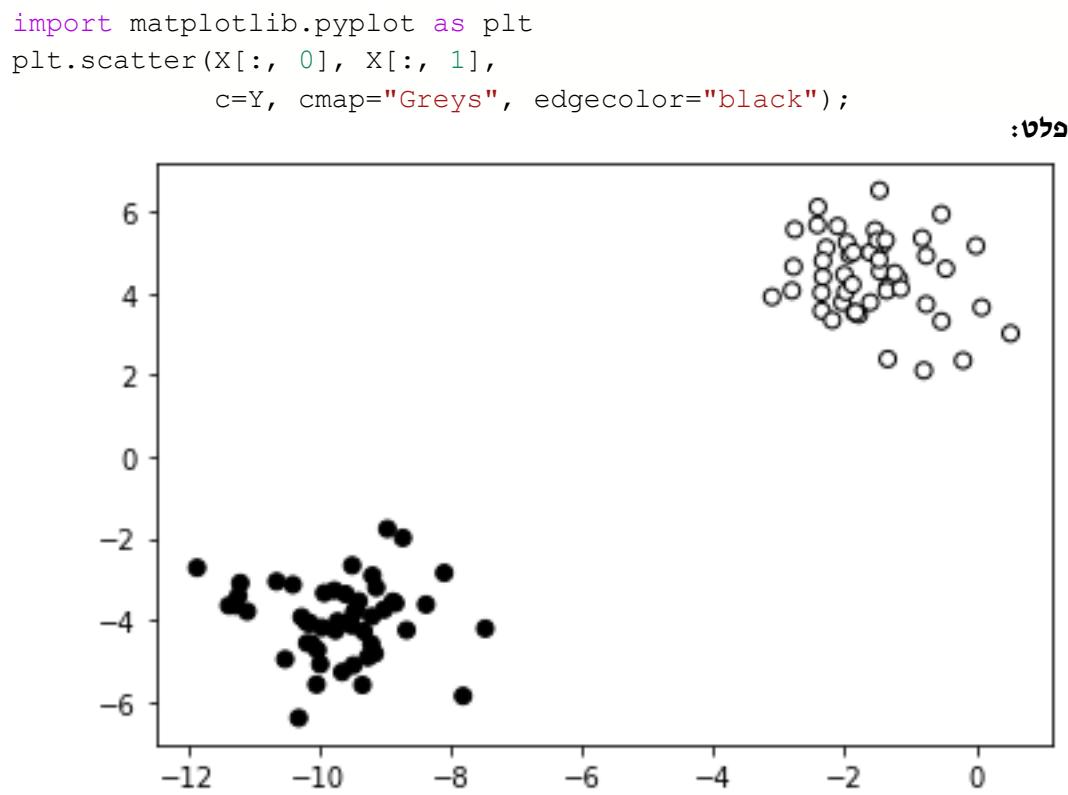
בעזרת הספרייה `scikit-learn` נוכל ליצור אוסף נתונים כאלו בקלות. בקטע הקוד שלහלן, ניצור בפקודה אחת דוגמאות רנדומליות משתי מחלקות שונות.

```
import sklearn.datasets as skds
X, Y = skds.make_blobs(n_samples=100, n_features=2,
                        centers=2, random_state=1)
print(X[:5,:], type(X))
print(Y[:5], type(Y))
```

פלט:

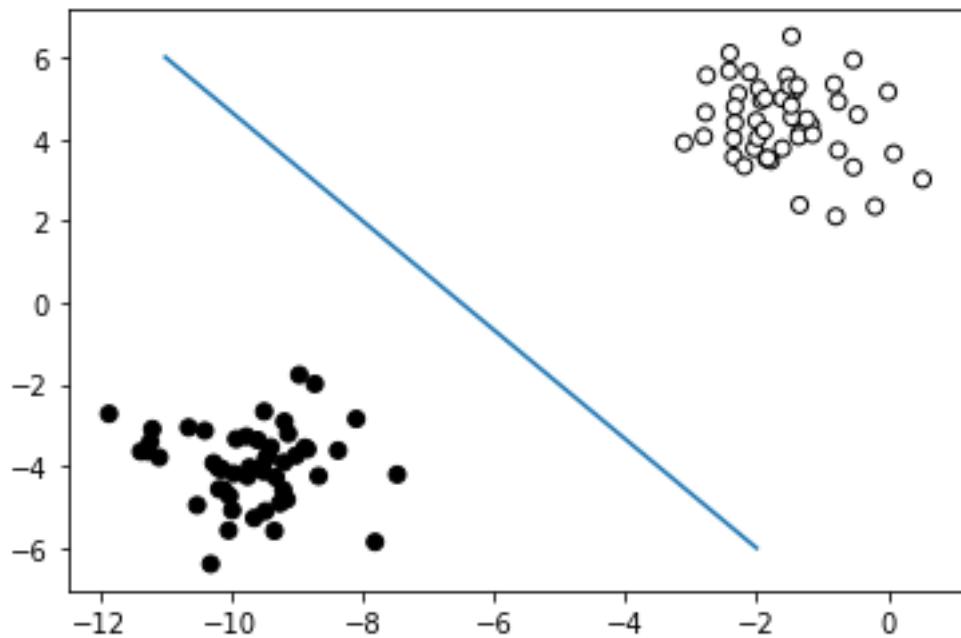
```
[[ -0.79415228  2.10495117]
 [ -9.15155186 -4.81286449]
 [ -3.10367371  3.90202401]
 [ -1.42946517  5.16850105]
 [ -7.4693868 -4.20198333]] <class 'numpy.ndarray'>
[0 1 0 0 1] <class 'numpy.ndarray'>
```

ניתן לראות שקובורדינטות הנתונים והסיווג למחולקות התקבלו במשתנים מסוג `ndarray` של NumPy, וכן נוכל להמיר אותם בקלות בעת הצורך לטזוריים של Pytorch באמצעות הפונקציה `.torch.tensor()`. בנוסף על כן, בעזרת הספרייה `matplotlib` נוכל לצייר את הנתונים במישור:



שימוש לב לשורה השניה בקריאה לפונקציה (`scatter()`), שבה אנו קובעים שהנקודות ייצבו בצבועים שונים לפי הערכים במשתנה `z`; שהצבועים האלו יהיו שחורים/לבנים; ושהלוי הנקודות יהיו צבועים בשחור (כדי שונכל לראות את הנקודות הלבנות); וזאת באמצעות שינוי של שלושת הפרמטרים המתאימים. ניתן לשנות פרמטרים רבים נוספים של הגרף, כגון החלפת העיגולים בריבועים או כוכבים, הפיכת העיגולים לחצי שקופים ועוד. אין צורך לזכור בעלפה פקודות אלו, אלא לחפש את הדורש בתיעוד של `.matplotlib`.

אוסף נתונים זה פשוט די כדי שנירנו יחד יכול להבדיל בין נקודות שחורות לבנות, וזאת מפני שהנקודות ניתנות להפרדה בעזירת פונקציה לינארית, כפי שנבין בהמשך הלימוד. לעת עתה רק נשים לב שבבירור ניתן לצויר במישור קו המפריד בין שתי המחלקות, למשל:



כעת כשאוסף הנתונים בידינו, נפנה להגדרת מודל הסיווג, המורכב מנוירון יחיד :

1. הנוירון קיבל כקלט את קואורדינטות הנקודות,  $(x_0, x_1)$ .

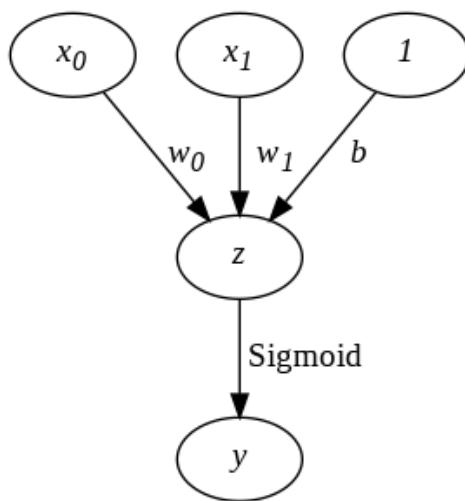
2. על הקלט תופעל פונקציה לינארית  $z = w_0x_0 + w_1x_1 + b$ .

3. תוצאה החישוב תועבר לפונקציית אקטיבציה (activation function)

$$y = S(z) = \frac{1}{1 + e^{-z}}$$

4. פלט הנוירון,  $y$ , הוא **הסתברות** שהנקודה הנתונה שייכת למחלקה הנקודות השחורות.

ובאיור,

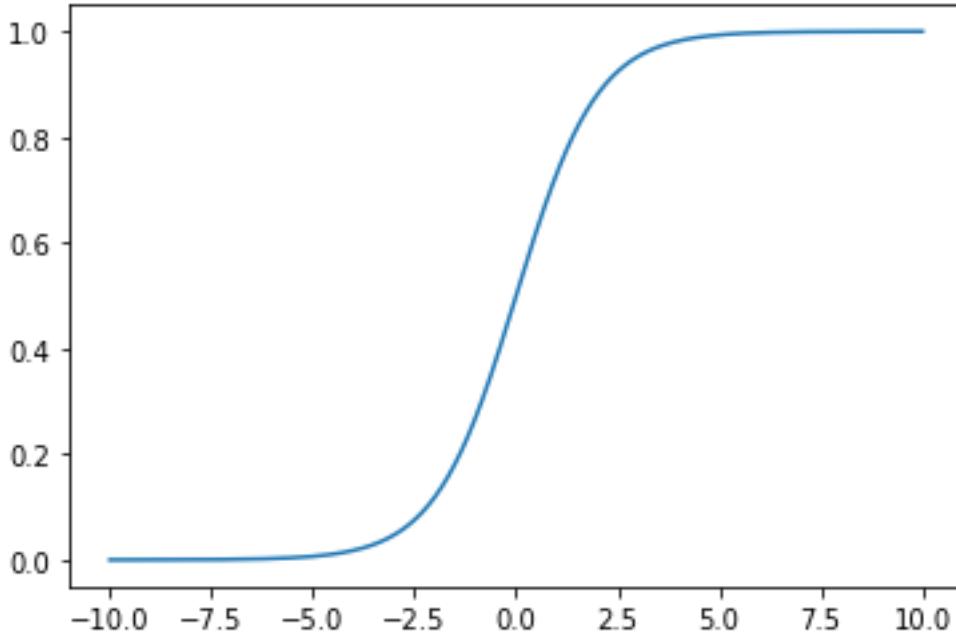


שימוש לב שפונקציית האקטיבציה עברו נוירון זה נקראת סיגמוואיד, שכן גוף הֆונקציה  $y = S(z)$  דומה בצורתו לאות S, אשר מתחילה ב-0 ונגמרת ב-1. בהתאם לכך, אפשר לפרש פלט

אקטיבציה זו כהסתברות, ולכן היא שימושית למשימות סיוג ביני. בקטע הקוד שלහלן אנו מצירירים את גרפ הפונקציה לצורך המחבר.

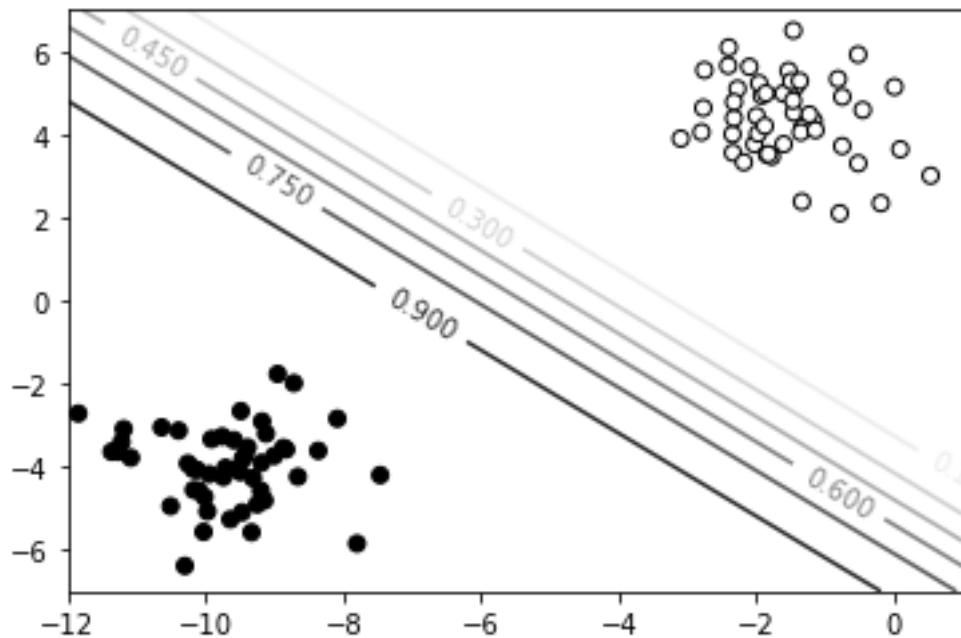
```
import numpy as np
z = np.linspace(-10,10,1000)
y = 1 / (1+np.exp(-z))
plt.plot(z,y);
```

פלט:



מטרתנו כעת היא למצוא את ערכי הפרמטרים של המודל, הלא הם  $a_0, a_1, b$ , כך שהמודל יבצע עבורותנו נאמנה: כאשר קלט הנירון יהיה נקודה לבנה, נרצה שהמודל יחזיר כפלט הסתברות קרובת ל-1, ועבור נקודה שחורה – הסתברות קרובת ל-0. במקרה הנוכחי מושימה זו קלה במיוחד, שכן פרמטרים אלו קובעים את מיקומו של הקו המפריד בין המחלקות, כמווייר לעיל, וככל שנקודות הקלט מרוחקת מהקו, כך ההסתברות שהמודל יחזיר תהיה קרובה יותר ל-1 (עבור נקודות מתחת לקו) או ל-0 (עבור נקודות מעל הקו). לכן, אם נבחר קו העובר "באמצע", נקבל מודל מעולה.

למשל עבור בחירה מסוימת של פרמטרים, נקבל את המודל המופיע בציור שלහלן. שימוש לב שלל כל קו מצוינת ההסתברות שנקודה הנמצאת עליו תהיה שחורה, על פי המודל. כדריש, על הקווים הקרובים למחלקה הלבנה מצוינת הסתברות נמוכה, ועל אלו הקרובים למחלקה השחורה – הסתברות גבוהה.



בعتיד, כאשר הביעות שנתמודד איתן יהיו מורכבות יותר, ובהתקף להן גם המודל, לא תהיה לנו אפשרות לבחור את הפרמטרים המתאימים באופן ידני, ועל כן נדרש לנו אלגוריתם שיבצע זאת אוטומטית. נשלב אלגוריתם כזה בדוגמה זו בהמשך הלימוד.

### שאלות לתרגול

1. הסבירו למה קווי הגובה (קוויים אשר לכל הנקודות עליהם הסתברות סיווג זהה) באירוע האחרון הם קוויים ישרים.

2. א. הניחו שבחרנו סט פרמטרים מסוימים עבור הנוירון. מצאו את משווהת הישר אשר כל נקודה מתחתתיו מסובגת בהסתברות גבוהה מ-0.5 להיות נקודה שחורה.

ב. האם יש משווהה נוספת המתארת את אותו קו?

ג. האם יש הבדל בין המודלים המתקבלים משנה סטים של פרמטרים אשר עברום הקוו המפריד זהה? אם כן, הסבירו מהו.

3. החליפו את פונקציית האקטיבציה של הנוירון לReLU :

$$\text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

א. ציירו גרף של פונקציית האקטיבציה.

ב. נסו להשתמש באקטיבציה זו במקום הסיגמוואיד לצורך סיווג הנקודות, והסבירו את הקשיים שאתם פוגשים.

ג. נסו לפתרור את הקשיים באמצעות שינוי קל של המודל.

## פונקציית המחיר ואלגוריתם האימון של הנוירון

### פונקציית מחיר

בפרק הקודם רأינו כיצד אפשר להשתמש בנוירון יחיד כדי לסוג אוסף נתונים לשתי מחלקות, ובפרק זה נלמד כיצד לעשות זאת אוטומטית. ראשית, נבחר **פונקציית מחיר** (cost function) אשר תבטא את מידת ההתאמה של המודל הנבחר לנtones נטענים שנשתמש בהם לאימון המודל. כאשר המודל יתאים לנtones, קרי יסוגם נכון, המחיר יהיה נמוך, וכאשר לא – המחיר יהיה גבוה.

נזכיר שהמודל הנידון לסייע נקודות שחורות ולבנות מקלט קווארדיינטאות של נקודות במרחב דו-ממדי, ומהזיר כפלט את הפונקציה

$$y = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + b)}}$$

לפייך, הפרמטרים של המודל שבאפשרותנו לשנות כדי להתאים לנtones הם  $w_0, w_1, b$ , ולכן פונקציית המחיר תהיה אך ורק בהם.

בבניות של סיוג נתונים למחלקות נהוג לבחור בפונקציית המחיר **האנטרופיה הצולבת** (cross entropy), אשר עבר בעיית הסיוג לשתי מחלקות מוגדרת כך:

$$C(w_0, w_1, b) = -\frac{1}{\#Data} \sum_{(x_0, x_1, y_t) \in Data} y_t \log(y(x_0, x_1)) + (1 - y_t) \log(1 - y(x_0, x_1))$$

יש לשים לב כמה פרטיים בנוסחה זו:

1. אוסף הנתונים שלנו מסומן ב- $Data$ , ומספר הדגימות בו ב- $\#Data$ .
2. הסכום רץ על כל הדגימות באוסף הנתונים, המייצגות בתור שלשות מהצורה  $(x_0, x_1, y_t)$ .
3.  $y_t$  הוא הסיוג האמתי של הדגימה.  $y_t = 1$  כאשר מדובר בנקודה שחורה, ו- $y_t = 0$  כאשר מדובר בנקודה לבנה.
4.  $y(x_0, x_1)$  היא ההסתברות שמקנה המודל לכך שהנקודה  $(x_0, x_1)$  היא שחורה. ערך זה הוא פונקציה של פרמטרי המודל וכאן באה כדי ביטוי התלות של פונקציית המחיר בפרמטרים.

5. כל נקודה דוגמה מופיעה רק פעם אחת בסכום, ועל כן אפשר לכתוב את הנוסחה כדלקמן,

$$C(w_0, w_1, b) = \frac{1}{\#Data} \sum_{(x_0, x_1, y_t) \in Data} H(x_0, x_1, y_t)$$

כאשר

$$H(x_0, x_1, y_t) = -[y_t \log(y(x_0, x_1)) + (1 - y_t) \log(1 - y(x_0, x_1))]$$

היא התרומהמחיר של נקודה הדגימה  $(x_0, x_1, y_t)$ .

מטרתנו كانت היא למצוא נקודת מינימום של פונקציית המחיר – ערכי פרמטרים המביאים את  $C$  לערך הנמוך ביותר. המודל שיתקבל עבור ערכי אלו יהיה מוצלח במיוחד, שכן סיוג שゴי של נקודות דוגמה כלשהי יוביל לכך שתרומתו למחיר תהיה גדולה. על מנת לראות זאת בפירוט, שימו לב לכך שבביטוי לתרומה למחיר מופיע סכום של שני איברים, אך אחד מהם תמיד מתאפס – בהתאם לסיוג האמתי של הנקודה לבנה או שחורה. למשל, אם הנקודה לבנה, אז  $y_t = 0$  והביטוי מצטמצם ל-

$$H(x_0, x_1, y_t) = -\log(1 - y(x_0, x_1))$$

כעת, על הנירון לסייע נקודה זו כלבנה, ובהתאם לאות ברצונו לקבל ב- $(x_0, x_1)$  את הסתברות הקרויה לו. לו היה הנירון שוגה בסיווג נקודה זו, היה הביטוי  $(x_0, x_1)$  – 1 קטן מאחד באופן משמעותי ולכך. הלוגריתם שלו היה שלילי וגדול בערכו המוחלט, דבר אשר היה מוביל לתורמה למחיר חיובית וגדולה. על כן, מודל שיסוווג נקודה זו נכונה "ייקנס" במחיר נמוך יותר, ומודל מוצלח במיוחד יימנע מסיווגים שוגיים מאוד עבור כל הנקודות.

חדי העין ישימו לב שהמודל שאנו רוצים לאמן הוא למעשה רגרסיה לוגיסטיבית, ופונקציית המחיר הנ"ל קשורה בקשר הדוק לפונקציית הנראות (likelihood) של בעיית רגרסיה זו. מציאת נקודת המינימום של  $C$  שקופה למציאת אומד הנראות המקסימלית בעיית הרגרסיה, ונקודת מבט זו מספקת צידוק נוספים לשימוש בה.

בעוד שניתן להוכיח של בעיית הרגרסיה הלוגיסטיבית קיים פתרון ייחיד, לרוב המצב עבור רשתות נוירונים הפוך: ישנים כמה ערכי פרמטרים שונים המזערירים את פונקציית המחיר, ולא רק זאת, אלא גם אין שיטה אחת המבטייחה את מציאתם. על כן נרבה להשתמש בשיטות נומריות לקירוב נקודת המינימום, וכן חשוב להכיר את האלגוריתם העומד בבסיסו כבר כעת.

## אלגוריתם האימון: מורד הגרדיאנט

מורד הגרדיאנט (gradient descent) הוא אלגוריתם קלאסי ופשוט למציאת מינימום של פונקציה. בהמשך הקורס נלמד גרסאות מתקדמות שלו שיש בהן שכליים שונים אשר נועדו להאיץ את פועלתו ולהתחרם מביעות נפוצות הצאות בעת אימון רשתות נוירונים. פרק זה נשתמש בגרסתו הבסיסית.

פעולתו של האלגוריתם מבוססת על כך שבכל נקודה במרחב הפרמטרים, תנועה קטנה בכיוון השילילי של הגרדיאנט  $\nabla C$  תוביל לירידה בערך פונקציית המחיר  $C$ , וכן סביר שסדרה של צעדים קטנים ככלו תוביל לנקודת מינימום.

על כן, תהליך מציאת הפרמטרים המתאימים ביותר לבעיית הסיווג שלנו יתבסס על שני צעדים שיש לחזור עליהם עד למציאת פתרון סביר:

1. חישוב הגרדיאנט עbor ערכי הפרמטרים הנוכחיים,

$$\nabla C(w_0, w_1, b) = \left( \frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial b} \right)$$

2. עדכון ערכי הפרמטרים הנוכחיים:

$$(w_0, w_1, b) = (w_0, w_1, b) - \alpha \nabla C(w_0, w_1, b)$$

שימוש לב שאות הפרמטר  $\alpha$  של האלגוריתם, שmbטאת את גודל הצעד בכל שלב, יש לקבוע לפני ההרצה. גודל צעד קטן מדי יוביל לכך שהאלגוריתם לא יתכנס גם לאחר מספר רב של איטרציות, בעוד שגודל צעד גדול מדי יוביל לאייציבות בתהליכי: ייתכן שעדכון הפרמטרים "נדלג" מעל נקודת המינימום של  $C$ , וכך ערך הפונקציה לא ירד. בהמשך הקורס נקבע תשומת לב רבה לפרמטר זה.

ובכן, בכספי לממש אלגוריתם זה בקורס, علينا לדעת כיצד לחשב את הגרדיינט. בעתיד נשתמש בכל הגזרה האוטומטית של PyTorch, מוביל להידרשות לעיסוק בפרטים הקטנים, אך עבור דוגמה פשוטה זו ולצורך התרגול נעשה זאת בעצמנו.

ראשית, נזכיר שפונקציית המחיר  $C$  היא מוצע התורומות למחיר של כל אחת מהנקודות הנתונות:

$$\cdot C = \frac{1}{\#Data} \sum_{(x_0, x_1, y_t) \in Data} H(x_0, x_1, y_t)$$

מכך נסיק שדי לנו להתרכז בחישוב הגרדיינט  $\nabla H$  לכל דוגימה ולבסוף לחשב ממוצעו, וזאת מפני שלפי כלל הנגזרת של סכום:

$$\cdot \nabla C = \frac{1}{\#Data} \sum_{(x_0, x_1, y_t) \in Data} \nabla H$$

כעת علينا לחשב את הגרדיינט

$$\cdot \nabla H = \left( \frac{\partial H}{\partial w_0}, \frac{\partial H}{\partial w_1}, \frac{\partial H}{\partial b} \right)$$

אך הפונקציה  $H$  תלואה בפרמטרים של המודל דרך האקטיבציה  $y = 1 / (1 + e^{-z})$  אשר בתורה תלואה בהם דרך האgregציה  $z = w_0 x_0 + w_1 x_1 + b$ , דהיינו  $H$  היא הרכבה של פונקציות:

$$\cdot H(w_0, w_1, b) = H(y(z(w_0, w_1, b)))$$

במקרה זה ככל השרשת יכול علينا את החישוב, שכן:

$$\frac{\partial H}{\partial w_0} = \frac{\partial H}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_0}$$

$$\frac{\partial H}{\partial w_1} = \frac{\partial H}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_1}$$

$$\frac{\partial H}{\partial b} = \frac{\partial H}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial b}$$

שימושו לב שנוסף על פירוק החישוב האנליטי של נגזרות אלו לשלבים פשוטים, בשימוש בכלל השרשת יש גם יתרון חישובי: נוכל לחשב את הנגזרות

$$\frac{\partial H}{\partial y}, \frac{\partial y}{\partial z}$$

פעם אחת בלבד בכלל איטרציה של האלגוריתם, ולכפול אותן בשלוש הנגזרות של  $z$  כדי לקבל את  $H$ .  
מחישוב ישיר נסיק כי:

$$\frac{\partial H}{\partial y} = - \left( \frac{y_t}{y} - \frac{1 - y_t}{1 - y} \right) = - \frac{y_t - y}{y - y^2}$$

$$\frac{\partial y}{\partial z} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} = y(1 - y)$$

$$\cdot \frac{\partial z}{\partial w_0} = x_0, \quad \frac{\partial z}{\partial w_1} = x_1, \quad \frac{\partial z}{\partial b} = 1$$

כעת יש בידינו את כל הדרושים כדי לממש את האלגוריתם בקורס, וזאת נעשה בפרק הבא.

## שאלות לתרגול

1. הציבו את התוצאות האחרונות בנוסחה עבורה רכיבי  $H_7$  בראש עמוד זה, ופשטו את הביטויים המתקברים.
2. פונקציית הנראות של מודל הסטברומי הנועד לחיזוי משתנה תלוי מבטאת עד כמה "סביר" המודל בהסתמך על הנתונים – עד כמה הפרמטרים הנבחרים מתאימים לנ נתונים. בעיית סיווג ביןרי, אשר יש להזות בה את ערך המחלקה  $y = 0$  או  $y = 1$ , פונקציית נראות נפוצה תהיה נתונה בביטוי

$$L(Model) = \prod_{Data} y^{y_t} (1-y)^{1-y_t}$$

כאשר הכפל מתבצע על כל נקודות הדגימה באוסף הנתונים,  $y_t$  הוא הסיווג האמיתי של נקודה הנקודה  $t$  באוסף ו-  $y$  הוא הסיווג שהמודל חוזה עבור נקודה זו.  
א. הסבירו מיהן ההנחות שיש להניח בנוגע לאוסף הנתונים ודרך איסופם כדי לקבל פונקציית נראות שכזו. זכרו שהצורה הכללית של הנראות היא

$$L(Model) = P(Data | Model)$$

קרי – ההסתברות לדגם את אוסף הנתונים מהמודל.

**رمזים:**

1. זכרו שכאש שני מאורעות הם בלתי תלויים מתקיים

$$P(A \text{ and } B) = P(A)P(B)$$

2. שימו לב ש-

$$y^{y_t} (1-y)^{1-y_t} = \begin{cases} y & y_t = 1 \\ 1-y & y_t = 0 \end{cases}$$

וזכרו ש-  $y$  הוא הסיכוי שהנקודה הנתונה היא שחורה, לפי המודל וכן ש-  $y_t = 1$  כאשר הנקודה היא אכן שחורה.

ב. הוכחו שהמקסימום של פונקציית נראות זו מתקבל באותה נקודה שמתקיים בה המינימום של פונקציית המחיר של האנתרופיה הצלובה.

**רמז:** הפעילו את פונקציית הלוגריתם על הנראות.

## אימון הנוירון - מימוש מהיסוד

כעת, כאשר ברשנותנו כל הדרוש לאימון הנוירון בעזרת אלגוריתם מורד הגראדיאנט, ניגש למשימה. ראשית נגדיר פונקציה המחשבת את ההסתברות החזואה, על פי המודל, לכך שנקודה נתונה היא לבנה.

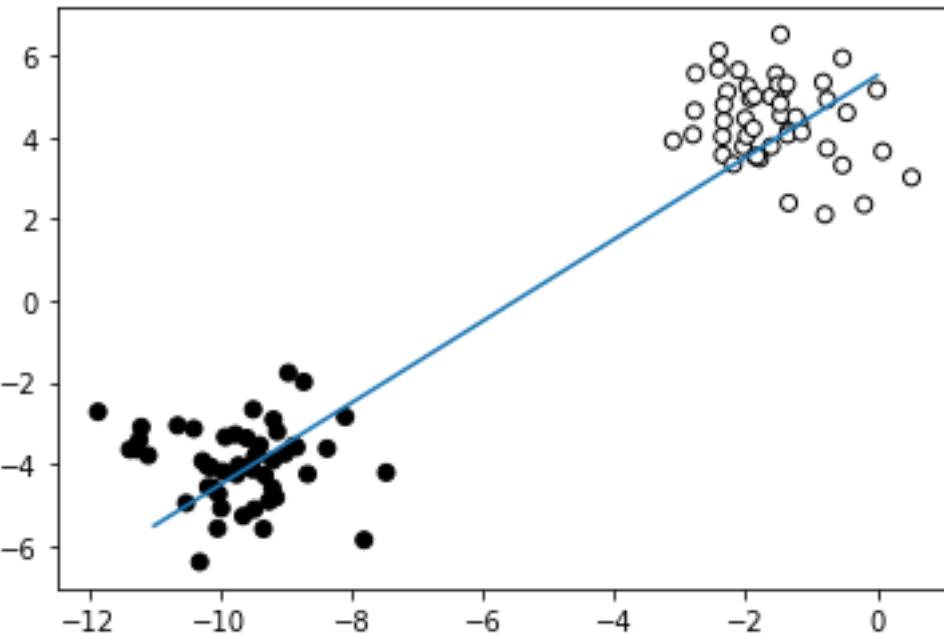
```
z = lambda w0, w1, b, x0, x1: w0*x0+w1*x1+b
y = lambda z: 1/(1+torch.exp(-z))
model = lambda w0, w1, b, x0, x1: y(z(w0, w1, b, x0, x1))
```

אחרי כו, נתחל את האלגוריתם בבחירה גרוועה מיוחד של פרמטרים, כדי להמחיש את פעלת הלמידה.

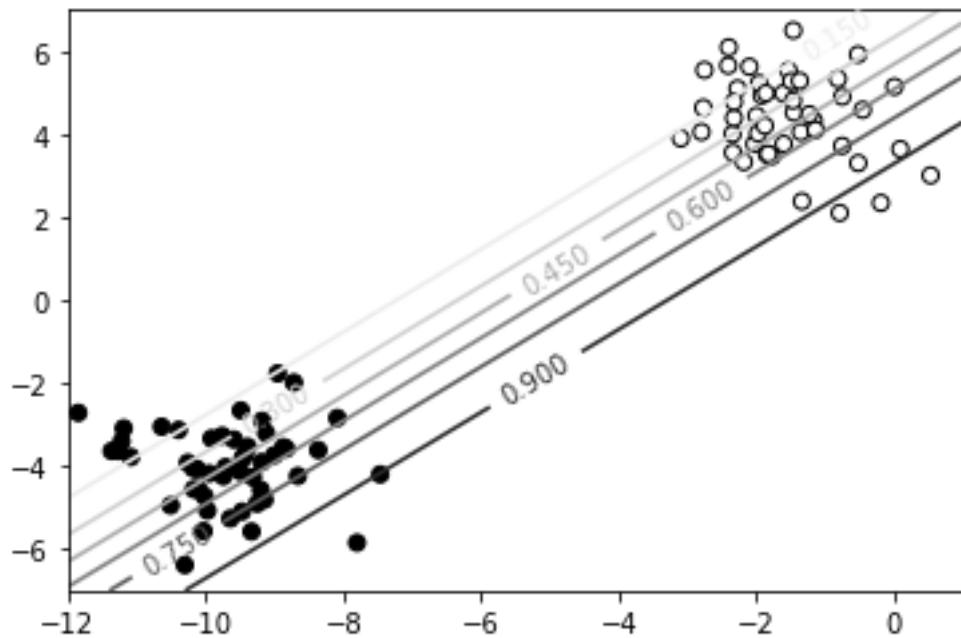
```
def draw_05_line():
    line = lambda x:-w0/w1*x-b/w1
    x0 = torch.tensor([-11, 0])
    x1 = line(x0)

    plt.plot(x0,x1);
    plt.scatter(X[:, 0], X[:, 1],
                c=Y, cmap="Greys", edgecolor="black");
    w0, w1, b = 1, -1, 5.5
draw_05_line()
```

פלט:



עבור ערכי פרמטרים הנבחרים, הקו המפ прид שמתוחתיו המודל יסוווג את הנקודות השחורות בהסתברות גבוהה מ-0.5 מצויר בכחול, וניכר שעבור בחירה זו כחצית מהנקודות יסוווגו נכונה. למעשה המודל לא יביע ביטחון רב בסיווג של אף נקודה, כפי שניתן לראות באירוע שלහן.



נחשב את הגרדיינט של  $H$  עבור כל אחת מנקודות הדגימה, ולבסוף נמצע את הערכים המותקבלים כדי לקבל את  $\nabla C(w_0, w_1, b)$ .

```

dHdy = lambda y,yt: -1*(yt-y) / (y-y**2)
dydz = lambda y: y*(1-y)
dzdw0 = lambda x0: x0
dzdw1 = lambda x1: x1
dzdb = 1
def calc_dC():
    dH = torch.zeros(len(Y), 3)
    for idx in range(len(Y)):
        data      = (X[idx,0], X[idx,1], Y[idx])
        y_model = y(z(w0, w1, b, data[0], data[1]))

        A = dHdy(y_model, data[2])
        B = dydz(y_model)
        dH[idx,0] = A*B*dzdw0(data[0])
        dH[idx,1] = A*B*dzdw1(data[1])
        dH[idx,2] = A*B*dzdb
    return dH.mean(0)
calc_dC()
tensor([ 2.4333,  1.7574, -0.0710])

```

פלט:

בשלב הבא עליינו לעדכן את ערכי הפרמטר לכיוון השילילי של הגרדיינט. נבחר גודל צעד קטן, ונעדק את ערכי הפרמטרים ונציגיר שוב את הקו המפריד בין הסיווג של הנקודות.

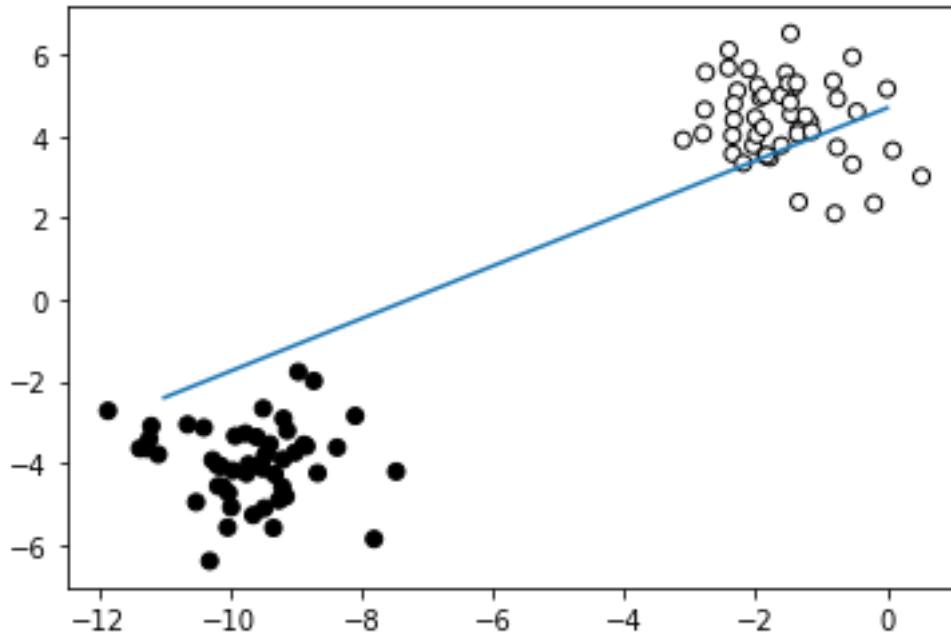
```

alpha = 0.1
dC = calc_dC()
(w0, w1, b) = torch.tensor((w0, w1, b)) - alpha*dC
print(w0, w1, b)
draw_05_line()

```

פלט:

```
tensor(0.7567) tensor(-1.1757) tensor(5.5071)
```



nicer מהציגו שהמודל מסוווג נכונה את רוב הנקודות השחורות. עכשו נחזר על פעולה זו באופן איטרטיבי:

```

w0, w1, b = 1, -1, 5.5
alpha = 0.1

H = lambda y,yt: -(yt*torch.log(y)+(1-yt)*torch.log(1-y))
cost_per_point = H(model(w0, w1, b, X[:, 0], X[:, 1]), Y)

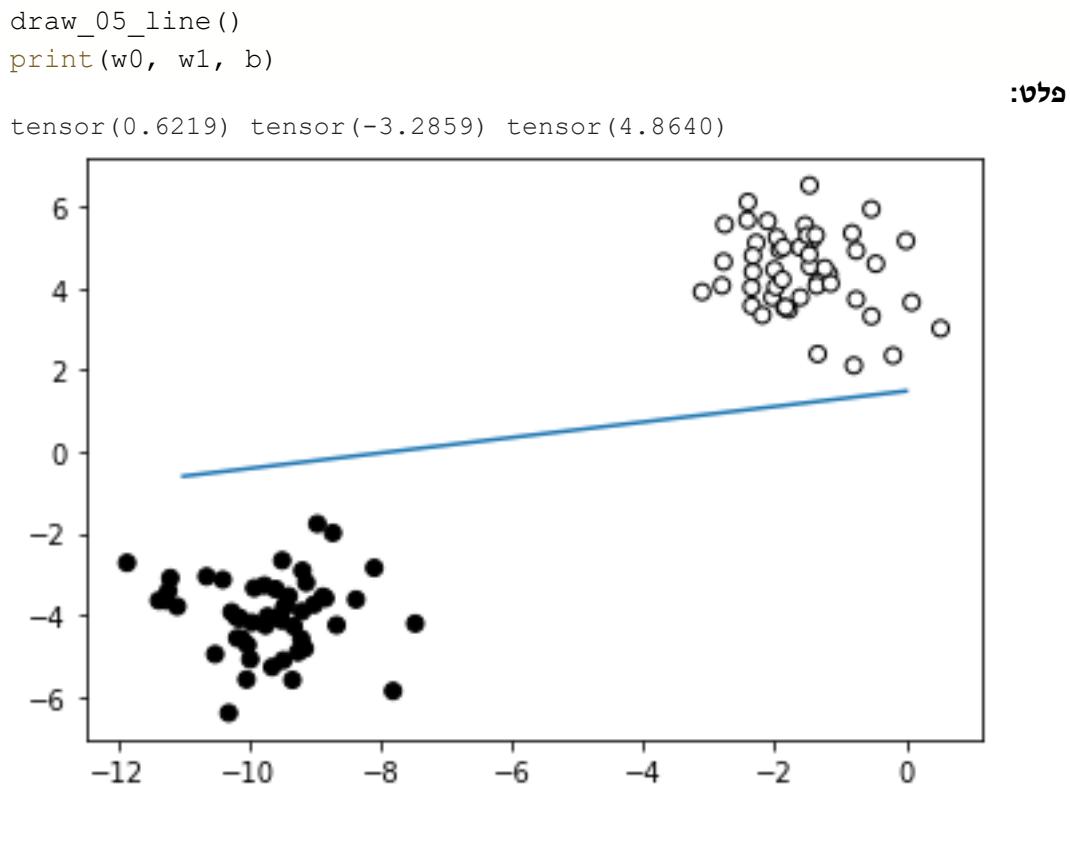
cost = torch.zeros(1000)
for iter_num in range(len(cost)):
    dC = calc_dC()
    params = torch.tensor((w0, w1, b))
    (w0, w1, b) = params - alpha*dC
    cost_per_point = H(model(w0, w1, b, X[:, 0], X[:, 1]), Y)
    cost[iter_num] = cost_per_point.mean()

```

שימוש לב שבסכל איטרציה של עדכון הפרמטרים אנו מחשבים את  $H \nabla$  עבור כל אחת מהנקודות של אוסף הנתונים. כבר בשלב זה, כאשר מדובר ב-100 נקודות בלבד, זמן הריצה של חישוב זה אינו זניח,

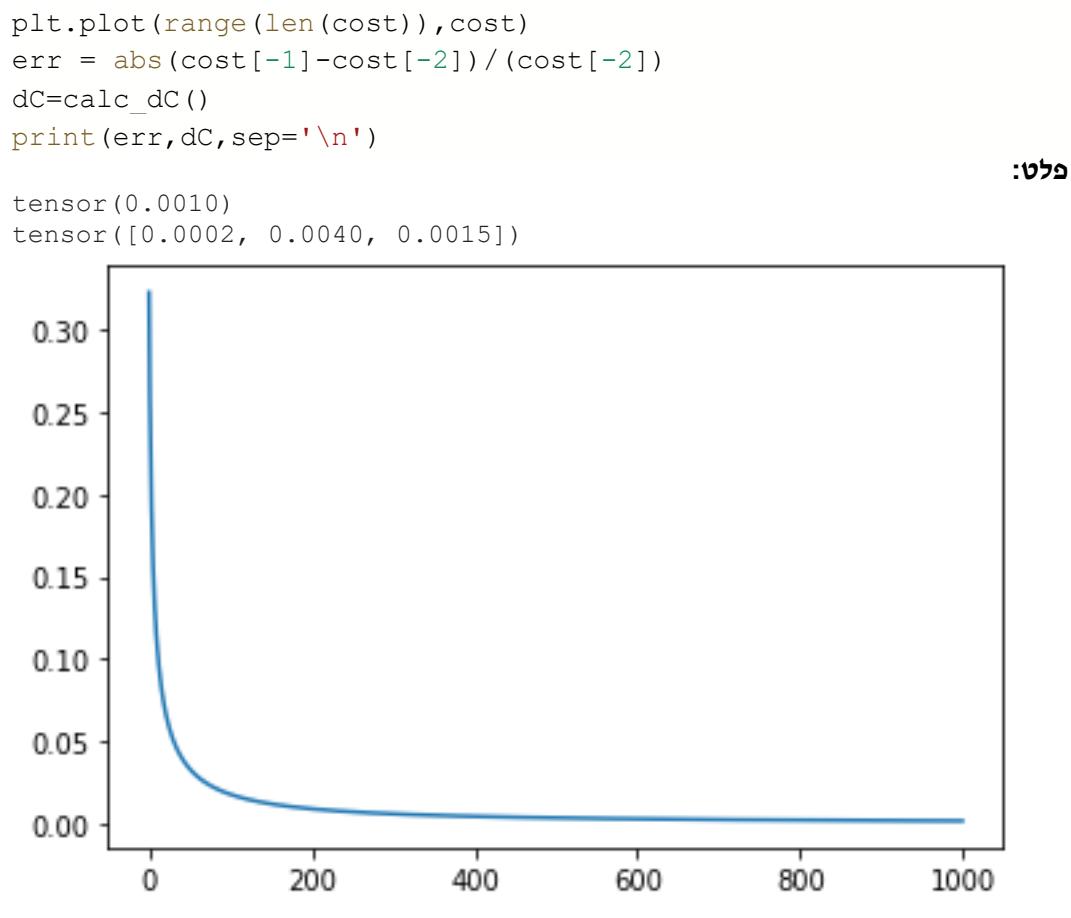
ובעתיד, כאשר באוסף הנתונים יהיו מאות אלפי ואף מיליוני דוגמאות, מחיר חישוב הגרדיאנט על בסיס כל הדוגמאות יהיה גבוה מדי. גם בפתרון בעיה זו נדוע בהמשך הקורס.

אחרי ריצת הקוד ננתן את התוצאות. ראשית נציג את הקו המפריד שהתקבל לאחר 1,000 איטרציות :



קו זה רחוק מלהיות הקו האידיאלי, שכן הוא עבר קרוב מאוד לנקודות הלבנות, ועל כן ייתכן שכאשר נדגים בעתיד נקודות לבנות חדשות הן יסווגו בשוגג כשחורות. האם איטרציות אימון נוספות יעוזו? מקטע הקוד שלහן ניתן לראות שלא, וזאת מפני ש' :

1. האיטרציה الأخيرة הורידה את מחיר המודל רק בעשרה אחוז מהערך קודם – שיפור זניח.
2. הגרדיאנט באיטרציה الأخيرة אפסי, ועל כן בתוספת איטרציות נוספות, ערכי הפרמטרים לא ישתנו כמעט.



כדי לקבל תוצאה טובה יותר, היה כדאי לאתחל את הפרמטרים בערכים קרובים יותר למטרה הרצויה, במקורה שלנו למשל, קו בשיפוע יורך. במקרים מסוימים יותר לא יהיה פשוט כל כך למצוא ערכי פרמטרים התחלתיים מוצלחים, ובכך נדונ בהמשך הקורס.

## שאלות לתרגול

- שנו את קבוע הלמידה `alpha` (הגדילו והקטינו אותו) באlgorigthm הניל ובדקו אם מתקבלות תוצאות טובות יותר.
- אתחלו את המודל עם ערכי פרמטרים טובים יותר ובדקו את ביצועיו.
- על סמך ערכי הפרמטרים המתקבלים בסוף רצת האלגוריתם, חשבו את הגרדיאנט  $H \nabla$  עבור נקודה שחורה אחת ונקודה לבנה אחת, ונסו להסביר למה ערכים אלו כה קטנים, אף על פי שמודל הסיווג שהתקבל אינם הטוב ביותר.
- שנו את קוד אימון הנוירון כדלקמן: הציבו את ערכי הנגזרות האנגליטיים של  $H \nabla$  במקום לשימוש בחישובי ביןים ובכל השרשת. השוו את ביצועי שתי הגרסאות. איזו מהן עדיפה לדעתכם? הסבירו מדוע.
- א. כתבו את קוד אימון הנוירון באופן וקטורי.

הנחיות:

- הניחו שהקלט הוא נקודות שחורות ולבנות במרחב  $\mathbb{R}^M$ , ושהחישוב שמבצע הנוירון לחיזוי הסתברות שנקודה נתונה היא שחורה הוא:

$$z = w_0x_0 + w_1x_1 + \dots + w_{k-1}x_{k-1} + b$$

$$y = \frac{1}{1 + e^{-z}}$$

- שימוש לב שבמקרה זה למודל יש  $k+1$  פרמטרים :  $b, w_0, w_1, \dots, w_{k-1}$ . לפיכך, יש לחשב נזורת של פונקציית המחיר עבור כל אחד מהפרמטרים, ולערכן את כולם בכל איטרציה של אלגוריתם האימון.
- ב. בדקו את הקוד שכתבתם בסעיף א באמצעות נקודות שחזורות ולבנות במרחב  $k$ -ממדי (בעזרת שינוי הפרמטר `n` של הפונקציה `make_blobs`) וaiימון הנוירון לסיוגן.

## אימון הנוירון - מימוש עם PyTorch

בפרק הקודם מימשנו את תהליך אימון הנוירון מהיסוד, וכעת נראה כיצד, בעזרת PyTorch, נוכל לעשות זאת באופן פשוט יותר ויעיל יותר מבחינה חישובית, בזמןית. ראשית נגדיר את מודל הנוירון היחיד שלנו בעזרת הספרייה `nn` המכילה את כל אבני הבניין של רשותות נוירוניים אשר נדרש להן.

```
from torch import nn
z = nn.Linear(2, 1)
y = nn.Sigmoid()
print(z)

פלט:
Linear(in_features=2, out_features=1, bias=True)
```

בעת ייצירת האובייקט `z`, הפרמטרים  $b$ ,  $w_0$ ,  $w_1$  אוטחלו באקראי, ולכן נוכל לחשב את תחזית המודל על אחת מהנקודות באוסף הנתונים שלנו, למשל:

```
y(z(X[0, :]))
פלט:
tensor([0.1221], grad_fn=<SigmoidBackward0>)
```

כמו כן, ניתן לגשת לפרמטרים של המודל ישירות, שכן הם מאפיינים של `z`, לבדוק שאכן החישוב המבוצע בקריאה ל`z` כפונקציה הוא  $b + w_0x_0 + w_1x_1$ :

```
print(z.weight, z.bias, sep='\n')
assert(z.weight[0, 0]*X[0, 0]+z.weight[0, 1]*X[0, 1]+z.bias[0] ==
      z(X[0, :]))
פלט:
Parameter containing:
tensor([-0.2000,  0.2818]), requires_grad=True)
Parameter containing:
tensor([-0.0825], requires_grad=True)
```

נוסף על כך, אם ברצוננו לשנות את ערכי הפרמטרים, נוכל לעשות זאת בקלות, אך יש לזכור שמערכת `autograd` פעילה ועוקבת אחרי הפעולות המתבצעות על פרמטרים אלו כדי שבעתיד נוכל לקבל אוטומטית את הנזירות לפיהם. מכיוון ששינויי ידני של הפרמטרים אינם חלק מתהליך האימון (ולמעשה הוא אף אינו פעולה גזירה!), יש להציגם בפורש של פעולה זו להתבצע ללא חישוב נזירות. אחת הדרכים לעשות זאת היא באמצעות שימוש ב`context manager`:

```
with torch.no_grad():
    z.weight[0,0], z.weight[0,1], z.bias[0] = (1, -1, 5.5)
print(z.weight,z.bias,sep='\n')
```

**פלט:**

```
Parameter containing:
tensor([[ 1., -1.]], requires_grad=True)
Parameter containing:
tensor([5.5], requires_grad=True)
```

כל הפעולות אשר מבוצעות בתוך ההקשר של `no_grad()` מבוצעות ללא מעקב מערכת הגזירה האוטומטית.

כעת נריץ את כל הנתונים קדימה בראשת, ונקבל את תחזית המודל עבורם בשורת קוד אחת, וכן גם את פונקציית המחיר שלנו (שימו לב שהוא משתמש כאנו באופרטורים שהורמו לפועל איבר-איבר על טנзорים) :

```
y_model = torch.squeeze(y(z(X)))
CE_loss = -1/len(Y)*torch.sum(Y*torch.log(y_model) +
(1-Y)*torch.log(1-y_model))
print(CE_loss)

tensor(-0.5340, grad_fn=<MulBackward0>)
```

**פלט:**

אחרי שהчисבנו את פונקציית המחיר, נחשב את הגרדיאנט שלו לפי הפרמטרים. כאן בא לידי ביטוי היתרונו הגדול ביותר של PyTorch : כל שיש לעשות הוא להריץ את הפוקודה `()` ולקבל את הנזרות באופן אוטומטי.

```
CE_loss.backward()
print(z.weight.grad, z.bias.grad, sep='\n')

tensor([[2.4333, 1.7574]])
tensor([-0.0710])
```

**פלט:**

נותר רק לעדכן את ערכי הפרמטרים באמצעות חישור הערך  $C^\alpha$  (שוב, ללא מעקב מערכת הגזירה האוטומטית), ולהזور על התחלתיך עד להתקנות. הליבה של לולאת האימון נראה אפוא כך :

```
z.zero_grad()
y_model = torch.squeeze(y(z(X)))
CE_loss = -1/len(Y)*torch.sum(Y*torch.log(y_model)+(1-
Y)*torch.log(1-y_model))
CE_loss.backward()
with torch.no_grad():
    z.weight -= alpha*z.weight.grad
    z.bias   -= alpha*z.bias.grad
```

זכרו שלפני כל איטרציה יש לאפס את הגרדיאנט, שכן בירית המclid של המתודה () backward היא צבירת הערכים המוחשבים זה עתה באיטרציה לאלו הקיימים כבר במאפייני grad של המשתנים.

## שאלה לתרגול

- כתבו את לולאת האימון, הריצו אותה והשו את התוצאות למימוש הקודם.
- האם מתקבלות תוצאות זהות?
  - איזה קוד מהיר יותר? בדקו בעורת פקודת `timeit Magic`.

## אוסף הנתונים

אוסף הנתונים של MNIST, המכיל 70,000 תמונות של הספרות 0–9 בכתב יד, הוא אחד המפורסמים ביותר בתחום למדינת המכונה, ומשמש לרוב בתורת הדוגמה הראשונה שמבצעים עליה אלגוריתם לסיווג: הקלט הוא אחת התמונות והפלט הדרוש הוא זיהוי ספרה המופיע בתמונה. למרות הפופולריות שלו, כיום אוסף זה הוא אתגר קל מדי – אפיו האלגוריתמים הפושטים ביותר מצלחים לסווג את הספרות נכונה בדיקת הגובה מ-95%. לפיכך, אנו נשתמש באוסף הנתונים Fashion-MNIST שלו מאפיינים דומים – 70,000 תמונות מחולקות ל-10 מחלקות שונות, כאשר מזובר **פריטי לבוש** במקום ספרות. היתרונו בשימוש באוסף זה הוא שסיווג פריטי הלבוש הוא משימה מוגדרת יותר, אשר מאפשר לנו לבחון את השיפור בביצועי רשת הנוירונים ככל שנוטס לה רכיבים מתקדמים יותר במהלך הלימוד.

לפני בניית הרשת אוימונה, נרצה לייבא את אוסף הנתונים ולטעון אותו לזכרוו המחשב. ניתן לעשות זאת ישירות בעזרת הספרייה PyTorch המכילה ממשק ליבוא אוסף נתונים נפוצים. ב��ע הקוד שלහן אנו מורידים את אוסף הנתונים וטוענים אותו לתוך המשטנה `mnist_train`

```
import torch
import torchvision
mnist_train = torchvision.datasets.FashionMNIST(
    root="/22961", train=True, download=True)
```

לאחר ריצה מוצלחת של הקוד, אוסף הנתונים ישמר באופן מקומי בתיקייה הנקובה בParmeter `root`, ויתקבל אובייקט דאטא לתוך המשטנה `mnist_train`, שבו נחקרו עתה.

```
print(len(mnist_train))
mnist_train.classes
```

**פלט:**

```
60000
['T-shirt/top',
 'Trouser',
 'Pullover',
 'Dress',
 'Coat',
 'Sandal',
 'Shirt',
 'Sneaker',
 'Bag',
 'Ankle boot']
```

הפקודה הראשונה מדפסה את אורכו של אוסף הנתונים, 60,000 פריטים. זהו סט האימון (training set), כאשר עשרות אלפי תמונות נוספות שמורות לבדיקת האלגוריתמים – סט הבדיקה (test set). אפשר להוריד גם את התמונות האלו, באמצעות שינוי הParmeter `train` של פונקציית הייבוא

ל-`False`. כמו כן, במאפיין `classes` אנו רואים את שמות המחלקות, וכן ניכר שמדובר בפרייתי לבוש.

כעת נטען את אחת התמונות לזכורן :

```
A = mnist_train[0]
print(type(A), len(A), A[0],
      A[1], mnist_train.classes[A[1]], sep='\n')
平淡:
<class 'tuple'>
2
<PIL.Image.Image image mode=L size=28x28 at 0x7EFEE80C8CD0>
9
Ankle boot
```

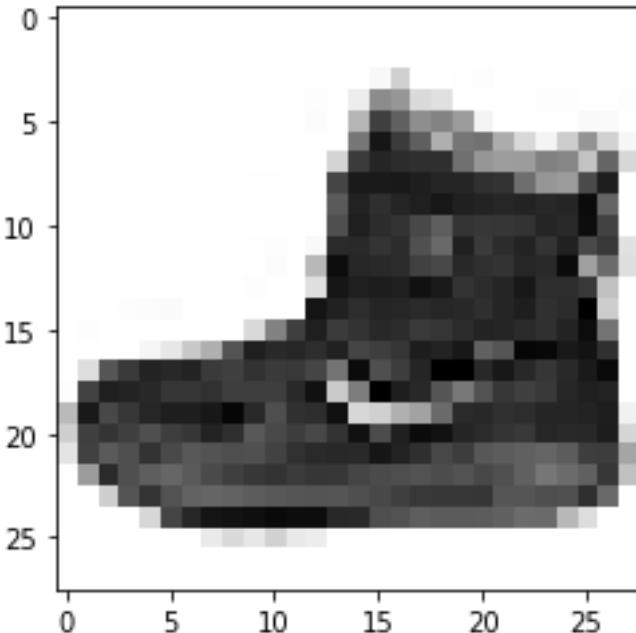
ניתן לראות שהנתונים נתונים במבנה של זוג סזר, כאשר האיבר הראשון הוא תמונה בפורמט PIL והשני הוא סיווג התמונה. נודא זאת, אך תחילה נמיר את התמונה לטנזור, כדי שנוכל להמשיך בעבודה בכלים המוכרים לנו.

```
convert = torchvision.transforms.PILToTensor()
img = convert(A[0])
print(type(img), img.size(), sep='\n')
平淡:
<class 'torch.Tensor'>
torch.Size([1, 28, 28])
```

נציר את התמונה בעזרת הפקנץיה `imshow` של `matplotlib`, אך ראשית נשים לב שפונקציה זו מקבלת כקלט מטריצה (טנזור דו-ממדי) ועל כן علينا להעלים את הממד הראשון המנוון בטנזור שלנו באמצעות הפקודה `squeeze`.

```
import matplotlib.pyplot as plt
plt.imshow(torch.squeeze(img), cmap='Greys');
```

פלט:



ואכן, נראה שמדובר ב מגן, המתאים לSieving הנתון למחלקה מס' 9.

בשלב הבא, לצורך ייעול תהליכי האימון, נרצה לטענו את הנתונים במנוגות (minibatches), למשל 64 תמונה יחיד בכל פעם, וכן לעשות זאת בריזומניות על רכיבי חומרה נפרדים, כגון מאיצים גרפיים או (Tensor Processing Unit) TPU – רכיבים ייעודיים לעיבוד טנזורי. לצורך זה השתמש בכלל PyTorch DataLoader של Dataset אובייקט Dataset קבלן PyTorch, שעובדנו אותו עד כה.

לאחר ריצת הקוד שלහן נוכל להשתמש בـ train\_dataloader כדי לעבור על אוסף הנתונים ביעילות. שימושו לב שבעור השימוש בטוען הנתונים יש לשלב את המרת התמונות מ-PIL לטנזור כבר בשלב ייצור ה- Dataset .

```
from torch.utils.data import DataLoader
train_data_transformed = torchvision.datasets.FashionMNIST(
    root="/22961", train=True, download=False,
    transform=torchvision.transforms.PILToTensor())

train_dataloader = DataLoader(
    train_data_transformed, batch_size=4)
```

בעת נוכל לקבל איטרטור מאובייקט זה, אשר יירוץ על אוסף נתונים האימון מתחילה ועד סוף, ובכל קרייה יחזיר minibatch של ארבע תמונות.



```

iterator = iter(train_dataloader)
imgs, labels = next(iterator)
class_names = train_data_transformed.classes
fig = plt.figure()
for i in range(4):
    ax = fig.add_subplot(2, 2, i+1)
    plt.imshow(torch.squeeze(imgs[i]), cmap='Greys')
    ax.set_title(class_names[labels[i]])
    ax.axes.get_xaxis().set_visible(False)
    ax.axes.get_yaxis().set_visible(False)

```

פלט:

Ankle boot



T-shirt/top

Dress

בדוגמה זו אנו מכירים לראשונה לעומק את דרך הפעולה של החבילת Matplotlib, ועל כן נסביר בפירוט את הפקודות והלוגיקה:

1. הפקודה `plt.figure()` יוצרת אובייקט תמונה אשר יכול את מה שנזכיר אחר כך.
2. הפקודה `(x, add_subplot(2, 2, i+1))` מחלקת את התמונה לשתי שורות ושתי עמודות, ונותנת לנו גישה לציר באחד הריבועים שנוצרו. למשל כאשר `i=1` נציג לתוכה הריבוע השמאלי העליון, ובאשר `i=3` נציג לתוכה הריבוע השמאלי התחתון.
3. הפקודות `ax.set_title`, `ax.set_xaxis`, `ax.set_yaxis` משנות את המאפיינים של מערכת הצירים הנוכחיית (אחד הריבועים הנ"ל).

ישנו מגוון רחב של מאפייני מערכת צירים אשר ניתן לשנות בהם בצורה דומה. כל שינוי כזה משפיע במעט על הנראות של התמונה. אין צורך לזכור אותם בעל פה, אלא כרגע לחשוף את הרצוי בתיעוד של `.matplotlib`.

לרוב נטען את ה-`minibatches` בלולאה ונעביר אותם למודל כך:

```
for imgs, labels in train_dataloader:  
    .  
    .  
    .
```

לולאה זו תעבור על אוסף הנתונים במלואו פעם אחת, וכך כדי ייעול תהליכי טעינת הנתונים לזכורם המובנה בתוך האובייקט.

לעתים יהיה שימושי לדעת גם את אינדקס ה-`batch` אשר נטען זה עתה לזכורו, ולשם כך נשתמש בפקודה `enumerate`, כלהלן:

```
enumerator = enumerate(train_dataloader)  
for batch_idx, (imgs, labels) in enumerator:  
    .  
    .  
    .
```

## שאלה לתרגול

בננו בעצמכם אובייקט `Dataset` אשר יוכל לאוסף הנתונים Fashion-MNIST וטענו ממנה תמונות בעזרת `DataLoader`.

**הנחיות:**

- א. הורידו 500 תמונות מאוסף הנתונים Fashion-MNIST מהקישור באתר הקורס.
- ב. כתבו מחלוקת בשם `FashionMNISTDataset` הירושת מהמחלקה של Pytorch `Dataset` של עלייכם למש 3 מетодות :

- `__init__(self, labels_file, dir, transform=None, target_transform=None)`  
מתודזה זו מתחילה את האובייקט, והקלט שלו הוא מיקום קובץ הסיווגים של כל התמונה ומיקום התיקייה שההתמונות מופיעות בה.
- `__len__(self)`  
מתודזה זו מחזירה את מספר הדגימות באוסף הנתונים.
- `__getitem__(self, idx)`  
מתודזה זומחזירה את הדגימה ה-`idx` באוסף הנתונים. הפלט שלה הוא מהצורה:

```
return image, label
```

- ג. אתחלו אובייקט מהמחלקה זו המצביע על התיקייה שבה נמצא אוסף הנתונים.
- ד. צרו אובייקט `DataLoader` העוטף את האובייקט הקודם.
- ה. טענו נתונים בעזרת `DataLoader` וציירו אותם בעזרת הפקודת `.plt.imshow()`

**הערה:** מומלץ להיעזר בתיעוד של PyTorch לפתרון שאלה זו. קישור מופיע באתר הקורס.

## הגדרת המודל ופונקציית המחיר

### הגדרת המודל

עלינו לתכנן רשות המסוגלת לסווג את התמונות הנתונות לעשר המחלקות השונות של פריטי הלבוש, ולצורך כך נרחיב את מודל הנוירון היחיד משני צידיו: ראשית, הקלט שלנוCut הוא טנзор בגודל 28X28, על כן **נשתח** אותו: נמיר אותו לטנзор חד-ממדי בגודל 784 באמצעות שרשרת השורות זו אחריו זו. ראו להלן הדוגמה לשיטתה טנзор:

```
A = torch.arange(2*5).reshape(2,5)
print(A, A.size())
A = A.flatten()
print(A, A.size())
tensor([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]]) torch.Size([2, 5])
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) torch.Size([10])
```

**פלט:**

طنзор המשותח ישמש כקלט החדש לרשת.

שנית, הנתונים מתחולקים לעשר מחלקות ולפייך יהיו במודל עשרה נוירוני פלט – אחד לכל מחלקה. את הפונקציה הלוגיסטיבית תחליף המקבילה הריב-מדנית שלה, הפונקציה softmax :

$$\text{softmax}(z_0, z_1, \dots, z_k) = \frac{1}{\sum_{n=0}^k e^{z_n}} \begin{pmatrix} e^{z_0} \\ e^{z_1} \\ \vdots \\ e^{z_k} \end{pmatrix}$$

שימוש לב שאיברי וקטורי הפלט הם מספרים חיוביים אשר סכומם אחת, ולכן הפלט הוא וקטור הסתברויות. פונקציה זו קרובה כך על שום התוכונה הבאה: אם אחד מהערכים בווקטור הקלט ( $z_0, z_1, \dots, z_m$ ) גדול מהאחרים, למשל  $z_1 > z_m$  לאחר  $m$  אחר, אז פעולה האקספוננט לרוב תק岑 פער זה, ויתקבל  $e^{z_1} \gg e^{z_m}$ . לאחר הנרמול, הפלט יהיה קרוב ל-  $(0, 1, 0, \dots, 0)$ : 1 היכן שההערך המקסימלי בקלט, ו-0 בשאר הערכים. זהו למעשה ה-argmax של הקלט. ככל ש- $z_1$  גדול יותר מאשר ערכי הקלט, כך הפלט יהיה קרוב לווקטור  $(0, 1, 0, \dots, 0)$ . תוצאה זו, יחד עם העובדה שהsoftmax היא פונקציה גזירה, מסבירה את נוכחות המילה soft בשםה.

כעת יש בידינו כל הדרוש להגדרת מודל הסיווג לרשותנו נוירונים :

1. הרשות תקבל כקלט טנзор חד-מדי,  $(x_0, x_1, \dots, x_{783})$ .
2. על הקלט יופעלו עשר פונקציות ליניאריות שונות, אחת לכל מחלקה :

$$z_0 = w_{0,0}x_0 + w_{0,1}x_1 + \dots + w_{0,783}x_{783} + b_0$$

$$z_1 = w_{1,0}x_0 + w_{1,1}x_1 + \dots + w_{1,783}x_{783} + b_1$$

:

$$z_9 = w_{9,0}x_0 + w_{9,1}x_1 + \dots + w_{9,783}x_{783} + b_9$$

או בכתיבה וקטורי,

$$Z = W \cdot X + b$$

כאשר

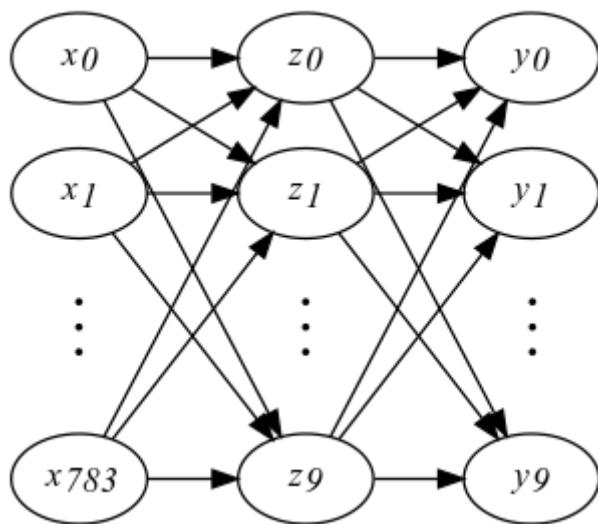
$$Z = \begin{pmatrix} z_0 \\ \vdots \\ z_9 \end{pmatrix}, X = \begin{pmatrix} x_0 \\ \vdots \\ x_{783} \end{pmatrix}, b = \begin{pmatrix} b_0 \\ \vdots \\ b_9 \end{pmatrix}, W = \begin{pmatrix} w_{0,0} & \cdots & w_{0,783} \\ \vdots & \ddots & \vdots \\ w_{9,0} & \cdots & w_{9,783} \end{pmatrix}$$

3. תוצאת חישוב זה תועבר לפונקציה softmax,

$$Y = \begin{pmatrix} y_0 \\ \vdots \\ y_9 \end{pmatrix} = \text{softmax}(Z) = \frac{1}{\sum_{n=0}^9 e^{z_n}} \begin{pmatrix} e^{z_0} \\ \vdots \\ e^{z_9} \end{pmatrix}$$

4. פלט הרשת,  $Y$ , הוא וקטור אשר כל אייר בו הוא הסתברות שהקלט הנתון שייך למחלקה המתאימה :  $y_k$  היא הסתברות השיכנות למחלקה ה- $k$ ית.

ובאיור סכמטי,



שימוש לב שפרמטרים של המודל הם המטריצה  $W$  והווקטור  $b$  אשר קובעים את **פונקציית האgregציה** (aggregation function) הלינארית ( $Z = W \cdot X + b$ )

## פונקציית המחיר

בעבר עסקנו בסיווג נקודות שחורות ולבנות ובחנו בפונקציית מחיר אשר "קונסט" מודל שמסווג בchroma שגואה נקודה שחורה נתונה באופן פרופורצional ל- $y(\log)$ , כאשר  $y$  הייתה ההסתברות שהנקודה שחורה, על פי תחזית המודל. בדומה, הכנס על כל נקודה לבנה פרופורצional ל- $(1 - \log(y))$ , ונשים לב ש- $y - 1$  הוא למעשה הастברות החזואה שהנקודה היא לבנה. עתה נכליל רעיון זה למקורה הרב-ממדי. ראשית, לכל נקודה דגימה נתונה,  $X = (x_0, x_1, \dots, x_{783})$ , נתון גם הסיווג לאחת המחלקות 0–9. נקודד סיווג זה בצורה one-hot ונקבל וקטור  $Y_t = (y_{t,0}, y_{t,1}, \dots, y_{t,9})$  אשר רק אחד מאיבריו הוא 1 ושאר האיברים הם 0:  $y_{tk} = 1$  אם ורק אם סיווג הנקודה הנתונה הוא למחלקה ה- $k$ -ית. ראו למשל,

```
imgs, labels = next(iter(train_dataloader))
print(labels)
torch.nn.functional.one_hot(labels, num_classes=10)

tensor([0, 3, 1, 9])
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
```

**פלט:**

כעת נגדיר את התרומהמחיר של כל דגימה כך,

$$H(X, Y_t) = -\sum_{n=0}^9 y_{tn} \log(y_n)$$

כאשר הווקטור  $Y = (y_0, \dots, y_9)$  הוא פلت הרשות עבור הקלט  $X$ . לפי הגדרת  $Y$ , כל האיברים בביטוי זה, למעט אחד, מתאפסים. נקבל

$$H(X, Y_t) = -\log(y_k)$$

כאשר  $k$  היא המחלקה שאליה נקודת דגימה זו שייכת, בדיק כמו במקרה הדווימדי. כדי לחשב את פונקציית המחיר הכללית, יותר רק לחשב ממוצע של פונקציה זו על כל נקודות הדגימה. התוצאה המתתקבלת היא **האנטרופיה הצולבת** עבור בעיית סיווג במספר רב של מחלקות:

$$C(W, b) = \frac{1}{\#Data} \sum_{(X, Y_t) \in Data} H(X, Y_t) = -\frac{1}{\#Data} \sum_{(X, Y_t) \in Data} \sum_{n=0}^9 y_{tn} \log(y_n)$$

כאן יש לשים לב שפונקציית המחיר תלואה בפרמטרים  $W, b$  דרך הסתברויות הסיווג  $(y_0, \dots, y_9)$ , אשר בתורן תלויות בפרמטרים דרך הארגזציה  $Z$ , ולכן כל השרשרת שוב יהיה שימושי לצורכי אימונו. המודל בעזרת אלגוריתם מورد הגרדיינט.

## שאלות לתרגול

1. מנו כמה פרמטרים בדיקק קיימים במודל הסיווג הניל.

$$2. \text{ חשבו את } \nabla H = \left( \frac{\partial H}{\partial W}, \frac{\partial H}{\partial b} \right).$$

**הנחיות:**

$$\text{א. עבור אחד מנירוני הפלט, חשבו את } \frac{\partial H}{\partial y_k}.$$

$$\text{ב. עבור אותו נירון, חשבו את הנגזרת לפי אחד מרכיבי הארגזיה, } \frac{\partial y_k}{\partial z_m}.$$

$$\text{ג. עבור רכיב ארגזיה זה חשבו את הנגזרת לפי אחד מהפרמטרים, } \frac{\partial z_m}{\partial w_{p,q}}. \text{ הפרידו את החישוב}$$

למקרים  $m = p$  או  $m \neq p$ .

ד. חקרו את כל התוצאות בעזרת כלל השרשרת. הייזרו באյור הרשות המופיע לעיל.

$$3. \text{ האם תוכלו לכתוב ביטוי פשוט עבור } \frac{\partial Z}{\partial b} ? \text{ ועבור } \frac{\partial Z}{\partial W} ?$$

4. הגדרו את מודל הרשות בשתי שורות קוד. רמז: הייזרו בספרייה `nn`.

5. הזינו לתוך הרשות שהגדרתם בשאלת הקודמת את סט האימון של אוסף הנתונים Fashion-MNIST במלואו וחקרו את פונקציית המחיר על התוצאה.

**הנחיות:**

- כבר בשלב טיענת הנתונים, קודם להזנת הנתונים, המירו אותם לפורת נתונים מתאים, `float`

**בעזרת הפונקציה**

`.torchvision.transforms.ConvertImageDtype`

- אחרי כן שטחו את התמונות בעזרת המתודה `()`.

## אלגוריתם האימון – מورد הגרדיאנט האקראי

בסוף הפרק הקודם הגדרנו את פונקציית המחיר של מודל הסיווג ל-10 מחולקות פריטי הלבוש כך:

$$C(W, b) = \frac{1}{\#Data} \sum_{(X, Y_t) \in Data} H(X, Y_t)$$

$$H(X, Y_t) = -\sum_{n=0}^9 y_m \log(y_n)$$

אם ברצוננו להשתמש באלגוריתם מورد הגרדיאנט כדי לאמן את הרשות, علينا לחשב **בכל איטרציה** את הגרדיאנט

$$\nabla C(W, b) = \frac{1}{\#Data} \sum_{(X, Y_t) \in Data} \nabla H(X, Y_t)$$

שימוש לב Ci בגרדיאנט זה אנו מוחשבים את הנגזרת של פונקציית המחיר לפי כל אחד מהפרמטרים של המודל, וכן שסכום זה רץ על כל דוגמאות פריטי הלבוש בסט האימון, שמספרן הוא 60,000. כבר עתה, כשמדבר במודל פשוט ביותר למשימת הסיווג, חישוב זה יוצר צוואר בקבוק בתהיליך האימון. בהמשך הלימוד, כשהנעסק במשימות ורשותות מורכבות יותר, בעלות אוסף נתונים גדול יותר, חישוב זה יעשה בלתי אפשרי. מעבר לכך, קיימת **יתירות** רבה באוסף הנתונים: ישנן דוגמאות דומות מאוד (למשל זוג מגפיים דומים), אשר הגרדיאנט  $\nabla H$  שלהם דומה אף הוא, ובעת חישוב  $\nabla C$  כנ"ל אנו חוזרים על חישובים זהים שלא לצורך וمبזבזים משאבים.

כדי להתמודד עם בעיות אלו, נבצע התאמות באלגוריתם מورد הגרדיאנט הקלסי. הראשונה שבוחן היא שימוש **בגרדיאנט האקראי** (stochastic gradient): במקומות לחשב את הממוצע של  $H$  על כל אוסף הנתונים, נדגום מהאוסף הגדול קבוצה קטנה (minibatch) אקראית, ונחשב ממוצע זה רק על איבריה:

$$\tilde{\nabla} C(W, b) = \frac{1}{\#minibatch} \sum_{(X, Y_t) \in minibatch} \nabla H(X, Y_t)$$

אם ה- $\tilde{\nabla} C$  אשר דגמוני נאמנה את אוסף הנתונים, אז לא נסידר הרבה במעבר לשימוש בקיורוב, שכן  $\tilde{\nabla} C \approx \nabla C$ . לעומת זאת, אלגוריתם מورد הגרדיאנט האקראי (SGD – Stochastic Gradient Descent) זהה לאלגוריתם המקורי: מעדכנים את ערכי הפרמטרים בכיוון השיליי של הגרדיאנט האקראי,

$$(W, b) = (W, b) - \alpha \tilde{\nabla} C$$

וחזורים על החישוב באופן איטרטיבי.

הrndומליות המובנית בתחום אלגוריתם זה אינה בהכרח דבר רע, שכן לפונקציות המחיר שנבעוד איתן בהמשך יהיה מספר רב של נקודות מינימום מקומי. דבר זה עלול להכשיל את אלגוריתם מورد הגרדיאנט הקלסי: הוא יעזור בכל אחת מהן, אם יגיע אליהן, שכן שם הגרדיאנט מטאפס. אם לעיתים האלגוריתם יעשה צעד שאיןו בכיוון הירידה הטלולה ביותר, תיווצר ההזדמנויות לבורוח ממינימום מקומי ולנוע אחר כך אל ערכי פרמטרים טובים יותר. למעשה קיימים אלגוריתמי אופטימיזציה אשר משלבים אלמנטים סטוכסטיים בכוונה תחיליה, בדיקות למטרה זו.

פרקטיית, כדי להשתמש במלוא הנתונים בסט האימון, אנו דוגמים **לא החזרה** עד אשר כל הסט הגיעו למיizio. במלילים אחרים, נחלק את **כל** סט האימון ל-**minibatches** אקרים, ועל סמך אלו

נחשב איטרציות של SGD. לאחר השימוש בccoli – נחלק שוב את אוסף הנתונים באקראי (ובשונה מהחלוקת הקודמת) לקבוצות קטנות ונחוור על התהיליך. מעבר אחד על כל הנתונים בסט האימון (על כל ה-*minibatches*) נקרא epoch. בהנחה שזמן המעבר על כל סט נתוני האימון קבוע, חלוקתו של minibatches קטנים תאפשר לנו לבצע יותר צעדים בepoch יחיד, באופןו מחריר חישובי, דבר אשר יוביל לרוב להתקנסות מהירה יותר.

## שאלות לתרגול

1. טענו לזכרון minibatch של 16 דוגמאות מסט האימון של אוסף הנתונים של Fashion-MNIST.
2. א. בצעו איטרציה אחת של SGD על הרשות שהגדרתם בפרק הקודם.

**הערות:**

- השתמשו במתודה () backward לחישוב הגרדיאנט.
  - הפרמטרים של המודל נמצאים במאפיינים *z.weight* ו- *z.bias*.
  - הגרדיאנט המוחושב נמצא במאפיין *grad* של הפרמטרים.
- ב. בצעו איטרציה אחת של מורד הגרדיאנט (הגרדיאנט המלא).
  - ג. השוו את זמני הריצה של איטרציה אחת בשני האלגוריתמים.

3. בדומה לבועית הסיווג הבינרי, פונקציית הנראות המתאימה למודל הסיווג ל-10 מחלקות היא

$$L(W,b) = \exp(-\#Data \cdot C(W,b)) = \prod_{(X,Y_i) \in Data} e^{-H(X,Y_i)} = \prod_{(X,Y_i) \in Data} \prod_{n=0}^9 y_n^{y_m}$$

וערכי הפרמטרים שמתתקבל בהם מינימום של  $C(W,b)$  הם אלו אשר מתתקבל בהם מקסימום הנראות.

א. חשבו את  $\frac{\partial L}{\partial w_{p,q}}$  עבור  $w_{p,q}$ , אחד הפרמטרים במודל, והסבירו בעזרת הביטוי המתתקבל מדוע.

עדיף לחפש את המינימום של  $C(W,b)$  על פני המקסימום של  $L(W,b)$ .

רמז: בפרק הקודם חישבתם את  $\frac{\partial H}{\partial w_{p,q}}$ . השתמשו בחישוב זה ובכלל השרשרת.

ב. לעיל טענו שעבור הגרדיאנט האקראי, המוחושב על בסיס minibatch, מתקיים  $\tilde{\nabla}C \approx \nabla C$ .  
הסבירו למה קשה יותר לחשב קיורוב ל- $\nabla C$  על בסיס minibatch קטן.  
רמז: השתמשו בתשובה לשיער הקודם.

## אימון הרשות

בשלב זה נפנה למימוש הרשות ואמוננה לפי אלגוריתם מورد הגדריאנט האקראי, תוך כדי שימוש במלוא יכולותה של הספרייה PyTorch. ראשית נגדיר את המודל,

```
from torch import nn
model = nn.Sequential(
    nn.Linear(784, 10),      # z
    nn.LogSoftmax(dim=1)     # log(y)
)
print(model)
```

**פלט:**

```
Sequential(
  (0): Linear(in_features=784, out_features=10, bias=True)
  (1): LogSoftmax(dim=1)
)
```

ונשים לב שבמוקום הפונקציה softmax בפלט הרשות אנו מחשבים את הלוגריתם שלה. הסיבה לכך היא שבתהליכי האימון משתמש בפונקציית המחיר של האנטרופיה הצולבת, שבעוראה עליינו לחשב הלוגריתם של הסתברויות הסיווג למחלקות,  $(y_0, \dots, y_9) = Y$ . חישוב אנליטי של הלוגריתם נותן תוצאה מדויקת יותר, שכן:

$$\log(\text{softmax}(z_0, z_1, \dots, z_k)) = \log\left(\frac{1}{\sum_{n=0}^k e^{z_n}} \begin{pmatrix} e^{z_0} \\ e^{z_1} \\ \vdots \\ e^{z_k} \end{pmatrix}\right) = \begin{pmatrix} \log(e^{z_0}) - \log\left(\sum_{n=0}^k e^{z_n}\right) \\ \log(e^{z_1}) - \log\left(\sum_{n=0}^k e^{z_n}\right) \\ \vdots \\ \log(e^{z_k}) - \log\left(\sum_{n=0}^k e^{z_n}\right) \end{pmatrix} = \begin{pmatrix} z_0 - \log\left(\sum_{n=0}^k e^{z_n}\right) \\ z_1 - \log\left(\sum_{n=0}^k e^{z_n}\right) \\ \vdots \\ z_k - \log\left(\sum_{n=0}^k e^{z_n}\right) \end{pmatrix}$$

מעבר לחישכון פועלות האקספוננט והפיכתה בעזרת הלוגריתם, בחישוב זה אנו מתחמקים מביעיות יציבות נומריות אשר עלולות להיווצר כאשר אחד או יותר מאיברי הפלט של הפונקציה softmax מתאפס. מלבד זאת, בשינוי זה אין לנו מפסידים דבר, שכן כל אימת שנרצה את פלט הרשות  $Y$ , יוכל להפעיל את האקספוננט ולקבלו.

בשלב הבא נגדיר את פונקציית המחיר, אשר כזכור עבור דוגמה יחידה מסט האימון היא  $H = -\log(y_n)$ , כאשר  $n$  היא המחלקה שהדוגמה מסוגת אליה. נמשז זאת בעזרת הפונקציה nn.NLLLoss: פונקציה זו מצפה לקבל כקלט את לוגריתם הסתברויות הסיווג של המודל, ומחזירה את מינוס הלוגריתם של פונקציית הנראות. בהתאם, פירוש ראשי התיבות בשם הוא Negative Log Likelihood Loss. הפעלה מיד לאחר LogSoftmax מניבה את האנטרופיה הצולבת, שבה אנו מעוניינים להשתמש.

כל שנותר לעשות הוא להגדיר אובייקט אופטימיזציה, המקבל כקלט את פרמטרי המודל, ולהשתמש בMETHODS המובנות שלו כדי לעדכן את הפרמטרים.

```
CE_loss = nn.NLLLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

בעזרת אובייקטים אלו, נוכל לכתוב את לולאת האימון בתמציתיות. שימוש לבבקע הקוד הבא שבכל איטרציה של לולאת האימון יבוצעו הפעולות שלහן, לפי הסדר:

1. טעינת minibatch של תמונות פריטי לבוש.
2. איפוס הגרדיינט של הפרמטרים (שכן כברירת המחדל הגרדיינט נצבר).
3. הזנת התמונות לתוך הרשות וчисוב הסטברויות חזויות.
4. חישוב פונקציית המחיר על minibatch זה.
5. חישוב הגרדיינט האקראי בעזרת המתוודה `.backward()`.
6. עדכון ערכי הפרמטרים לפי הנוסחה  $(W, b) = (W, b) - 0.1 \tilde{C}$  בעזרת המתוודה `.step()`. נוסחת עדכון זו נקבעה כאשר הגדרנו את אובייקט האופטימיזציה.

```
def iterate_batch(imgs, labels):
    imgs = imgs.flatten(start_dim=1)
    optimizer.zero_grad()
    y_model = model(imgs)

    loss = CE_loss(y_model, labels)
    loss.backward()
    optimizer.step()
```

נותר לנו רק למדוד את ביצוע האלגוריתם לאורך האימון. המدد הבסיסי ביותר שנרצה למדוד הוא מידת דיקוק הסיווג של המודל (accuracy) – **שיעור הדגימות שהמודול מסוווג נכונה**. ראשית, נמיר את וקטור ההסתברויות המתתקבל מהמודול לסיוג חד-משמעי: נניח שהמודול מסוווג כל דוגמה נתונה אל המחלקה בעלת ההסתברות הגבוהה ביותר בפלט שלה,  $Y = \text{arg max } Y$ . לשם כך יש להוסיף את שורות הנקודות שלහן לאיטרציית האימון.

```
def iterate_batch(imgs, labels):
    .
    .
    .
    predicted_labels = y_model.argmax(dim=1)
    acc = (predicted_labels == labels).sum() / len(labels)
    return loss.detach(), acc.detach()
```

מכיוון שברצוננו להשתמש במדדי הביצועים לצור גרפים בהמשך, ומכיון שאין טעם (ואף אין תמייקה בכך) לעקוב אחרי הפעולות אשר מבצעת הספרייה matplotlib בעזרת מערכת ה-`autograd`, ננטק אותן ממערכת זו לפני החזרתם בעזרת המתוודה `.detach()`.

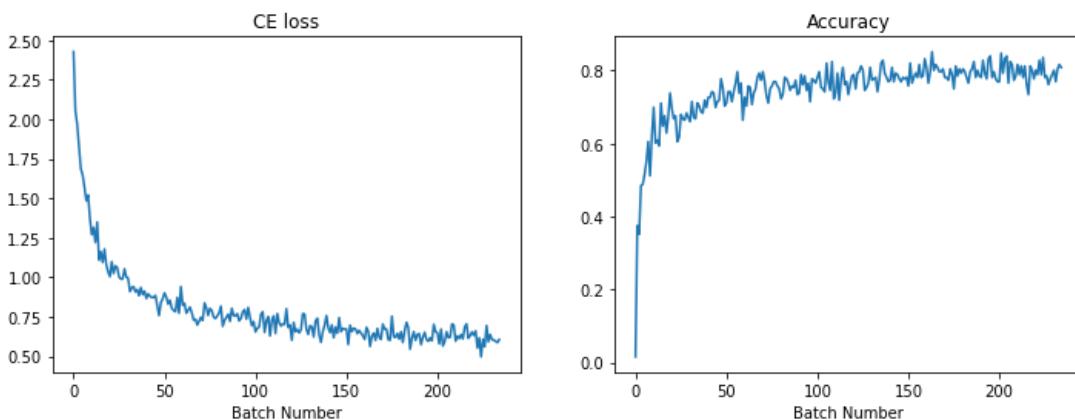
עתה נחלק את הנתונים `batches` בגודל 256 דוגמאות ונבצע אימון לאורך epoch ייחיד, כלומר מעבר ייחיד על כל אוסף הנתונים, תוך כדי טעינת ה-`batches` לזיכרונו בסדר אקראי.

```

train_dataloader = DataLoader(
    train_data_transformed, batch_size=256, shuffle=True)
batches = len(train_dataloader)
    = torch.zeros(batches) loss
    = torch.zeros(batches) acc
for idx, (imgs, labels) in enumerate(train_dataloader):
    loss[idx], acc[idx] = iterate_batch(imgs, labels)

```

באיור המצורף בהמשך מוצגות פונקציית ההפסד ומידת הדיווק, וניתן לראות בו שהרשת אכן למדה לסוג את הנתונים, עד כדי דיוק של 0.8. כמו כן, אפשר לבדוק בתנודות האקרראיות במדדים, שכן בכל איטרציה הם מחושבים על פני *batch* יחיד, אשר אינם מייצג בדיקות מושלmas את אוסף הנתונים כולו.



בשלב הבא יש לחזור על פעולה זו פעמיים אחדות, ולבחו את ביצועי המודל על פני נתונים שלא השתמשנו בהם לאימון. כדי לחסוך בפעולות ולקצר את זמן הריצה, בדיקה זו תבוצע בסוף כל epoch בלבד.

## שאלות לתרגול

1. הסבירו מהי הבעה הנומרטי אשר עלולה להיווצר בחישוב האנטרופיה הצלבתית כאשר

$$\frac{e^{z_p}}{\sum_{n=0}^k e^{z_n}} \text{ עבור } p \text{ בלבד.}$$

2. החליפו את פונקציית המחיר של המודל למידת הדיווק (accuracy), ונסו לאמן את הרשת מחדש. היכן נכשל התהליך? הסבירו באופן תיאורתי מדוע מידת הדיווק לא מתאימה כפונקציית מחיר לאלגוריתם מבוסס גרדיאנט.

3. כתבו את לולאת האימון המלאה, לארך epochs, וכן הוסיפו לה את בדיקת ביצועי המודל בסוף כל epoch על פני כל סט הבדיקה.

**תזכורת:** ניתן לטעון את סט הבדיקה בדרך דומה לטעינת סט האימון, בשינוי הפרמטר `train` של פונקציית טעינת הנתונים ל-`False`.

4. חזרו על האימון עם `minibatches` בגודלים: 1, 4, 16, 512, והשו את התוצאות. באיזה גודל `batch` העדיפו להשתמש?

## רשותות عمוקות

עד כה למדנו את יסודות תהליכי האימון של רשותות נוירונים عمוקות, אך למעשה הפענו אותו על רשות "שטוחה" בלבד, כלומר על רשות בעלת שכבת נוירונים אחת. לאחר האימון ראננו שרתת מסוג זה מסוגלת לסוג את פריטי הלבוש של אוסף הנתונים Fashion-MNIST בדיק של כ-80%. אם נרצה להציג לדיק גובה יותר, علينا לבחור במודל מרכיב יותר – **רשת عمוקה**. רשות כזו מורכבות משכבות של נוירונים המחברות **לנוירונים נוספים**, וرك אחרי כמה שכבות יועבר הפלט לפונקציה softmax למטרת חישוב הסתברויות. כאן בא לידי ביטוי כוחה של הספרייה PyTorch : כדי לאמן רשות عمוקה, علينا לשנות את הגדרת המודל בלבד, בעוד kod הדרוש לאימון הרשות נשאר ללא שינוי. לדוגמה, בקטע הקוד המצורף אנו מגדרים רשות בעלת שתי שכבות.

```
model = nn.Sequential(
    nn.Linear(784, 20),
    nn.ReLU(),
    nn.Linear(20, 10),
    nn.LogSoftmax(dim=1)
)
```

чисוב הסתברויות החזויי ברשות זו מתבצע כדלקמן :  
1. **תמונה פריטי הלבוש** (לאחר שעברו שיטוח) מועברות לשכבה הראשונה, שבה מופעלת פונקציה **ליינארית על הקטל** :

$$Z^{(1)} = \begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ \vdots \\ z_{19}^{(1)} \end{pmatrix} = \begin{pmatrix} w_{0,0}^{(1)}x_0 + w_{0,1}^{(1)}x_1 + \dots + w_{0,783}^{(1)}x_{783} + b_0^{(1)} \\ w_{1,0}^{(1)}x_0 + w_{1,1}^{(1)}x_1 + \dots + w_{1,783}^{(1)}x_{783} + b_1^{(1)} \\ \vdots \\ z_{19}^{(1)} = w_{19,0}^{(1)}x_0 + w_{19,1}^{(1)}x_1 + \dots + w_{19,783}^{(1)}x_{783} + b_{19}^{(1)} \end{pmatrix}$$

שנית, על הפונקציה הלינארית מופעלת אקטיבציה לא ליינארית מסוג **ReLU** :

$$\text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

שמננה מתקיים הפלט של השכבה הראשונה,

$$Y^{(1)} = \begin{pmatrix} y_0^{(1)} \\ y_1^{(1)} \\ \vdots \\ y_{19}^{(1)} \end{pmatrix} = \text{ReLU}(Z^{(1)}) = \begin{pmatrix} \text{ReLU}(z_0^{(1)}) \\ \text{ReLU}(z_1^{(1)}) \\ \vdots \\ \text{ReLU}(z_{19}^{(1)}) \end{pmatrix}$$

2. פلت זה מועבר לשכבה השנייה, אשר בה מופעלת שוב פונקציה ליינארית, **שונה מהקודמת** :

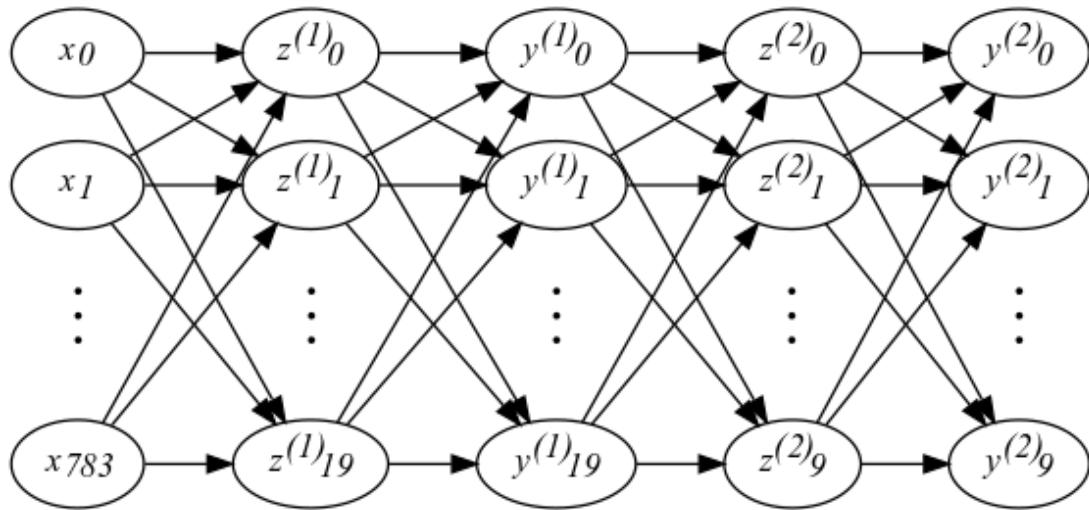
$$Z^{(2)} = \begin{pmatrix} z_0^{(2)} \\ z_1^{(2)} \\ \vdots \\ z_9^{(2)} \end{pmatrix} = \begin{pmatrix} w_{0,0}^{(2)}y_0^{(1)} + w_{0,1}^{(2)}y_1^{(1)} + \dots + w_{0,19}^{(2)}y_{19}^{(1)} + b_0^{(2)} \\ w_{1,0}^{(2)}y_0^{(1)} + w_{1,1}^{(2)}y_1^{(1)} + \dots + w_{1,19}^{(2)}y_{19}^{(1)} + b_1^{(2)} \\ \vdots \\ w_{9,0}^{(2)}y_0^{(1)} + w_{9,1}^{(2)}y_1^{(1)} + \dots + w_{9,19}^{(2)}y_{19}^{(1)} + b_9^{(2)} \end{pmatrix}$$

3. לבסוף, תוצאת החישוב מועברת לפונקציה softmax,

$$Y^{(2)} = \begin{pmatrix} y_0^{(2)} \\ y_1^{(2)} \\ \vdots \\ y_9^{(2)} \end{pmatrix} = \text{softmax}(Z^{(2)}) = \frac{1}{\sum_{n=0}^9 e^{z_n^{(2)}}} \begin{pmatrix} e^{z_0^{(2)}} \\ \vdots \\ e^{z_9^{(2)}} \end{pmatrix}$$

אשר הפלט שלו הוא הסתברויות הסיווג הסופיות – פלט המודל.

באיור סכמטי,



בכתב וקטורי, פעולה החישוב המבוצע ברשות היא

$$Y^{(2)} = \text{softmax}\left(W^{(2)} \text{ReLU}\left(W^{(1)}X + b^{(1)}\right) + b^{(2)}\right)$$

ונניתן לראות שהפרמטרים של המודל הם המטריצות  $W^{(1)}, W^{(2)}$  והוקטורים  $b^{(1)}, b^{(2)}$ , אשר קובעים את ערך הארגזיות הלינאריות בכל שכבה.

למודל זה כמו יתרונות על פני הרשות השטויחה, הראשון שבhem הוא היוטו לא מונוטוני כפונקציה של הקלט. זכרו שערבי וקטור הקלט  $X$  הם הפיקסלים של תמונת פריט לבוש כלשהו – ערך קרוב ל-1 מייצג פיקסל כהה וערך קרוב ל-0 מייצג פיקסל בהיר. כמו כן, זכרו שהחישוב המבוצע ברשות השטויחה הוא

$$Y = \text{softmax}(W \cdot X + b)$$

הן הפונקציה softmax והן הארגזציה הלינארית מונוטוניות בקלט שלו, ולפיכך הפלט מונוטוני אף הוא. משמעות הדבר היא שאם, לדוגמה, אחרי האימון התקבל משקל חיובי  $w_{k,m} > 0$  אז ככל שהפיקסל  $i$  שי בקלט כהה יותר, כך ההסתברות לסיווג פריט הלבוש במלבוקה ה- $k$  ית תהיה גבוהה יותר. מובן מالיאו שתכונה זו אינה מתאימה לבעה המורכבת של זיהוי פריט לבוש הדורשת מהמודל להבין את ההקשר בין הפיקסלים השונים. הוספת שכבה נוספת לרשת, עם אקטיבציה לא לינארית כגוןReLU, תאפשר את הגמישות הרצiosa במקרה זה – למודל תהיה אפשרות להימנע מהתלות המונוטונית בקלט.

שימוש לבשה האקטיבציה הלא לינארית בין שכבה היא **הכרחית** למימוש זו. לו היוו מפעלים פונקציה לינארית בלבד בעבר משכבה לשכבה, קרי מחשבים את  $Z^{(2)} = W^{(2)}Z^{(1)} + b^{(2)}$ , היה מתאפשר מודל השקול לרשות השטווחה, כפי שתוכicho בשאלות לתרגול בסוף פרק זה. לעיתים גמישות רצiosa זו גובה מחיר – אימון הרשות מתגזר יותר, וכן הרשות עלולה לשונן את אוסף הנתונים המשמש לאימונו, במקומם ללמידה הכללים פשוטים העומדים מאחוריו. הכלים העומדים לרשותנו כדי להתמודד עם אתגרים אלו הם נושאינו שטי היחידות הבאות בקורס זה.

## שאלות לתרגול

1. הוכיחו שלאחר אימון רשות שטווחה לזיהוי פריטי הלבוש, תמיד יתקבל מודל מונוטוני בקלט.

**הנחייה:** חשבו את הנגזרת  $\frac{\partial y_k}{\partial x_m}$  בעזרת כל השורשות, והסתכלו על הסימן של הביטוי.

2. הראו שניתן לבחור את הפרמטרים של הרשות העמוקה שתוארה לעיל, כך שהיא אינה מונוטונית.

**הנחיות:**

- התבוננו בתלות של  $y_0$  בפיקסל הקלט  $x_0$  בלבד.
- בחרו את  $w_{0,0}^{(1)}$  ואת  $w_{1,0}^{(1)}$  בסימנים הפורכים: האחד חיובי, השני שלילי. שאר הפרמטרים בשכבה הראשונה יכולים להיות 0.
- בחרו את  $w_{0,0}^{(2)}$  ואת  $w_{0,1}^{(2)}$  בסימנים זרים. שאר הפרמטרים בשכבה השנייה יכולים להיות 0.

3. הוכיחו שчисוב שתי שכבות של ארגזציה לינארית בזו אחר זו שකלה לחישוב ארגזציה לינארית אחת. בambilים אחרים, הוכיחו שאם מחשבים את  $Z^{(2)}$  כך,

$$Z^{(1)} = W^{(1)}X + b^{(1)}$$

$$Z^{(2)} = W^{(2)}Z^{(1)} + b^{(2)}$$

אז למעשה, ניתן לחשב ישרות  $c = AX + Z^{(2)}$ . מצאו את המטריצה  $A$  ואת הווקטור  $c$  המתאים. הסיקו שרשות עמוקה ללא אקטיבציה לא לינארית בין שכבה לשכבה שකלה לרשות שטווחה בעל שכבה לינארית אחת.

4. א. צרו אוסף נתונים של נקודות שורות ולבנות בעוזרת הפונקציה `make_moons` מהספרייה `sklearn.datasets`.

ב. אמנו את מודל הסיווג מוצלח? נמקו תשובתכם בבדיקה הילמוד הנווכית על אוסף נתונים זה.

ג. האם מודל הסיווג מוצלח? נמקו תשובתכם בבדיקה על יכולות המודל ועל מבנה אוסף הנתונים.

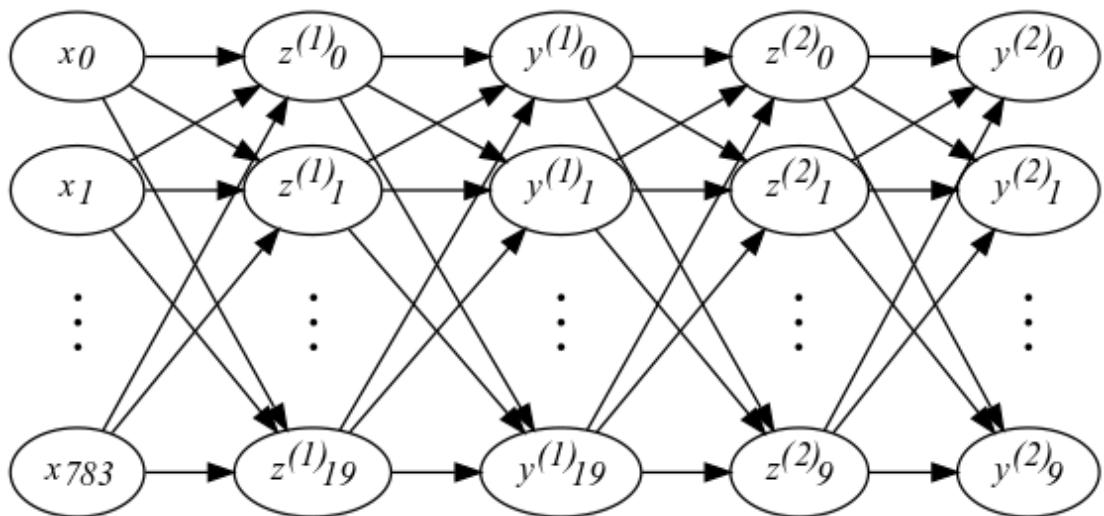
- ד. הרחיבו את המודל – הוסיפו שכבות בין הקלט לבין נוירון הסיווג, ואמנו את המודל החדש.
- ה. האם ביצועי המודל השתפרו? הסבירו מדוע.

## יחידה 3: אופטימיזציה

### התפshootות לאחר

בסיום יחידת הלימוד הקודמת ראיינו כיצד לאמן רשת נוירונים عمוקה בעזרת אלגוריתם מורד הגראדיאנט, אך השארנו את הפרטים של חישוב הנזירות למערכת הגזרה האוטומטית. בעת נבייט בឋיליך זה לעומק, ודרך נבין את האלגוריתם העומד מאחורי חישוב הגראדיאנט עצמו בכל איטרציה – התפshootות לאחר (back propagation).

האיור שלහלן יזכיר לכם את הרשות שבה אנו עוסקים.



תמונה הקלט מוגנת אל הרשות כווקטור פשוט,  $X$ , והחישוב המבוצע עליו הוא

$$Y^{(2)} = \text{softmax}\left(W^{(2)} \text{ReLU}\left(W^{(1)} X + b^{(1)}\right) + b^{(2)}\right)$$

אנו מפרשים תוצאה זו בתוור הסתברות הסיווג ל-10 המחלקות של פריטי הלבוש באוסף הנתונים Fashion-MNIST. כדי להעריך את טיב החיזוי של הרשות עבור הקלט הנוכחי, אנו מחשבים את פונקציית המחיר של האנתרופיה הצולבת,

$$H(X, Y_t) = -\sum_{n=0}^9 y_n \log(y_n^{(2)})$$

לאחר מכן, אנו גוזרים את הפונקציה לפי כל ערכי הפרמטרים המופיעים ברשות, כאשר במקרה הנוכחי הם משקל השכבות הлиינאריות, המטריצות  $W^{(1)}, W^{(2)}$  והווקטורים  $b^{(1)}, b^{(2)}$ . לאחר חישוב גראדיאנט

זה (או למעשה ביזמגנית), יש לחזור על הפעולה עבור minibatch של נתונים, ולבסוף בשביל צעד עדכון ייחיד של אלגוריתם SGD אנו מחשבים ממוצע של הגראדיאנטים,

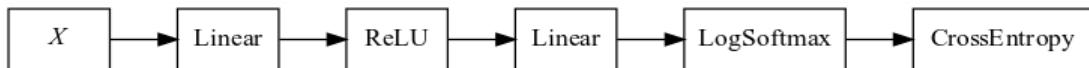
$$\tilde{\nabla}C = \frac{1}{\#\text{minibatch}} \sum_{(X, Y_t) \in \text{minibatch}} \nabla H(X, Y_t)$$

- נעסק כאן בפירוט התהליך שבו מוחושב  $\nabla H(X, Y_t)$ . התובנות העיקריות מאחוריה השיטה זו:
1. לצורך חישוב פונקציית המחיר  $H$  אנו מפעילים פונקציות פשוטות זו על זו, ואת הנזורות של כל אחת מפונקציות אלו לפי הקלט שלה קל לנו לחשב.
  2.  $H$  תלויות בערכי הפרמטרים דרך מספר רב של פונקציות (מספר השכבות בראשת), ואת הנזורות של  $H$  לפי פרמטרים ניתן לקבל מהnezורת הפשוות **בעזרת כלל השרשota**.
  3. כל הערכים הדורשים לחישוב הנזורות אלו כושבו במהלך הקלט בראשת וחישוב ההסתברויות החזויות. לפיכך, כדי להציג את חשיבותו לאלגוריתם backpropagation, שלב הזנת הקלט נקרא גם forward propagation.
  4. עלות חישוב  $H$  היא כפולה החישוביות של צעד forward propagation ייחיד.

אם כן, ניגש ממשימה מהסוף אל ההתחלת, אך קודם לכן נזכיר שבעת מימוש הרשות בחרנו לחשב בשכבה האחורונה את הלוגריתם של הפונקציה softmax במקום את הפונקציה עצמה. בעת נראה יתרונו נוספת לבחירה זו – חישוב הנזורות נעשה קל במידה ניכרת. נניח אפוא שפלט הרשות הוא

$$\log(Y^{(2)}) = \log \text{softmax}\left(W^{(2)} \text{ReLU}\left(W^{(1)} X + b^{(1)}\right) + b^{(2)}\right)$$

ונזכיר את הרשות ברמת הפשטה גבואה יותר, המדגישה את המבנה שלה כפונקציה מקוונת.



בשלב הראשון עליינו לחשב את הנזורת של פונקציית המחיר  $H$  לפי הקלט שלו, :

$$\frac{\partial H(X, Y_t)}{\partial \log(y_k^{(2)})} = -\sum_{n=0}^9 \frac{\partial}{\partial \log(y_k^{(2)})} \left( y_{tn} \log(y_n^{(2)}) \right) = -y_{tk}$$

ויש לציין ש-  $y_{tk} = 1$  אם ורק אם  $k$  היא המולקה האמיתית שאליה שייכת תומנת הקלט  $X$  באוסף נתונים האימון.

כעת ממשיך לנوع אחרה בראשת, ונתמקד בשכבה הפלט שלו.



עלינו לגוזר את הפונקציה LogSoftmax. פונקציה זו מקבלת כקלט וקטור k-ממדי  $U$ , ומחזירה וקטור k-ממדי  $V$ , אשר הגדרתו היא

$$V = \text{LogSoftmax}U = \text{LogSoftmax} \begin{pmatrix} u_0 \\ \vdots \\ u_k \end{pmatrix} = \begin{pmatrix} u_0 - \log \left( \sum_{n=0}^k e^{u_n} \right) \\ \vdots \\ u_k - \log \left( \sum_{n=0}^k e^{u_n} \right) \end{pmatrix}$$

לפיכך, יש לחשב את הנגזרת של כל אחד ממערכות וקטור הפלט,  $v_m$ , לפי כל אחד ממשתני הקלט  $u_n$ . למעשה, אנו רוצים לחשב את מטריצת היוקוביון:

$$J_{\text{LogSoftmax}} = \begin{pmatrix} \frac{\partial v_0}{\partial u_0} & \dots & \frac{\partial v_0}{\partial u_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial v_k}{\partial u_0} & \dots & \frac{\partial v_k}{\partial u_k} \end{pmatrix}$$

чисוב ישיר יניב עבור איברי האלכסון של מטריצה זו,

$$\frac{\partial v_k}{\partial u_k} = 1 - e^{v_k}$$

עבור שאר המטריצה:

$$\frac{\partial v_m}{\partial u_n} = -e^{v_n}$$

עלינו לזכור שהקלט לשכבה זו בראשת שלנו הוא  $Z^{(2)}$ , וכן שפלט שכבת ה-softmax ללא הלוגריתם הוא  $Y^{(2)}$ . נוסף על כך נשים לב לכך ש-

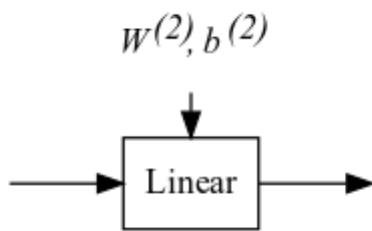
$$e^V = \exp(\text{LogSoftmax}U) = \text{Softmax}U$$

על כן, היוקוביון המתkeletal במקרה הנוכחי הוא מטריצה מסדר  $10 \times 10$

$$\cdot \begin{pmatrix} \frac{\partial \log(y_0^{(2)})}{\partial z_0^{(2)}} & \frac{\partial \log(y_0^{(2)})}{\partial z_1^{(2)}} & \dots & \frac{\partial \log(y_0^{(2)})}{\partial z_9^{(2)}} \\ \frac{\partial \log(y_1^{(2)})}{\partial z_0^{(2)}} & \frac{\partial \log(y_1^{(2)})}{\partial z_1^{(2)}} & \dots & \frac{\partial \log(y_1^{(2)})}{\partial z_9^{(2)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \log(y_9^{(2)})}{\partial z_0^{(2)}} & \frac{\partial \log(y_9^{(2)})}{\partial z_1^{(2)}} & \dots & \frac{\partial \log(y_9^{(2)})}{\partial z_9^{(2)}} \end{pmatrix} = \begin{pmatrix} 1 - y_0^{(2)} & -y_1^{(2)} & \dots & -y_9^{(2)} \\ -y_0^{(2)} & 1 - y_1^{(2)} & \dots & -y_9^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ -y_0^{(2)} & -y_1^{(2)} & \dots & 1 - y_9^{(2)} \end{pmatrix}$$

נמשיך לנوع אחריה בראשת וליחס נגזרות, ונשים לב בividוד לכך שלראשונה הגענו לשכבה שיש בה פרמטרים אשר לפיהם, בין השאר, יש לחשב את גרדיאנט פונקציית המחיר. זו השכבה הילינארית השנייה, ובאיור:





כעת נחשב את הנזירות של פלט שכבה זו **לפי הפרמטרים**: פלט השכבה הוא וקטור עשר-ממדי,  $Z^{(2)}$ , אשר יש לגזור כל איבר שלו לפי כל אחד מהפרמטרים: איברי המטריצה  $W^{(2)}$  ואיברי הווקטור  $b^{(2)}$ . זכרו שהפעולה הلينיארית המתבצעת בשכבה זו היא

$$Z^{(2)} = \begin{pmatrix} z_0^{(2)} \\ z_1^{(2)} \\ \vdots \\ z_9^{(2)} \end{pmatrix} = \begin{pmatrix} w_{0,0}^{(2)}y_0^{(1)} + w_{0,1}^{(2)}y_1^{(1)} + \dots + w_{0,19}^{(2)}y_{19}^{(1)} + b_0^{(2)} \\ w_{1,0}^{(2)}y_0^{(1)} + w_{1,1}^{(2)}y_1^{(1)} + \dots + w_{1,19}^{(2)}y_{19}^{(1)} + b_1^{(2)} \\ \vdots \\ w_{9,0}^{(2)}y_0^{(1)} + w_{9,1}^{(2)}y_1^{(1)} + \dots + w_{9,19}^{(2)}y_{19}^{(1)} + b_9^{(2)} \end{pmatrix}$$

ולכן ערכי הנזירות הם :

1. עבור פרמטרים ופלט מסוימת שורה,  $\frac{\partial z_k^{(2)}}{\partial w_{k,m}^{(2)}} = y_m^{(1)}$ ,  $\frac{\partial z_k^{(2)}}{\partial b_k^{(2)}} = 1$
2. עבור פרמטרים ופלט משוראות שונות:  $\frac{\partial z_k^{(2)}}{\partial w_{n,m}^{(2)}} = 0$ ,  $\frac{\partial z_k^{(2)}}{\partial b_n^{(2)}} = 0$  כאשר  $n \neq k$

מטרתנו היא לחשב את הנזירות של  $H$  לפי ערכי הפרמטרים, וכעת יש בידינו כל הדרוש לכך עבור הפרמטרים  $W^{(2)}$  ו-  $b^{(2)}$ . יש להלץ נזירות אלו מהערכיהם שהושבו לעיל בעזרת שימוש חזרה בכלל השרשרת. לאחר התבוננות במבנה הראשית, ניתן לראות כי  $H$  תלויה בפרמטרים אלו אך ורק דרך פלט השכבה,  $Z^{(2)}$ , ועל כן נקבל מכלל השרשרת, למשל עבור  $w_{p,q}^{(2)}$ :

$$\frac{\partial H}{\partial w_{p,q}^{(2)}} = \sum_{k=0}^9 \frac{\partial H}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial w_{p,q}^{(2)}}$$

את הנזירות מהצורה  $\frac{\partial H}{\partial z_k^{(2)}}$  יש לחשב פעם אחת בלבד, ולהשתמש בהן עבור כל הפרמטרים.שוב כלל השרשרת יהיה שימושי,

$$\frac{\partial H}{\partial z_k^{(2)}} = \sum_{n=0}^9 \frac{\partial H(X, Y_t)}{\partial \log(y_n^{(2)})} \frac{\partial \log(y_n^{(2)})}{\partial z_k^{(2)}}$$

בשלב זה הגענו לביטויים אשר את ערכיהם חישבנו לעיל, ולכן כל שנותר לעשות הוא להציב:

$$\frac{\partial H}{\partial z_k^{(2)}} = \sum_{n=0}^9 -y_{tn} \left( 1\{n=k\} - y_n^{(2)} \right)$$

כאשר

$$\cdot \mathbf{1}\{n=k\} = \begin{cases} 1 & n=k \\ 0 & n \neq k \end{cases}$$

לבסוף נקבל, למשל בעבר :  $w_{0,1}^{(2)}$

$$\begin{aligned} \frac{\partial H}{\partial w_{0,1}^{(2)}} &= \sum_{k=0}^9 \frac{\partial H}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial w_{0,1}^{(2)}} = \frac{\partial H}{\partial z_0^{(2)}} \frac{\partial z_0^{(2)}}{\partial w_{0,1}^{(2)}} = \\ &= \sum_{n=0}^9 -y_m \cdot \left( \mathbf{1}\{n=0\} - y_n^{(2)} \right) \cdot y_1^{(1)} \end{aligned}$$

ובערך זה נשתמש לעדכון הפרמטר בצעד הבא של אלגוריתם האופטימיזציה.  
מכיוון שהרשota عمוקה, התהילה איןנו נגמר בשלב זה, שכן יש עוד פרמטרים לעדכן בכל צעד. נמשיך לגזור דרך השכבות עד אשר נגיע אליהם. ראשית, עלינו לגזור את השכבה הילינארית הנ"ל לפי הקלט שלו, וההתוצאה המתתקבלת היא

$$\cdot \frac{\partial z_k^{(2)}}{\partial y_m^{(1)}} = w_{k,m}^{(2)}$$

שנית, עלינו לחשב את הנגזרות לפי **קלט השכבה**,  $\frac{\partial H}{\partial Y^{(1)}}$ . נעשה זאת שוב בעזרת כל השרשת ונשים לב שאנו חוזרים ומשתמשים בו, אשר חישבנו לעיל :

$$\cdot \frac{\partial H}{\partial y_m^{(1)}} = \sum_{k=0}^{19} \frac{\partial H}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial y_m^{(1)}} = \sum_{k=0}^{19} \sum_{n=0}^9 -y_m \left( \mathbf{1}\{n=k\} - y_n^{(2)} \right) w_{k,m}^{(2)} .$$

נשתמש בערכים אלו כדי להמשיך ולהחשב את הנגזרות מעבר לאקטיבציה ה-ReLU. נזכיר ש-

$$\begin{pmatrix} y_0^{(1)} \\ \vdots \\ y_{19}^{(1)} \end{pmatrix} = Y^{(1)} = \text{ReLU}(Z^{(1)}) = \begin{pmatrix} \text{ReLU}(z_0^{(1)}) \\ \vdots \\ \text{ReLU}(z_{19}^{(1)}) \end{pmatrix}$$

וכן שי

$$\text{ReLU}(x) = x \cdot \mathbf{1}\{x \geq 0\} = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

נתעלם מכך שפונקציה זו אינה גזירה בנקודה 0, שכן הסתברות שהקלט שלו יהיה בדיק 0 היא זניחה, ונקבל שכאשר  $n \neq m$ ,  $\frac{\partial y_m^{(1)}}{\partial z_n^{(1)}} = 0$  וכאשר יש שווויון,

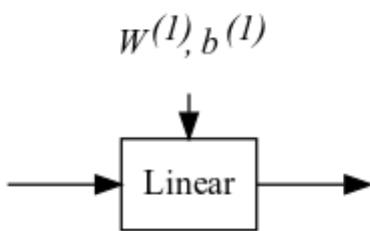


$$\cdot \frac{\partial y_m^{(1)}}{\partial z_m^{(1)}} = 1\left\{z_m^{(1)} \geq 0\right\} = \begin{cases} 1 & z_m^{(1)} \geq 0 \\ 0 & z_m^{(1)} < 0 \end{cases}$$

נותר רק להפעיל שוב את כלל השרשרת,

$$\begin{aligned} \frac{\partial H}{\partial z_m^{(1)}} &= \sum_{p=0}^{19} \frac{\partial H}{\partial y_p^{(1)}} \frac{\partial y_p^{(1)}}{\partial z_m^{(1)}} = \frac{\partial H}{\partial y_m^{(1)}} \frac{\partial y_m^{(1)}}{\partial z_m^{(1)}} = \\ &= \sum_{k=0}^{19} \sum_{n=0}^9 -y_{tn} \left(1\{n=k\} - y_n^{(2)}\right) w_{k,m}^{(2)} 1\left\{z_m^{(1)} \geq 0\right\} \end{aligned}$$

בשלב זה הגיענו שוב לשכבה LINEARית עם פרמטרים, כמו צג באיור,



ובידינו הנגורות של פונקציית המחיר לפי פלט השכבה,  $\frac{\partial H}{\partial Z^{(1)}}$ . החישוב החל מכיוון אנלוגי לזה שנעשה

לעיל עבור השכבה הLINEARית הקודמת, ויסטאים כאשר יהיה בידינו גרדיאנט של  $H$  לפי כל הפרמטרים של הרשת.

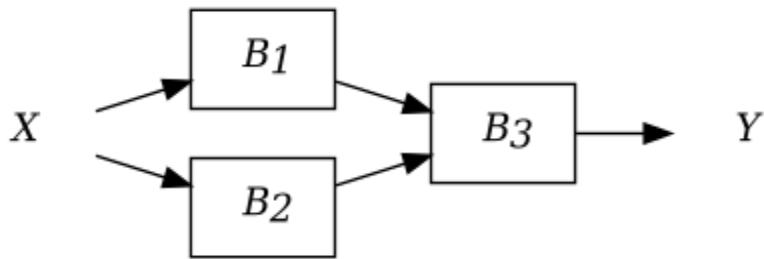
מדוגמה זו נוכל להבין את האלגוריתם הכללי, כפי שהוא פועל על רשות של **בלוקים** המוחברים זה לזה בראץ', כמו צג באיור:



בלוק ייחיד עשוי להיות מורכב מכמה שכבות ולבצע חישוב מורכב, אך כל עוד ניתן לגזרו דרכו: לגזר את פלט הבלוק לפי הקלט שלו, אלגוריתם התפשטות לאחריו יהיה שימושי. יש לזכור להעביר אחריה את הנגורות של פונקציית המחיר דרך הבלוק, וכן לחשב את הנגורות הפרמטרים של הבלוק עצמו. שימושו לבשחמלקה Sequential nn למעשה מימוש ארכיטקטורת בלוקים זו – בעת הגדרת אובייקט, המשמש מעביר את פרטי כל הבלוקים ואת הסדר שבהם מופיעים ברשות.

## שאלות לתרגול

1. השלימו את חישוב  $H$  : חשבו את הנזירות של פונקציית המחיר לפי הפרמטרים של השכבה הילינארית הראשונה.
2. כתבו בפסודו קוד את אלגוריתם ההתפשטות לאחר עבור רשת המורכבת מרצף בלוקים.
3. הסבירו כיצד יש לשנות את אלגוריתם ההתפשטות לאחר כדי להתמודד עם רשת שבה מבנה הבלוקים הזה :



כיצד יש לחשב את  $\frac{\partial Y}{\partial X}$ ? רמז: שימו לב לחבר המתמטי בין הפלט לקלט:  
 $, Y = B_3(B_2(X), B_1(X))$

והשתמשו בכלל השרשרת עבור פונקציה ריבת-משתנים.

4. הריצו את קטע הקוד המצורף:

```

from torch import nn
net = nn.Sequential(
    nn.Linear(2,1),
    nn.ReLU(),
    nn.Linear(1,2),
    nn.ReLU(),
    nn.Linear(2,1),
    nn.Sigmoid()
)
  
```

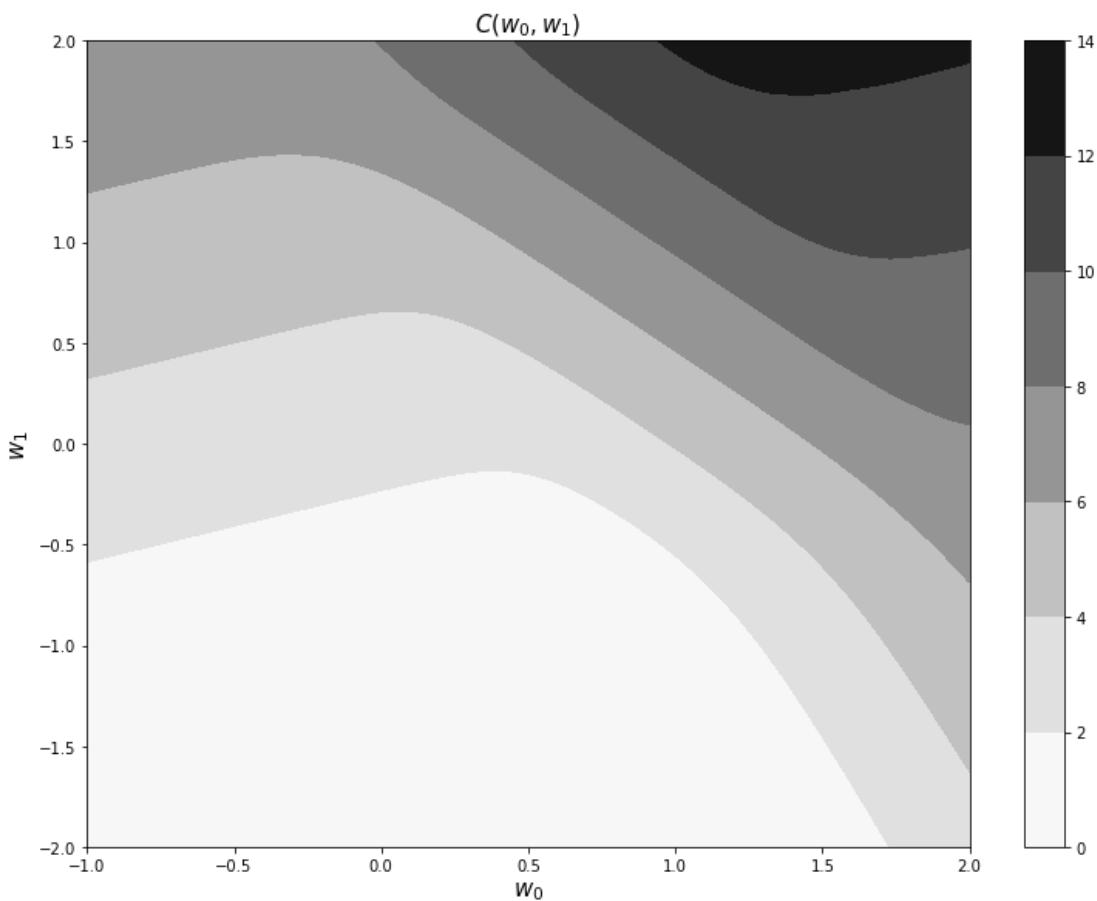
- א. ציירו את רשת המתקבלת. על הציר להראות את מבנה השכבות של הרשת וכן מהו הקלט ומהו הפלט של כל ניורון.
- ב. הדפיסו את ערכי הפרמטרים של הרשת, וכתבו בשורה אחת את החישוב שהרשת מבצעת.
- ג. הזינו לתוך רשת את הקלט  $(1,1) = X$  והניחו שהפלט הצפוי הוא 1.
- ד. חשבו ידנית ולפי אלגוריתם ההתפשטות לאחר את  $H$  עבור קלט זה, בהנחה שפונקציית המחיר היא אנטרופיה צולבת.
- ה. השוו את החישוב שלכם לזה המתתקבל ממערכת הגזירה האוטומטית.

## קצב הלמידה

בפרק זה נחזר למודל הנוירון היחיד אשר מסוג נקודות לשתי מחלקות במרחב דו-ממדי – נקודות שחרורות ונקודות לבנות. נזכיר שהמודל תלוי בשלושה פרמטרים,  $w_0, w_1, b$  ומצביע את הסתברות השيءיות למחלקות הנΚודות השחרורות לפי הנוסחה

$$y = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + b)}}$$

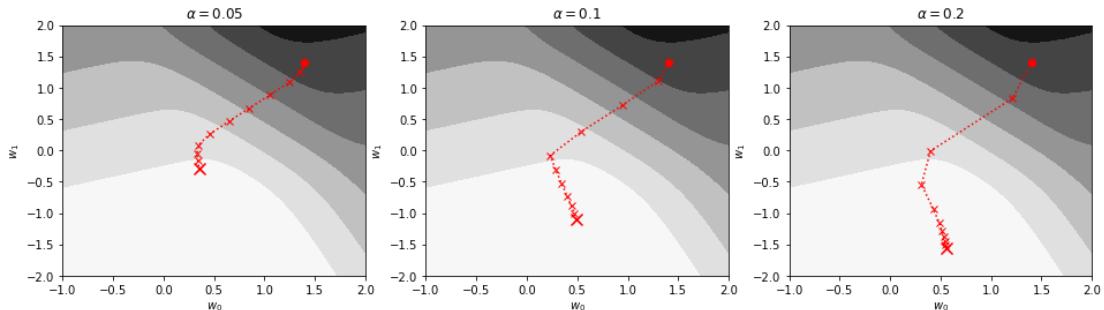
כעת נניח כי הפרמטר  $b$  קבוע מראש, וכי בעת אימון הנוירון אנו משנים את הערכים של שני הפרמטרים האחרים. עבור בחירה מסוימת של פרמטרים אלו מקבל מודלים מוצלחים, ועבור בחירה אחרת – מודלים מוצלחים פחות. נציג את מחיר האנטרופיה הצולבת המתקבל כתלות ב- $w_0, w_1$ , כאשר ערכו של  $b$  נקבע ל-5.



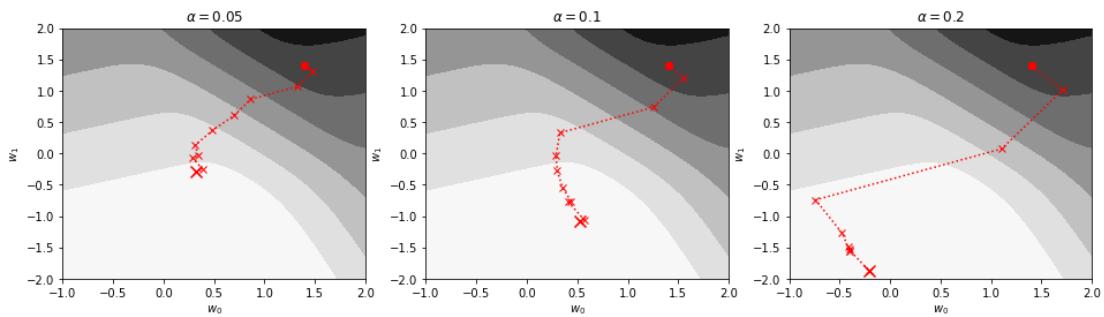
באירועים שלහן נדגים את התוצאות של הפעלת אלגוריתם מורד הגדיאנט (המלא), שבו אנו מעדכנים את הפרמטרים לפי הנוסחה

$$(w_0, w_1) = (w_0, w_1) - \alpha \nabla C(w_0, w_1)$$

עבור ערכי קצב למידה  $\alpha$  שונים. שימושו לב שבעל אחת מהפעמים אתחלנו את האלגוריתם בערכים  $(w_0, w_1) = (1.4, 1.4)$ , וכן שעשרה הצעדים הראשוניים מסומנים בא撇ה.



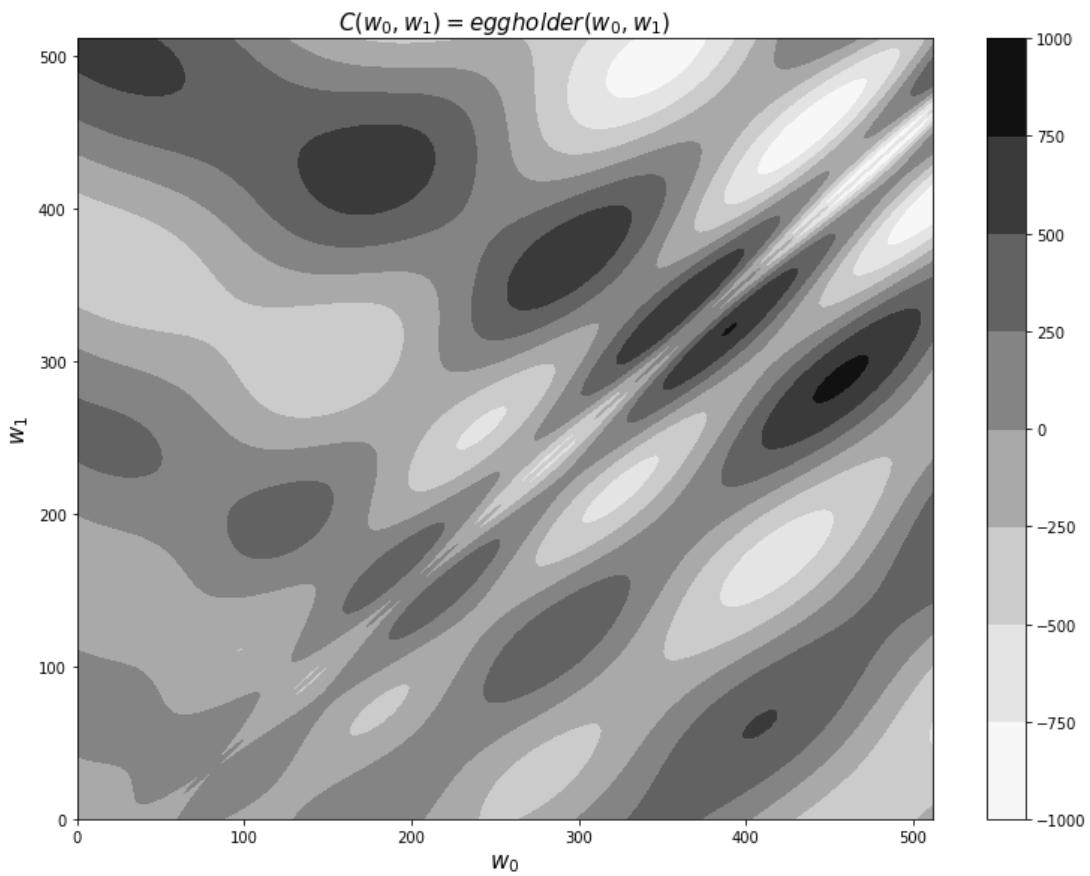
מайורים אלו ניתן לראות את החשיבות של בחירת קצב הלמידה הנכון כבר בדוגמה פשוטה שכזו : קצב איטי מדי יוביל להתקנות איטית. כאשר אנו משתמשים בגרדיאנט אקראי, קצב הלמידה חשוב אף יותר. נחליף את האלגוריתם בחישוב שלעיל ל-SGD, כאשר נחשב כל איטרציה על פני minibatch של שתי דוגמאות ונקבל את התוצאות המופיעות באירור שלහן.



במעבר לגרדיאנט האקראי ניכר כי ההתקנות איטית יותר, שכן לעיתים הגרדיאנט האקראי אינו קירוב מספיק טוב של הגרדיאנט המלא, ולפיכך כיוון התנועה אינו אידיאלי. מודגמה זו ניכר כי פתרון אפשרי הוא להאיץ את קצב הלמידה. לרוב פתרון זה אינו מתאים, שכן פונקציית המחיר של רשותות عمוקות יותר אינה נוחה לעובדה כמו הנויל, וקצב התקנסות מהיר מדי עלול להוביל להתקדרות האלגוריתם.

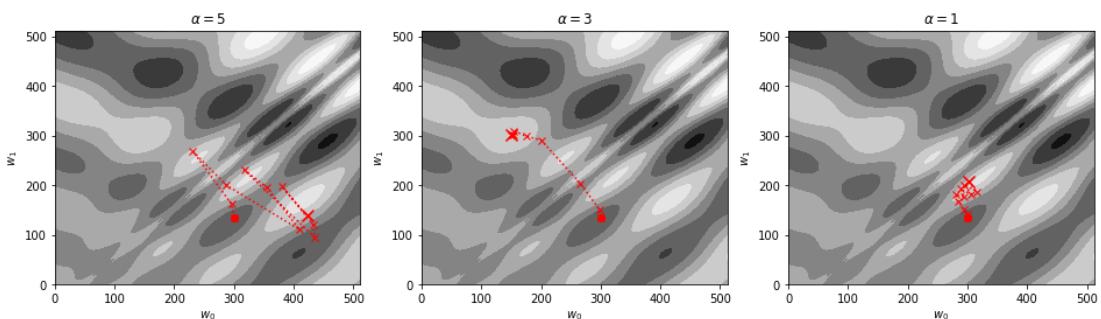
נניח לצורך הדיוון כי אנו עוסקים במודל בעל שני פרמטרים, אך במקרה בו פונקציית המחיר היא פונקציה "תבנית הביצים", אשר הגדרתה וצורה מופיעים להלן.

$$\text{eggholder}(w_0, w_1) = -(w_1 + 47) \sin\left(\sqrt{\left|w_1 + \frac{w_0}{2} + 47\right|}\right) - w_0 \sin\left(\sqrt{\left|w_0 - (w_1 + 47)\right|}\right)$$

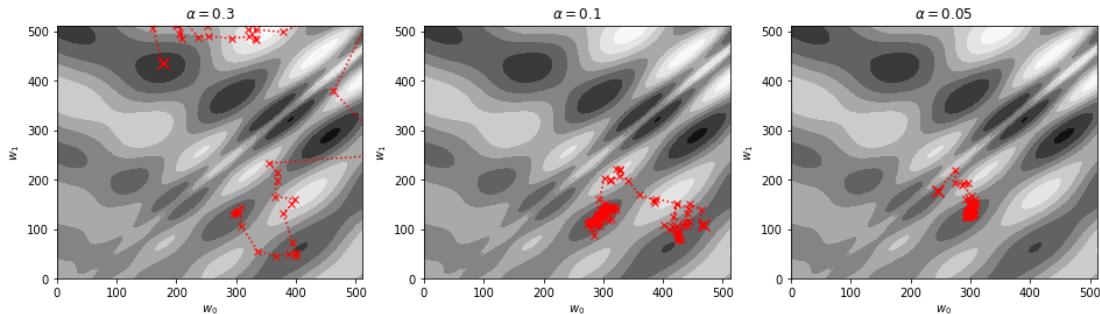


מצירוף הפונקציה ניתן להבין מדוע נראה תבנית ביצים, והיא בבחינת דוגמה מאתגרת לאלגוריתמי אופטימיזציה, שכן יש בה מספר רב של נקודות מינימום מקומיות.

תוצאות הריצה של 100 איטרציות של אלגוריתם מורד הגרדיאנט (המלא) מאוירות להלן, ומהן ניכר שעבור קצב למידה מהיר, האלגוריתם כבר אינו מתכנס.



המצב עבור מورد הגרדיאנט האקראי חמוץ אף יותר מאשר בקצבים למידה איטיים, כפי שמודגס באירוע שלහלו.



שימו לב שהוא עוסקים כאן בסימולציה, שכן בדוגמה זו אין ברשותנו אוסף נתונים אשר ממנו אנו יכולים לדוגם minibatch ועל בסיסו לחשב גרדיאנט אקראי. לפיכך, אנו **מדמים** זאת באמצעות חישוב הגרדיינט האמיתי והוספה של רעש אקראי, כפי שניתנו להראות בשורת הקוד מלולאת האימון שבה אנו מעדכנים את ערכי הפרמטרים :

```
with torch.no_grad():
    w -= alpha*(w.grad*torch.normal(1, 15, size=w.size()))()
```

עם זאת, דוגמאות אלו מיצגות נסמה מצבי שניתקל בהם בעת אימון רשת عمוקה.

בהתבוננות באירורים שלעיל ניתן לראות שעבור אף אחד מקצבי הלמידה הנבחרים SGD לא הוכנס כלל – ערכי הפרמטרים שבהם עצר האלגוריתם רחוקים מלהיות מינימום מקומי, שכן הם אינם נמצאים במרכז "עמק" לבן. ככלים לפרטנו בעיה זו נלמד בהמשך ייחידה זו.

## שאלות לתרגול

1. לפונקציית תבנית הביצים נקודת מינימום מקומי בקרבת הנקודה  $(w_0, w_1) = (400, 130)$ . נסו לצורך אותה בעזרת אלגוריתם מורד הגרדיינט עם נקודות התחלה שונות וקטבי למידה שונות.
2. חזרו על פוליה זו עבור נקודת המינימום המקומי הקרוב ל-  $(w_0, w_1) = (450, 420)$ .
3. הסבירו למה קשה יותר למצוא את הנקודה השנייה מאשר את הראשונה.



## תזכון קצב הלמידה

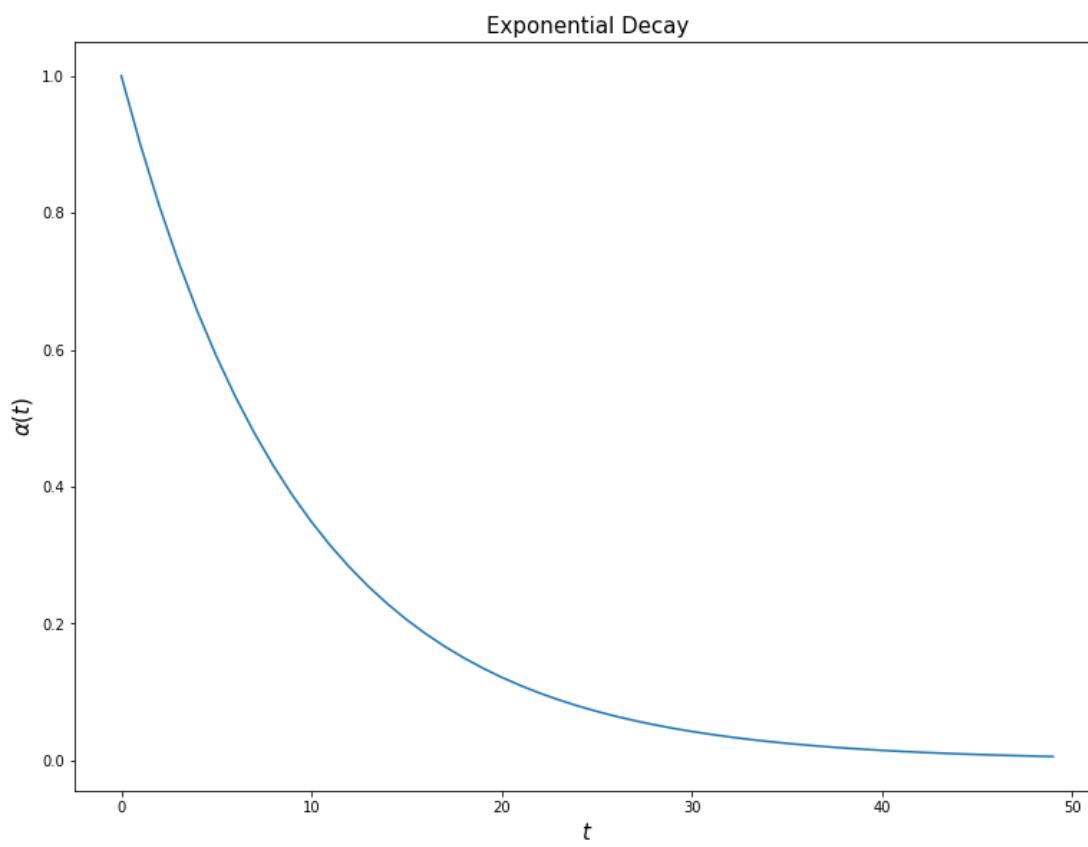
השיפור הראשון אשר נשלב באלגוריתם האופטימיזציה SGD הוא **דיעיכת קצב הלמידה** – בכל איטרציה נקטין את קצב הלמידה לפי כלל קבוע, כך שלאחר מספר רב של איטרציות, גם אם הגרדיינט עוד לא התאפס – קרי האלגוריתם עוד לא הוכנס לערכי פרמטרים אשר מהווים נקודת מינימום של פונקציית המחיר – קצב הלימוד יהיה אפסי בפני עצמו, ותהליך עדכון הפרמטרים יתייצב. נוסחת העדכון של הפרמטרים במודול בכל איטרציה תהיה

$$\text{Parameters} = \text{Parameters} - \alpha(t) \tilde{\nabla} C$$

כאשר  $t$  הוא מספר **epoch** הנוכחי, ו-  $\alpha(t)$  היא פונקציה יורדת לשתי. דוגמה פשוטה ו שימושית היא הקטנת את קצב הלמידה **פי** קבוע הנבחר מראש, כלומר,

$$\alpha(t) = \gamma \alpha(t-1) = \gamma^t \alpha(0)$$

כאשר  $1 > \gamma > 0$  כמובן. הגרף של פונקציית דעיכה זו עברו  $0.9 = \gamma$  מצויר להלן.



השימוש בקצב למידה משתנה בספרייה PyTorch הוא פשוט יותר, ומאפשר בעזרת אובייקט `lr_scheduler`. לאחר הגדרת **אובייקט האופטימיזציה**, מעבירים אותו כפרמטר **לאובייקט זמן קצב הלמידה**, יחד עם פונקציה הקובעת את הדעיכה **בכל צעד**, כדלהלן.

```

optimizer = torch.optim.SGD(model, lr=1)

decay      = lambda previous_lr: 0.9 ** previous_lr
scheduler = torch.optim.lr_scheduler.LambdaLR(
    optimizer, lr_lambda=decay)

```

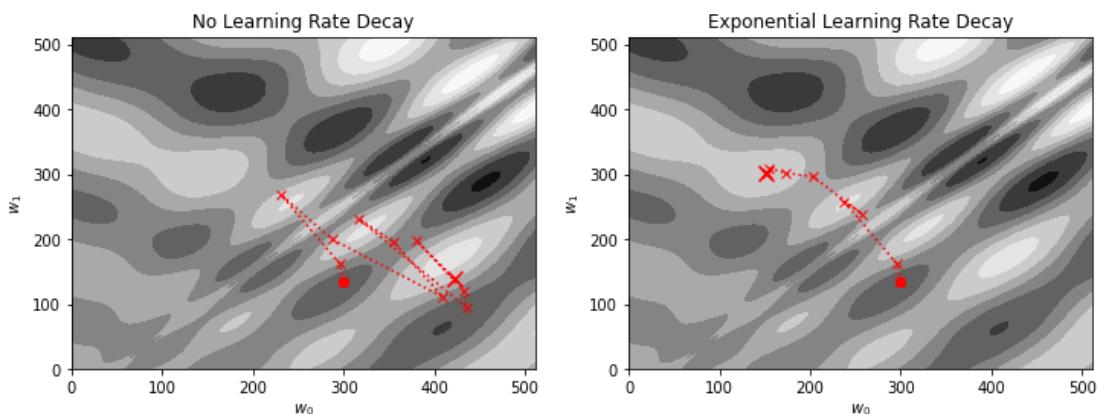
אחרי כן, יש להוסיף ללולאת האימון פקודה המעדכנת את קצב הלמידה **בסוף epoch**, זו הפקודה `scheduler.step()` המופיעה בסוף קטע הקוד שלහן.

```

for epoch in range(num_epochs):
    for batch in range(num_minibatches):
        .
        .
        .
        optimizer.step()
    scheduler.step()

```

ראו באירור למטה את ההשפעה של דעיכת קצב הלמידה על ריצת אלגוריתם מורד הגרדיינט (המלא), כפי שהופעל על פונקציית תבנית הביצים מהפרק הקודם.



הפונקציה המשמשת לחישוב המסלול מופיעה בקטע הקוד המצורף, ויש לשים לב שאנו מפעילים את דעיכת קצב הלמידה לפי ערכו של `decay_flag`, ושהת המסלול אנו שומרים במשתנה `.history` מנטקת את המשטנה ממערכת הגזירה האוטומטית. זכרו שהמתודה `detach()`

```

def GD_egg(alpha=5,start=[300.,135.],decay_flag=False):
    w      = torch.tensor(start,requires_grad=True)
    model = [w]
    num_epochs = 10
    optimizer = torch.optim.SGD(model, lr=alpha)
    if decay_flag:
        decay     = lambda previous_lr: 0.9 ** previous_lr
        scheduler = torch.optim.lr_scheduler.LambdaLR(
            optimizer, lr_lambda=decay)

    history      = torch.zeros(1+num_epochs,2)
    history[0,:] = w.detach()

    for epoch in range(num_epochs):
        optimizer.zero_grad()
        cost = eggholder(w[0], w[1])
        cost.backward()
        optimizer.step()
        if decay_flag:
            scheduler.step()
        history[1+epoch,:] = w.detach()

    return history

```

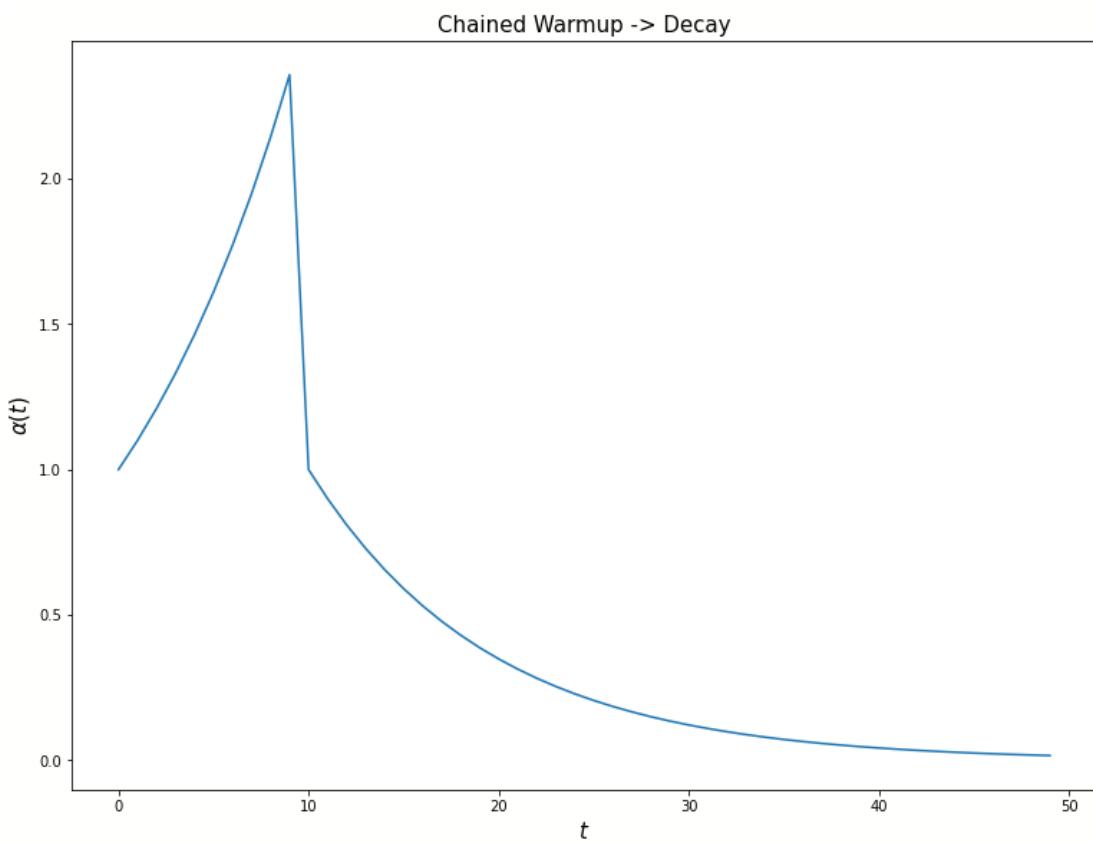
תזמיןו קצב הלמידה הוא ווחם פעיל בחקר רשתות נוירונים عمוקות, שכן לקבע הלמידה השפעה קריטית על ערכי הפרמטרים המתובלים. הספרייה `torch.optim.lr_scheduler` מספקת מספר רב של אובייקטים שבזורותם ניתן למש אסטרטגיות שונות, כפי שהתרפרמו בספרות המדעית. היריסטיקה מעניינת לדוגמה נקראת "חימום" (warmup), והרעיון העומד מאחוריה הוא שבתחלת ריצת האלגוריתם מומלץ דואק**הגביר** את קצב הלמידה אחרי כל epoch, כדי לאפשר לאלגוריתם לחזור איזור גודל יותר במרחב הפרמטרים. אחרי תקופה החימום עוברים לשלב של קצב למידה קבוע, לטובת התכונות האלגוריתם. בקטע הקוד שלහן אנו ממשים רעיון זה בעזרת אובייקט המחבר מתזמים שונים, בזה אחר זה.

```

warmup = lambda previous_lr: 1.1 ** previous_lr
decay  = lambda previous_lr: 0.9 ** previous_lr
scheduler1 = torch.optim.lr_scheduler.LambdaLR(optimizer,
                                                lr_lambda=warmup)
scheduler2 = torch.optim.lr_scheduler.LambdaLR(optimizer,
                                                lr_lambda=decay)
scheduler  = torch.optim.lr_scheduler.SequentialLR(optimizer,
                                                    schedulers=[scheduler1, scheduler2],
                                                    milestones=[10])

```

הfonקציה `SequentialLR` קושרת את שני מתזמי קצב הלמידה, כאשר בפרמטר `milestones` מופיע ה'epoch' שבו יש להחליפם. מתזמן קצב הלמידה המתබל מאויר להלן.



### שאלות לתרגול

1. אתרו את נקודות המינימום של פונקציית הבנייה הביצים בקרבת הנקודות  $(w_0, w_1) = (400, 130)$  ו-  $(w_0, w_1) = (450, 420)$  בעזרת אלגוריתם מורד הגראדיאנט בעל קצב למידה דועץ. השוו את התוצאות לאלגוריתם בעל קצב למידה קבוע.
2. הוסיפו מתזמן קצב למידה דועץ למודל הסיווג של התמונות מאוסף הנתונים Fashion-MNIST והשו את התוצאות עם דעיכת קצב הלמידה ובלעדיה. שימו לב שדעיכת קצב הלמידה מותבצעת בסוף כל epoch בלבד.

## שיפורים לאלגוריתם מורד הגרדיינט

### מומנטום

לאלגוריתם SGD פורסמו תוספות ושיפורים רבים, אשר הנפוץ בהם הוא שימוש במומנטום. הרעיון העומד מאחוריו פשוט: נדמיין את האלגוריתם חלקיים הנע במרחב הפרמטרים במורד הר – זו פונקציית המחיר. עם תנועתו, החלקיים צובר מהירות אשר משפיע על כיוונו. על החלקיים מופעל כוח בכיוון השילילי של הגרדיינט בנקודת הנוכחית, בכיוון התולול ביותר מטה, אך תנועתו אינה מושפעת רק מכוח זה, שכן הוא הגיע אלנקודת הנוכחית ב מהירות מסוימת. זהו המומנטום הדוחף אותו להמשיך את התנועה בכיוון המקורי, שאינו בהכרח ההפוך יותר בנקודת זו.

בנוסחאות, תוספת זו באה לידי ביטוי בכך שאנו שומרים ממוצע רץ של הגרדיינטים שהושבו באיטרציות הקודמות. זהו וקטור  $\text{המירות } v_t$ , ולו אנו מוסיפים את הגרדיינט החדש  $\tilde{\nabla}C_t$  ובודקתו זה אנו משתמשים לעדכון ערכי הפרמטרים, בדומה לSGD ללא מומנטום. נוסחת העדכון המתקבלת היא

$$v_t = \beta v_{t-1} + \tilde{\nabla}C_t$$

$$\text{Parameters} = \text{Parameters} - \alpha v_t$$

כאשר  $\beta$  הוא פרמטר המומנטום אשר לרוב נבחר להיות בין 0 ל-1. הביטוי המפורש עבור  $v_t$  הוא

$$v_t = \sum_{k=0}^t \beta^{t-k} \tilde{\nabla}C_k$$

מנוסחה זו אפשר להבין את משמעות הפרמטר  $\beta$ : הוא מבטא את המשקל שהגרדיינטים הקודמים מקבלים בעת עדכון הפרמטרים. כאשר  $1 = \beta$  מתקבל

$$v_t = \sum_{k=0}^t \tilde{\nabla}C_k$$

ומושמות הדבר שככל הגרדיינטים הקודמים עד כה מקבלים משקל שווה בחישוב הצעד הבא של האלגוריתם. כאשר  $0 = \beta$  משקלם של הגרדיינטים הקודמים מתאפס, והאלגוריתם זהה לSGD ללא מומנטום. לרוב נבחר ערך  $0 < \beta < 1$ , וכך גרדיאנטים חדשים יותר יקבלו משקל יתר בחישוב הצעד הנוכחי.

שימוש לב שבדומה לחלקיק הצובר מהירות בתנועתו במורד ההר, כך גם גודל הצעד ב-SGD עם מומנטום עשוי להיות גדול ככל שהגרדיינטים הקודמים נצברים בוקטור המהירות. כמו כן, גרדיאנט אשר חושב באיטרציה  $t$  מושך להשפיע על התנועה גם באיטרציות העוקבות: באיטרציה הבאה יתווסף לווקטור המהירות הערך  $\beta \tilde{\nabla}C_t$ , באיטרציה שאחריה  $\beta^2 \tilde{\nabla}C_t$  וכן הלאה. לבסוף, השפעתו הכוללת של גרדיאנט זה על ערכי הפרמטרים, בהנחה שבוצעו  $T$  איטרציות נוספות, היא

$$\cdot -\alpha \left( \sum_{k=0}^T \beta^k \tilde{\nabla}C_t \right) = \left( -\alpha \sum_{k=0}^T \beta^k \right) \tilde{\nabla}C_t \approx -\frac{\alpha}{1-\beta} \tilde{\nabla}C_t$$

לפיכך, נהוג לבחור את גודל הצעד ההתחלתי, או את מזומנים קבועים של המורה,

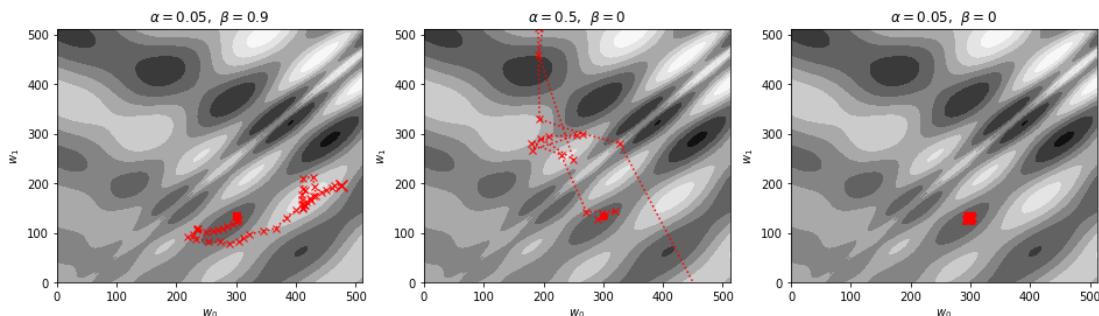
$$\text{האפקטיבי יכול להגיע אף עד } \frac{\alpha}{1-\beta} \text{ ולא רק ל } \alpha.$$

מלבד האצת התוכנות הבאה לידי ביטוי בצעדים גדולים יותר, יתרונות אפשריים של שימוש במומנטום הם הפחתת השפעה של רנדומליות הגרדיאנט האקראי על כיוון התנועה עם התקדמות האלגוריתם, וכן האפשרות לדלג מעל נקודות מינימום מקומיים אם האלגוריתם מגיע אליהן במתינות חיובית.

בחיותו כה נפוץ, אלגוריתם המומנטום מבוסנה בתוך אובייקט האופטימיזציה הבסיסי של PyTorch. כל שיש לעשות הוא להעביר ערך מתאים עבור פרמטר המומנטום, כלהלן.

```
optimizer = torch.optim.SGD(model, lr=alpha, momentum=beta)
```

ראו באյור המצורף דוגמה להשפעה של השימוש במומנטום על ריצת אלגוריתם מורד הגרדיאנט האקראי, כפי שהפעילנו אותו על פונקציית הביצים מהפרק הקודם.

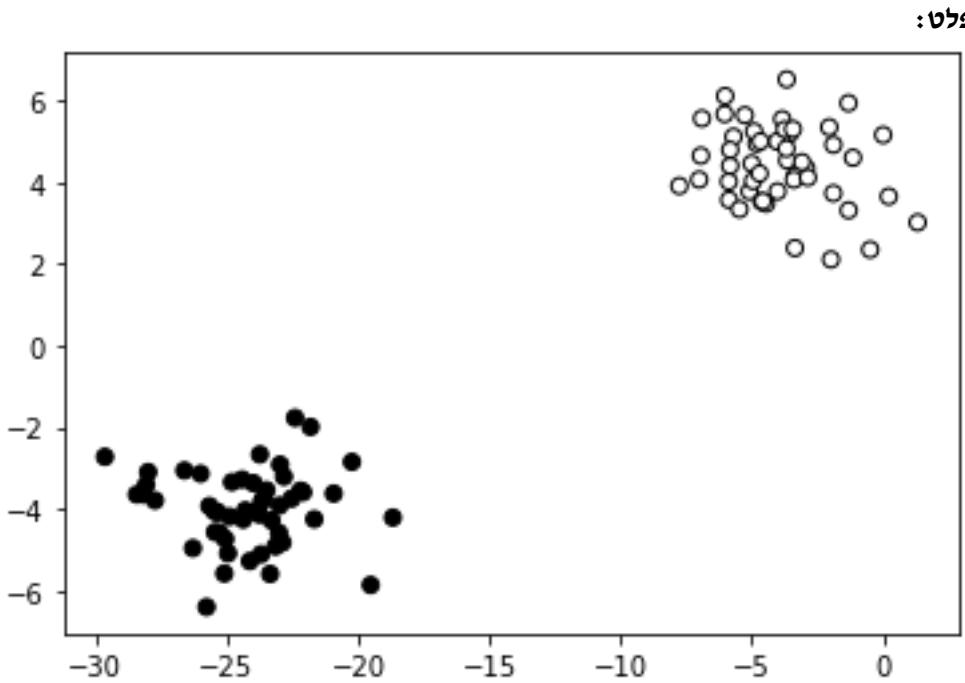


שימוש לבכך שבאיור הימני קצב הלמידה הוא נמוך (וain מומנטום) ובהתחשב לכך האלגוריתם אינו זוז מנקודת ההתחלה, ואילו באյור השמאלי קצב הלמידה ההתחלתי נמוך, אך האלגוריתם צובר תאוצה וגודל הצעד גדול עד אשר לבסוף המומנטום מביא את האלגוריתם אל נקודות המינימום הקרובה. באյור האמצעי ניתן לראות שימוש בקצב הלמידה הגבול של SGD עם מומנטום,  $\frac{\alpha}{1-\beta}$ , ללא המומנטום עצמוו, מוביל להתבדרות.

## קצב למידה אדפטיבי

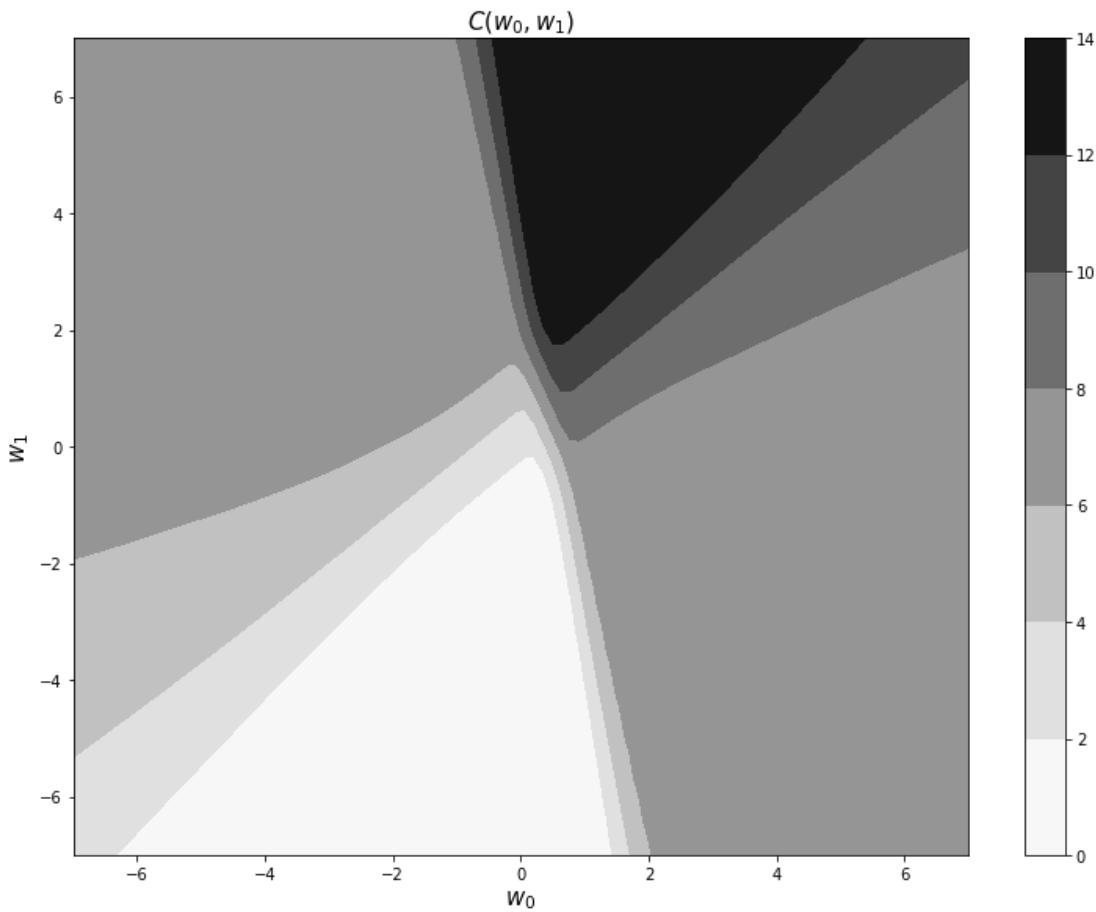
לעתים פונקציית המחיר תליה בכל אחד מהפרמטרים של המודל בצורה שונה – שינוי קטן באחד ישפייע מאוד על ערכה, בעודו שינוי קטן לאחר תהיה השפעה פחותה. מצב זה, שבו הפרמטרים בסדרי גודל שונים, יוצר קשיים באופטימיזציה, ולצערנו נפגש בו לא פעם. נוכל לדמות במקרה פשוט ששיעור אחד מושפע ממספר פרמטרים אחרים, וכך השינויים יתפזרו על כל פרמטר. נזכיר את הנקודות כפי שעשינו בעבר, אך לאחר מכן – נכפול את אחד הממדים בקבוע גדול.

```
import sklearn.datasets as skds
X, Y = skds.make_blobs(n_samples=100, n_features=2,
                       centers=2, random_state=1)
X, Y = torch.tensor(X), torch.tensor(Y)
X[:, 0] = 2.5*X[:, 0]
plt.scatter(X[:, 0], X[:, 1],
            c=Y, cmap="Greys", edgecolor="black");
```

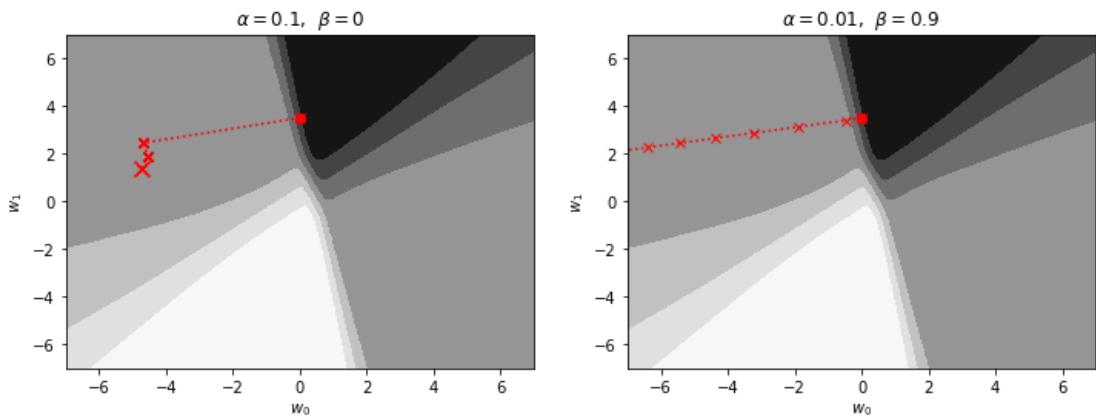


ראו כתת השינוי בציר ה- $x$ : השינויים בו גודלה במידה ניכרת מבעבר.

נזכיר שבמודל זה שלושה פרמטרים,  $w_0, w_1, b$ . כמו בדיון הקודם, נקבע את ערכו של  $b$  להיות 5 ונציר את משטח המחיר התלויה בשני הפרמטרים האחרים.



מайור זה ניכרת ההשפעה של כפל הממד הראשון בקבוע גדול על פונקציית המחיר – היא רגישה הרבה יותר לשינויים בפרמטר  $w_0$  מאשר ב- $w_1$ . לפיכך, הנזורה  $\frac{\partial C}{\partial w_1}$  תהיה גדולה הרבה יותר מאשר  $\frac{\partial C}{\partial w_0}$  וכיום התנווה של אלגוריתם מורד הגרדיאנט, לכל גודל צעדי, יהיה כמעט אופקי. גם שימוש במומנטום לא יעזור במקרה זה, כפי שניתן לראות באירוע:



אחד הפתרונות לבעה זו הוא שימוש בגודלי צעד שונים עבור הפרמטרים השונים: פרמטרים רבים השפעה יעדכנו בעדינות ובצעדים קטנים, ופרמטרים המשפיעים פחות יעדכנו בעדדים גדולים יותר. רעיון זה עומד מאחורי האלגוריתמים Adagrad ו-RMSprop, ולבסוף בא לידי ביטוי באלגוריתם

הפולורי Adam, המשלב **גודל צעד אדפטיבי לכל פרמטר** יחד עם מומנטום. בـ Adam אנו שומרים, בנוסף על וקטור המהירות, גם וקטור סקלה המתעדכן בדומה, ומשתמשים בו כדי לנормל את עוצמת התנועה עבור כל פרמטר בנפרד. נוסחת עדכון הפרמטרים היא אפוא

$$\begin{aligned} v_t &= \beta_1 v_{t-1} + (1 - \beta_1) (\tilde{\nabla} C_t) \\ s_t &= \beta_2 s_{t-1} + (1 - \beta_2) (\tilde{\nabla} C_t)^2 \\ \text{Parameters} &= \text{Parameters} - \alpha \frac{v_t}{\sqrt{s_t}} \end{aligned}$$

כאן יש לשים לב חישוב  $(\tilde{\nabla} C_t)^2$  מותבצע באמצעות הعلاה בריבוע איבר-איבר של הווקטור  $\tilde{\nabla} C_t$ . התוצאה המתקבלת היא שפרמטרים שליהם נזירות קטנות יעדכנו בקצב מהיר יותר. לאלגוריתם שלושה פרמטרים אשר יש לקבוע לפני ריצתו:  $\beta_1$ , המקביל במשמעותו לפרמטר המומנטום המקורי,  $\beta_2$ , אשר משמש כפרמטר זיכרון של וקטור הסקלה, ו-  $\alpha$ , קצב הלמידה המוכר לנו. חשוב לזכור את הדעת לכך שהזהו קצב הלמידה **האפקטיבי**, בשונה מ-SGD עם מומנטום, שכן וקטור העדכון של הפרמטרים עובר נרמול בכל איטרציה, ולכן המהירות הנצברת "מקוזצת" עם הנסיבות גורם הנרמול.

נוסחת העדכון הנ"ל שימושית להבנת תפקוד האלגוריתם, אך במקרה שיש לה חסרונות אחדים: ראשית, תיתכן חלוקה באפס בעת הנרמול, וכן מוסיפים למכנה מספר קטן; שנית, נהוג לבחור פרמטרים  $\beta_1, \beta_2$  הקרובים ל-1, ולפיכך באיטרציות הראשונות ערכי  $v_t$  ו-  $s_t$  הם קטנים מאוד. כדי

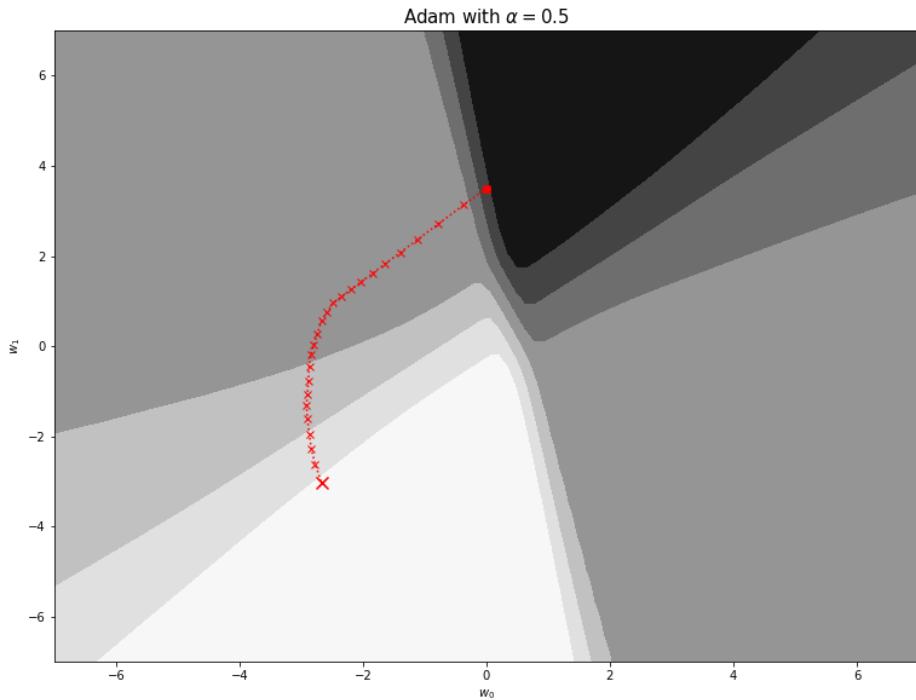
להתגבר על בעיה זו מנוטמים את הווקטורים בכל איטרציה, ונוסחת העדכון המתבלט היא

$$\begin{aligned} \hat{v}_t &= \frac{v_t}{1 - (\beta_1)^t}, \quad \hat{s}_t = \frac{s_t}{1 - (\beta_2)^t} \\ \text{Parameters} &= \text{Parameters} - \alpha \frac{\hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon} \end{aligned}$$

השימוש בـ Adam בـ PyTorch הוא פשוט. כל שיש לעשות הוא להחליפ את אובייקט האופטימיזציה, **כדלהלן**.

```
optimizer = torch.optim.Adam(model, lr=alpha)
```

תוצאת הפעלה של Adam על הדוגמה الأخيرة מאיירת להלן, וניתן לראות בבירור את היתרונו של השיטה על פני הקודמות.



נסים פרק זה בסיג' חשוב: ישן מעט תוצאות תיאורטיות המבטיחות את התכונות האלגוריתמיים למיניהם מקומי של פונקציית המחיר, ועוד פחות מכך תוצאות המאפשרות לדעת מראש איזה אלגוריתם עדיף על פני אחר. המצב חמור עוד יותר בהקשר של רשותות נוירונים עמוקות, שכן התוצאות הקיימות אינן תקפות לפונקציית המחיר של להונ. לפיכך, עיקר הידע בבחירה האלגוריתם הנכון מctrבר בדרך של ניסוי וטעייה. בהמשך הלימוד, נועל לא פעם לפי כללי אכבע מקובלים אשר אינם מוגבלים בחוכחה תיאורטית, אך ישנו קונצנזוס מקצועי הtoutmc באפקטיביות שלהם. למשל נבחר את הפרמטרים של Adam להיות  $\beta_2 = 0.99$ ,  $\beta_1 = 0.9$ .

## שאלות לתרגול

- הניחו שפונקציית המחיר היא לינארית בפרמטרים, קרי  $C(w_0, \dots, w_n) = \sum_{k=0}^n c_k w_k + b_c$ , כאשר  $(w_0, \dots, w_n)$  הם הפרמטרים של המודל, וכן שאלגוריתם מורד הגרדיאנט עם מומנטום מאוחת בנקודה  $(0, \dots, 0)$ .  
  - מצאו ביטוי פשוט לערכי הפרמטרים לאחר האיטרציה ה- $t$  יא.
  - חשבו את המרחק בין ערכי הפרמטרים בין שתי איטרציות.
  - האם תוכלו להסביר מדוע קצב הלמידה הגבולי של SGD עם מומנטום הוא (במקרים מסוימים)

$$\frac{\alpha}{1-\beta}$$

**רמז:** השתמשו בנוסחת הסכום של טור הנדסי.

- חוירו לדוגמה الأخيرة שבה הראיינו שי-Adam עדיף על SGD ( בלי או עם מומנטום) ושנו את נקודת ההתחלה של האלגוריתמים, קצב הלמידה ואת פרמטר המומנטום. מצאו שילוב ערכיים שבהם ביצעוו של Adam פחותים.

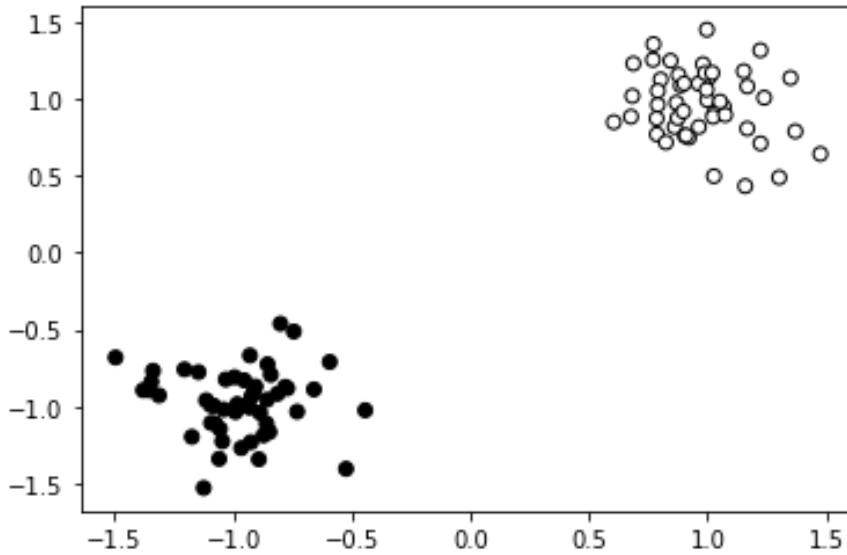
## שכבות נורמליזציה

בסוף הפרק הקודם יצרנו באופן מלאכותי בעיה בסט הנתונים באמצעות שינוי הסקלה של אחד ממשתני הקלט לרשות. הבעיה זו יש לפתור פשוט הרבה יותר מהחלפת אלגוריתם האופטימיזציה, והוא תקנון אוסף הנתונים לפני הזנתו לרשות: לאחר ייצור הנΚודות נחסיר מכל ממד את הממוצע שלו ונחלק בסטיית התקן, כלהלן.

```
X = (X-X.mean(dim=0)) / X.std(dim=0)
print(X.mean(dim=0), X.std(dim=0), sep='\n')
plt.scatter(X[:, 0], X[:, 1],
            c=Y, cmap="Greys", edgecolor="black");
```

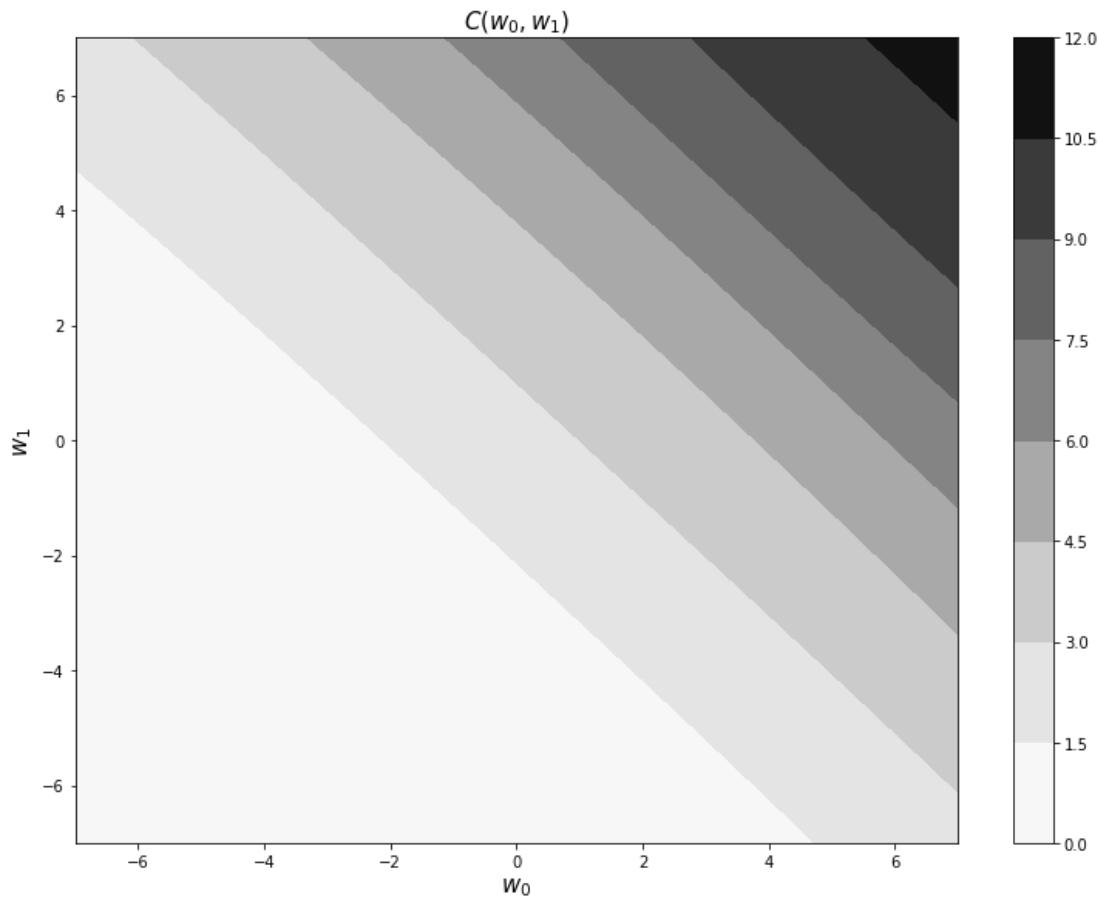
**פלט:**

```
tensor([-3.5527e-17,  6.6613e-18], dtype=torch.float64)
tensor([1.0000, 1.0000], dtype=torch.float64)
```



זכרו שהממוצע וסטיית התקן הן מתודות רדוקציה, והפרמטר `dim` מנהה אותן לאורך איזה ממד יש לבצע את הפעולה.

כעת, אם נסתכל על משטח המחיר של ניירון הסיווג כפונקציה של הפרמטרים  $w_0, w_1$  (שוב קבוע  $b=5$ ), נראה את התוצאה הזו:



מайור זה ניכר שכל אלגוריתם אופטימיזציה יתמודד בהצלחה עם פונקציית מחיר זו, שכן הגרדיינט שלה מצביע לאוטו כיון בכל נקודה. כאן מתחילה ומסתיימת העבודה כאשר מדובר בנירון יחיד, אך עבור רשות עמוקה הקלט לרשות הוא הקלט לשכבה הראשונה בלבד. לאחר מעבר הקלט לשכבה זו סביר שהפלט שלו, שהוא הקלט לשכבה הבאה, כבר לא יהיה מתוקן; וביחוד לאחר כמה שכבות. שיקול זה הוביל לתוספת של שכבות נורמליזציה (normalization layers) לרשתות נירוניים מודרניות, דבר אשר ייעיל במידה ניכרת את תהליך אימון הרשות, בין השאר משום שהוא מאפשר שימוש בקצב מידע מוגבר.

הדוגמה הראשונה והנפוצה ביותר לשכבה נורמליזציה היא BN (Batch Normalization), אשר מקבלת ה קלט מהשכבה הקודמת ברשות טנזור  $X = (x_1, \dots, x_d)$  ועbor קואורדינטה כלשהי,  $x_i$ , בטנזור זה היא מבצעת את החישוב בשני שלבים :

1. ראשית, מחושבים הממוצע והשונות של  $x_i$  על פני **הbatch הנומי** (מכאן מגע שמו השכבה), ובעזרתם מתקנים את הערך של  $x_i$  :

$$\mu = \frac{1}{N} \sum_{k=1}^N x_i^{[k]}$$

$$\sigma^2 = \frac{1}{N} \sum_{k=1}^N (x_i^{[k]} - \mu)^2$$

$$\hat{x}_i^{[k]} = \frac{x_i^{[k]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

שימוש לב ש כדי להציג את תלות החישוב ב- $\hat{x}$ , העותקים השונים של  $\hat{x}$  אשר חושבו בר-זמנית על סמך הקלטים השונים ב- $\hat{x}$ , נתונים בסוגרים מרווחים. את גודל ה- $\hat{x}$  אנו מסמנים ב- $N$  וכרגיל, למכנה אנו מוסיפים ערך קטן כדי להימנע מחולקה באפס. לרוב נהוג לבחרו  $\epsilon = 10^{-5}$ .

2. שנית, ייתכן שאקטיציות מתקנות אינן בהכרח אופטימליות עבור המשך הרשת, ולכן הפלט הסופי של השכבה הוא

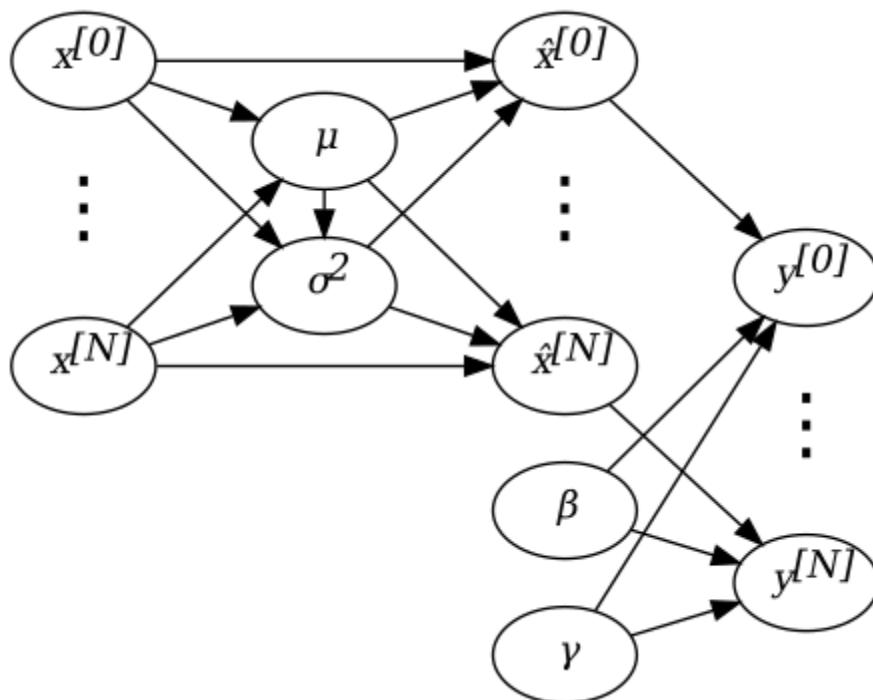
$$y_i^{[k]} = \gamma_i \hat{x}_i^{[k]} + \beta_i$$

כאשר  $\gamma_i, \beta_i$  הם פרמטרים אשר יילמדו בתהליך האימון, אך מאותחלים כך שראשית מתקיים  $y_i = \hat{x}_i$ .

כמו כן, יש לשים לב פרטיים האלה :

1. החישוב בשכבה BN מתבצע בויזמנית ובאופן בלתי תלוי עבור כל קואורדינטה של  $X$ .
2. לכל קואורדינטה של  $X$  יש שני פרמטרים שונים בשכבה, אשר נלמדים ללא תלות באחרים.
3. השכבה יוצרת תלות בין הדגימות השונות אשר נמצאות ב- $\hat{x}$ , ולפיכך הפלט שלו עבור קלט כלשהו יהיה תלוי בשאר הקלטים הנמצאים ב- $\hat{x}$  שלו.

שלבי החישוב בשכבה BN מאוירים להלן,



נמשש שכבה זו ב-`PyTorch`, ודרך תהליך זה נלמד כיצד לכתוב **שכבות חדשות**. המחלקה הבסיסית של שכבת רשת נוירונים היא `nn.Module`, והשכבה שניצור תירש ממנה. ובשורט קוד :

```
class BN(nn.Module):
```

כעת, צריך להגדיר את הבניי של השכבה שלנו. הוא יקרא לבניי של `Module`.`nn`, ואחרי כן יאותל את הפרמטרים  $\beta$ ,  $\gamma$ , אשר מהם יש לנו עותק לכל קווארדיינטה של טנזור הקלט. הבניי מקבל את גודל הקלט לשכבה בפרמטר `in_features`, בדומה לשכבות הליינאריות שהשתמשנו בהן בעבר.

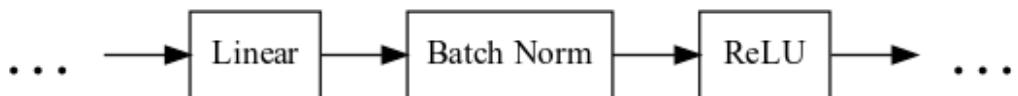
```
class BN(nn.Module):
    def __init__(self,in_features):
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(in_features))
        self.beta = nn.Parameter(torch.zeros(in_features))
```

אנו משתמשים בפונקציה `nn` כדי להציג על משתנים אלו כפרמטרים של הרשת, כך שאובייקט האופטימיזציה ידע לעדכן אותם בתהליכי אימון הרשת.

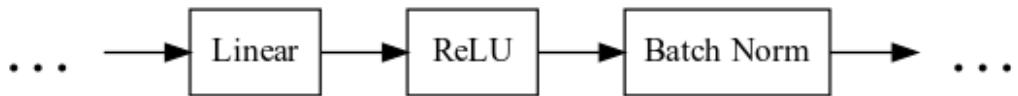
אחרי כן, כל שנוטר לעשות הוא להגדיר את פלט השכבה – מחשבים אותו במתודה `forward()`. ראו להלן שהממוצע והשונות מחושבים לאורך הממד 0 של טנзор הקלט. זה ממד ה-`batch` באופן מוקובל, וגם בכל הרשותות שמיישנו עד כה. שאר החישוב מתבצע בעזרת שידור לאורך כל הדגימות `input` בנטו `minibatch` במשתנה `batchnorm`.

```
def forward(self,input):
    mu      = input.mean(dim=0)
    sigma2 = input.var(dim=0)
    xhat   = (input-mu) / torch.sqrt((sigma2)+10**-5)
    y      = self.gamma*xhat + self.beta
    return y
```

לאחר שכבה זו הוגדרה, ניתן לשולבה ברשת נוירונים ככל שכבה מובנית של `nn`.`torch`.`nn`. נחוג לשלב שכבה זו בין שכבה ליינארית לאקטיבציה, כמוואר להלן,



או מייד לאחר האקטיבציה,



אין לנו צורך למש את ההתפשטות לאחר דרכ שכבת זו, שכן מערכת `autograd` עוקבת אחרי כל החישובים המבוצעים בהתרפשטות פנים, ועל כן ההתפשטות לאחר תtabcvau אוטומטית. עם זאת, יש ערך בחישוב האנלייטי של ההתפשטות לאחר: דרכו ניוכח שחישוב היוצר תלויות בין דגימות שונות מאוסף הנתונים אליו בהכרח בעיתוי מבחינת אלגוריתם האופטימיזציה. ניגש אפוא למשימה. פלט

השכבה הוא  $Y$  וממד טנוור הקלט  $X$ . נניח שבידינו גראדיאנט פונקציית המחיר לפי הפלט,

ובעת עליינו לחשב את  $\frac{\partial Y}{\partial \beta}$  בשליל עדכו ערכי הפרמטרים של שכבה זו, וכן את  $\frac{\partial Y}{\partial \gamma}$  כדי לחשב

לאחר את הנזירות בשכבות קודומות לשכבה ה- $BN$ . ראשית נשים לב שכל הנזירות מהצורה

$$\frac{\partial y_i^{[k]}}{\partial \beta_m}, \frac{\partial y_i^{[k]}}{\partial \gamma_m}, \frac{\partial y_i^{[k]}}{\partial x_m^{[n]}}$$

כאשר  $m \neq i$  מטאפסות, שכן עבור כל קווארדיינטה של הקלט  $X$  מתבצע חישוב בלתי תלוי בקווארדיינטות האחרות. שנית, באופן מיידי לפי הנוסחה  $y_i^{[k]} = \gamma_i \hat{x}_i^{[k]} + \beta_i$  מתקיים

$$\begin{aligned} \frac{\partial y_i^{[k]}}{\partial \beta_i} &= 1 \\ \cdot \frac{\partial y_i^{[k]}}{\partial \gamma_i} &= \hat{x}_i^{[k]} \end{aligned}$$

כעת, נחשב את הנזירות המיידיות של כל קודקוד לפי כל אב מיידי שלו בגרף החישוב לעיל:

$$\begin{aligned} \hat{x}_i^{[k]} : \frac{\partial y_i^{[k]}}{\partial \hat{x}_i^{[k]}} &= \gamma_i \\ \sigma^2 : \frac{\partial \hat{x}_i^{[k]}}{\partial \sigma^2} &= (x_i^{[k]} - \mu) \cdot \frac{-1}{2} (\sigma^2 + \epsilon)^{-3/2} \\ \mu : \frac{\partial \hat{x}_i^{[k]}}{\partial \mu} &= \frac{-1}{\sqrt{\sigma^2 + \epsilon}}, \quad \frac{\partial \sigma^2}{\partial \mu} = \frac{1}{N} \sum_{k=1}^N -2(x_i^{[k]} - \mu) \\ x_i^{[k]} : \frac{\partial \hat{x}_i^{[k]}}{\partial x_i^{[k]}} &= \frac{1}{\sqrt{\sigma^2 + \epsilon}}, \quad \frac{\partial \sigma^2}{\partial x_i^{[k]}} = \frac{2(x_i^{[k]} - \mu)}{N}, \quad \frac{\partial \mu}{\partial x_i^{[k]}} = \frac{1}{N} \end{aligned}$$

כל שנותר לעשות הוא לחבר את הנזירות המיידיות בעזרת שימוש זהיר בכל השרשרת – יש לגזור אב מיידי לפי כל אחד מבניו, ולהצטרף לתוצאה. למשל עבור  $x_i^{[k]}$ ,

$$\cdot \frac{\partial C}{\partial x_i^{[k]}} = \frac{\partial C}{\partial \mu} \frac{\partial \mu}{\partial x_i^{[k]}} + \frac{\partial C}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial x_i^{[k]}} + \frac{\partial C}{\partial \hat{x}_i^{[k]}} \frac{\partial \hat{x}_i^{[k]}}{\partial x_i^{[k]}}$$

התלות הנוצרת בין הדגימות השונות ב- $batch$  מועילה למטרת האופטימיזציה, אך אינה רצויה כאשר הרשת כבר אומנה: אנו לא מעוניינים שהחיזוי עבור דוגימה נתונה ישנה בהתאם לשאר הדגימות המוזנות בד בבד עימה ברשות. לכן בעת החיזוי נשתמש בהערכות אלטרנטטיביות עבור  $\mu$  ו- $\sigma^2$ . התבוננו שוב בגרף החישוב שלעיל והיווכחו שהדגימות השונות ב- $batch$  תלויות זו בזו רק דרך ערכים אלו. מכאן שנייהוק התלות שלהם ב- $batch$  יוביל לחישוב דטרמיניסטי: החישוב עבור כל דוגמה יהיה בלתי תלוי בדגימות אחרות. אחת הדרכים המקובלות לעשות זאת היא לשמור בעת האימון ממוצע רץ של ערכים

אלו, בדומה לנעשה בשיטת המומנטום עבר וקטור המהירות. כאשר נרצה לעבור לחיזוי, נציב ממוצעים אלו במקומם של  $\sigma^2$ , מ. שאר החישוב יבוצע ללא שינוי.

שכבה BN היא הרכיב הראשון של רשותות ניירוניים שאנו פוגשים שהתנהגותו שונה בעת אימון ובעת חיזוי, ונכיר עוד רכיבים כאלו בהמשך הלימוד ונשתמש בהם לרוב. כדי להקל על המעבר בין אימון לחריזוי, קיימות ב PyTorch המתודות `model.train()` ו `model.eval()`, אשר מעבירות את הרשות בין המצביעים. אם ברצוננו להתנות את החישוב המתבצע בשכבה מסוימת בהיותה במצב אימון/חריזוי, נוסיף תנאי זה למתודות החישוב, כדלקמן.

```
def forward(self, input):
    if self.training:
        .
        .
        .
    else:
        .
        .
        .
    .
    .
    .

```

אינה שכבה הנורמליזציה היחידה בשימוש נפוץ, וישנו נוספת כגון layer normalization או group normalization שמשתמשים בהן בקובאורדייניות השונות (בכלן, או בחלקן, כתלות בשיטה) בטנוור הקלט לשכבה כדי לנרמל את האקטיבציות. מכיוון ששיטות אלו אין יוצרות תלות בין דוגמאות שונות בbatch, הן לרוב עדיפות על BN כאשר batch קטן: במקרה זה הפרמטרים  $\mu$  ו  $\sigma^2$  של שכבה BN המוחשיים בכל איטרציה אינם יציבים – ערכם משתנה בהתאם למעט הדוגמאות אשר נמצאות באותה העת בזיכרונו. השיטות הנפוצות מומשו שכבות סטנדרטיות ב `.torch`.

אין ספק בקרב הקהילה המדעית שכבות הנורמליזציה חיוניות לאימון רשות בצורה אפקטיבית, אולם הסיבות לכך הן נושא לדין ערך. יתכן בהחלטה שהאינטואיציה המקורית שהנחתה את פיתוחן, קרי נרמול הקלט בכל שכבה ברשות, אינה הסיבה העיקרית לכך. יתכן שבדריכים אחרות, שעדיין אין ידועות לנו, שכבות אלו הופכות את משטח המחיר לפשטוט יותר עבור אלגוריתמי אופטימיזציה מבוססי גרדיאנט, שכן המחקר בנושא ממשיך להתפתח גם כיום.

## שאלות לתרגול

1. א. צרו רשות عمוקה המשרתת שכבות לינאריות ושכבות ReLU בזו אחר זו, כך שהקלט שלה הוא דומם, ואנו אותה לסוג אוסף נתונים הנוצר בעזרת הפונקציה `.make_moons`
- ב. הזינו לתוכה את נתוני הנקודות השחורות והלבנות לאחר נרמול, שכבה אחר שכבה, ושמרו את **הפלט של כל שכבה** במשתנה חדש.
- ג. חשבו את הממוצע ואת סטיית התקן של הפלט של כל זוג שכבות  $\text{ReLU} \rightarrow \text{Linear}$ . שימו לב שיש לבצע את הבדיקה על ממד `batch`.

- ד. ציירו את התוצאה בגרפים: בציר ה- $x$  – מיקום השכבה בראש. בציר ה- $y$  – התוחלת/סטיית התקן של כל ניירון בשכבה זו. **הערה:** מומלץ להשתמש למטרה זו בפקודה `boxplot` של הספרייה `matplotlib`.
2. חזרו על שאלה 1 לאחר הוספת שכבות נורמליזציה לרשף הנוירוניים. זכרו להעביר את הרשות למצב חיזוי לאחר סעיף א.
3. השלימו את החישוב בתפניות לאחורי דרך שכבת BN: בטאו את  $\frac{\partial C}{\partial x_i^{[k]}}$  בעזרת ערכים שחושו בתפניות לפנים בראש ובעזרת  $\frac{\partial C}{\partial y_i^{[k]}}$  בלבד. פשטו את הביטוי ככל האפשר.
4. הוסיפו למחלקה BN המוגדרת לעיל מצב חיזוי כدلקמן:
- א. שמרו משתנים פרטיים בתחום האובייקט עבור הממצאים הרצים של  $\sigma^2, \mu$ .
  - ב. בכל התפניות לפנים בשכבה, כאשר מצב אימון מופעל, יש לעדכן את הממוצע של  $\mu$  לפי הנוסחה  $\mu = 0.9\mu_{avg} + 0.1\sigma^2$ , כאשר  $\mu$  חושב על בסיס ה-`batch` הנוכחי. בדומה לכך יש לעדכן את  $\sigma^2$ . שימושם לב משתנים אלו לא משתנים בתהליכי האימון, ולכן אין צורך לעקוב אחריהם באמצעות מנגנון `autograd`.
  - ג. כאשר מצב חיזוי מופעל, יש להשתמש ב- $\sigma_{avg}^2, \mu_{avg}$  במקום  $\sigma^2, \mu$ .
5. שימוש ב-BN דורש `minibatch` בגודל שתי דוגמאות לפחות. הסבירו דרישת זו.

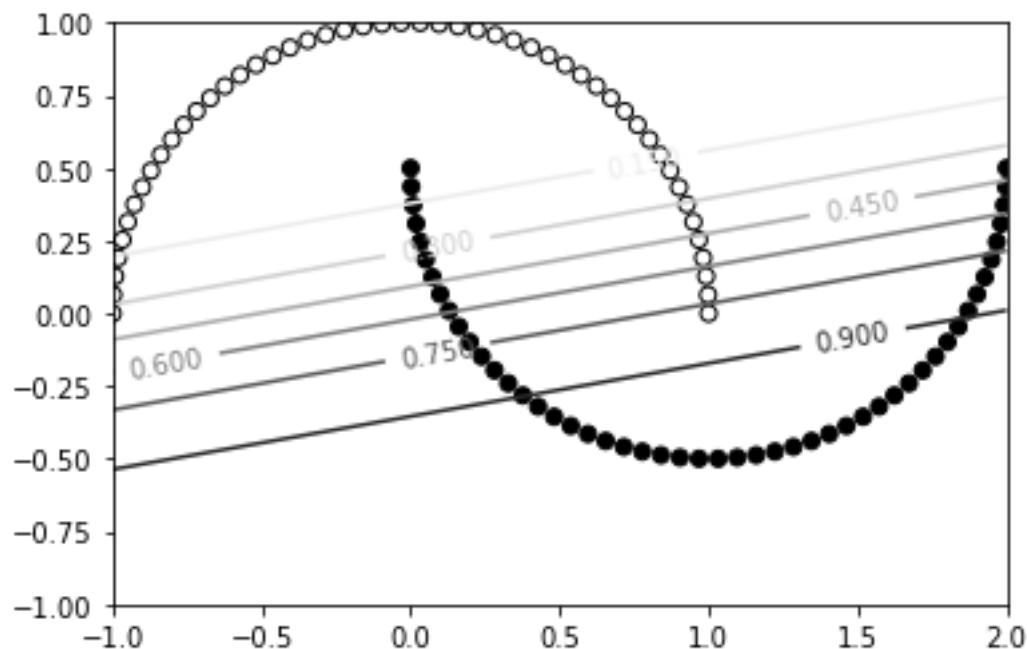
## ATCHOL פרמטרים

עד כה, כאשר הגדכנו מודל רשת ניורונים בעזרת PyTorch, לא הקדשו מחשבה לערכי הפרמטרים הראשוניים: הם אוחתלו אקרואית בעת הגדרת אובייקט הרשות, ושימוש נקודת התחליה עבור אלגוריתם האופטימיזציה הנבחר. עם זאת, כמו שכיוון התנועה וגודל הצעד של האלגוריתם חשובים, כך גם לנקודת התחליה השפעה מכרעת. בחרה שגואה של שיטת אתחול פרמטרים עלולה להוביל לכמה בעיות.

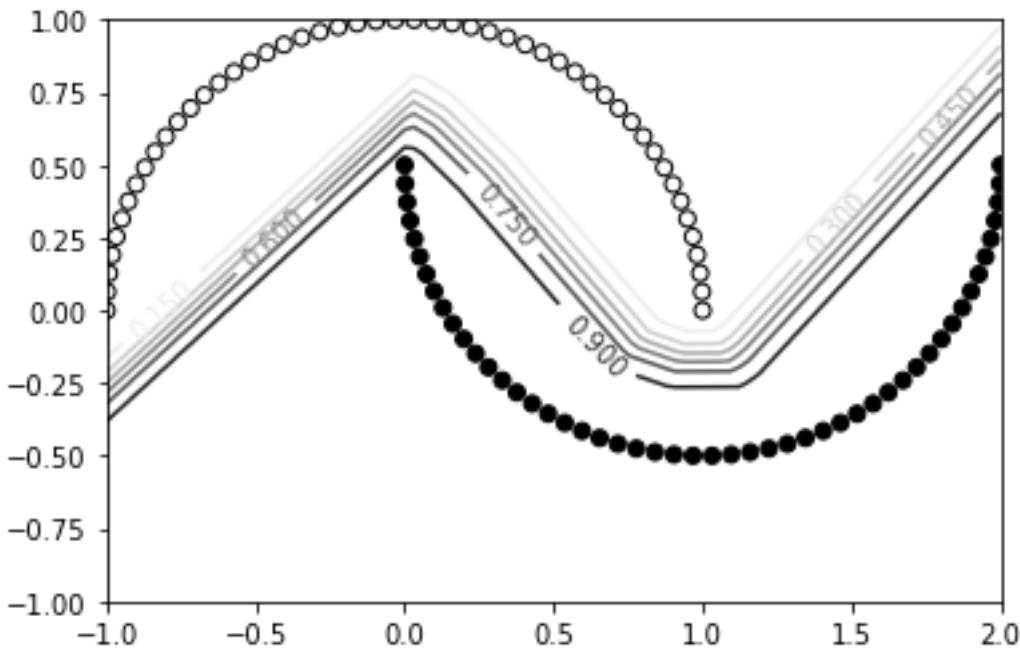
ראשית, נשים לב שאთחול פרמטרים זהה עברו שני ניורונים באותו שכבה יוביל לכך שהם יבצעו את אותו החישוב, ובהתפשטות לאחרו של השגיאה, נזירות פונקציית המחרך לפי פרמטרים אלו יהיו אף הן זהות. התוצאה של אתחול כזו תהיה שני הניורונים יעדכו בצורה זהה באמצעות אלגוריתם האופטימיזציה, וגם באיטרציות הבאות ימשיכו לבצע את אותו החישוב. כתוצאה זאת הרשות תאבד יכולת החישוב שלה. ראו בדוגמה שלහן כיצד אנו מגדירים רשת עמוקה ומתחלים אותה בערכי פרמטרים קבועים.

```
model = nn.Sequential(nn.Linear(2,10),
                      nn.ReLU(),
                      nn.Linear(10,10),
                      nn.ReLU(),
                      nn.Linear(10,1),
                      nn.Sigmoid())
for x in model.parameters():
    torch.nn.init.constant_(x, 0.1)
```

אם נאמנו רשות זו לסוג נקודות שחורות ולבנות (אשר יצרנו בעזרת הפונקציה `make_moons`, תתקבל התוצאה שלහן עברו הסתברויות הסיווג (נזכיר שההסתברות שהמודל חוצה לכך שנקודה נתונה היא שחורה מופיעה על הקוו העובר דרךה).



מайיר זה ניכר שבעוד הרשות עמוקה ובעל מספר רב של נוירונים, אותה תוצאה הייתה מתקבלת גם במודל סיווג של נוירון יחיד. לשם השוואה, התוצאה המתבקשת מהaimon לאחר שימוש בפרמטרים המאוחתלים לפי בריית מחדל בעת הגדרת הרשות מאiorת להלן, ומайיר זה ניכר ש모רכבות הרשות באה לידי ביטוי בסיווג מוצלח יותר.



על כן, הדרישה הראשונה שלנו מאסטרטגיית האתחול היא "שברית סימטריה" (symmetry breaking) – נרצה שהפרמטרים יאותחלו כך שכל נוירון יבצע חישוב אחר, ובהתאם לכך יתעדכן כל נוירון באופן שונה עם איטרציות אלגוריתם האופטימיזציה. כדי לעמוד בדרישה זו, נאותחל את הפרמטרים באקראי.

אתחול אקראי אינו הפתרון לכל הבעיה, שכן פרמטרים קטנים מדי יובילו לערכי נזירות קטנים מדי, ואלגוריתם האופטימיזציה לא יוזז מנקודת ההתחלה. תופעה זו נקראת "גרדיאנט נעלם" (vanishing gradient) ונפוגש בה בהמשך הלימוד, גם בהקשר של ארכיטקטורות רשת מסויימות. וקיים גם הבעיה ההפוכה: אתחול ערכי פרמטרים גדולים מדי (במערכות המוחלט) יוביל לערכי נזירות גדולים מדי והאלגוריתם יתבדר. תופעה זו מכונה "גרדיאנט מתפוצץ" (exploding gradient). לשתי בעיות אלו ישנו כמה פתרונות, ומדובר לאו דווקא רק באתחול פרמטרים שגוי. עם זאת, בחירת סדר גודל נכון כבאים לפרמטרים ההתחלתיים מאפשרת לעיתים להתחמק ממנה. הכללים המקובלים בשימוש נפוץ קובעים את השונות של מוחלט המספריים האקראיים עבור הפרמטרים של שכבה מסוימת, כך שבז'זמנית בהתפשטות לפנים (בעת הזנות הנתונות בראש) ובהתפשטות לאחר (בעת חישוב הגרדיאנט), הערכים המוחושבים יישמרו באותו סדר גודל. הקו המנחה מאחרוי שיטות אלו הוא שכך לשכבה מסוימת יש יותר משני קלט ופלט, כך יש לאותחל את הפרמטרים באמצעות דגימת התפלגות עם שונות קטנה יותר.

שיקולים אלו הובאו בחשבון בעת תכנון הבנאים של המודולים המובנים ב-`PyTorch`: לכל מודול או שכבה בעלי פרמטרים יש מתודה בשם `reset_parameters()` אשר מאותחלת את הפרמטרים לפי

הכלל המתאים. למשל בשכבה לינארית Linear. nn. **יאוначלו הפרמטרים באקראי** לפי התפלגות אחידה בקטע  $\left[ -\frac{1}{\sqrt{K_{in}}}, \frac{1}{\sqrt{K_{in}}} \right]$ , כאשר  $K_{in}$  הוא ממד הקלט של השכבה.

## שאלה לתרגול

1. א. **יאוначלו את הפרמטרים** של הרשת המוגדרת לעיל **לפי שיטת Xavier** : הפרמטרים **יאוначלו באקראי** לפי התפלגות אחידה בקטע  $\left[ -\frac{\sqrt{6}}{\sqrt{K_{in} + K_{out}}}, \frac{\sqrt{6}}{\sqrt{K_{in} + K_{out}}} \right]$ , כאשר  $K_{out}$  הוא ממד פלט השכבה, ו-  $K_{in}$  הוא ממד הקלט. **רמז:** השתמשו בפונקציה `torch.nn.init.uniform_`
- ב. **אמנו את הרשת** לסוג את הנקודות השחורות והלבנות אשר נוצרו מישימוש בפונקציה `.make_moons`.
- ג. **השו את התוצאות** לרשת המאותחלת **לפי** בירית המחדל.

## ICHIDE 4: רגוליזציה

### רשותות ניירונים לרגרסיה

בכל הדוגמאות לרשותות ניירונים שעשכנו בהן ביחידות הקודמות, פلت הרשות היה הסתברויות חזויות, ומטרתנו הייתה לפתור בעיתת סיוג לכמה מחלקות. אולם אין להסיק מכך לרשותות ניירונים שימושיות למשימות אלו בלבד. למעשה, ניתן להשתמש בהן לפתרון מספר רב של משימות, כגון זיהוי אובייקטים בתמונה, דחיסת מידע, תרגום אוטומטי של קטעי טקסט ועוד רבות אחרות. למטרות שונות קיימות ארכיטקטורות רשות אפקטיביות מיוחד, ועל חלון נלמד בהמשך הקורס, אולם כל שנדרש באופן בסיסי כדי לאמן רשות לביצוע משימה שאינה סיוג למחלקות הוא להחליף את "ראש" הרשות, השכבה האחורה, ואת פונקציית המחיר. בפרק זה נראה דוגמה כיצד בעזרת שינויים קלים אלו יוכלתנו לפתור בעיתת רגרסיה.

נניח כי ברשותנו סט אימון של זוגות מספרים מהצורה  $(x_0, y_0), (x_1, y_1), \dots, (x_N, y_N)$ , ומטרתנו היא לאתר את הקשר הפונקציוני בין  $x$  לבין  $y$ , כלומר את ה"כלל" המחבר בין שני מספרים אלו, כפי שבא לידי ביטוי בנקודות. לצורך המasha, נניח בהמשך הדוגמה שהכלל המקשר בין שני המשתנים הוא

$$y(x) = x + \sin(x) + \cos(2x)$$

אך איסוף נתונים האימון התבצע בתהליך "ירועש", ולכן הנקודות שנדגמו אינן מקיימות כלל זה בדיקות, אלא רק בקירוב. ראו יישום רענון זה בקטע הקוד המצורף.

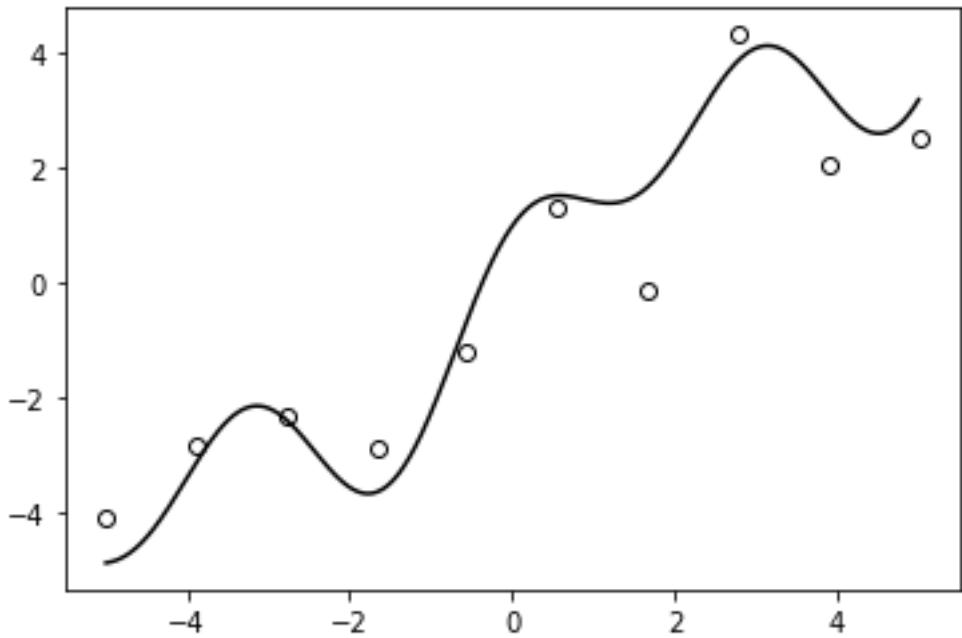
```

fx= lambda x:torch.sin(x)+torch.cos(2*x) + x
X = torch.linspace(-5,5,1000).resize(1000,1)
Y = fx(X)

Xtrain = torch.linspace(-5,5,10).reshape(10, 1)
torch.manual_seed(1)
Ytrain = fx(Xtrain)+1.2*torch.randn(size=Xtrain.size())

plt.plot(X, Y, color='black');
plt.scatter(Xtrain, Ytrain, color="white", edgecolor="black");
plt:

```



הקו השחור מייצג את הקשר המתמטי המדוייק, בעוד הנקודות הלבנות מייצגות את אוסף הנתונים אשר נדגם, בתוספת הרעש. זכרו שבשימושיםמציאותיים לא נוכל לציר את הקו השחור – הקשר בין המשתנים לא יהיה ידוע לנו מראש.

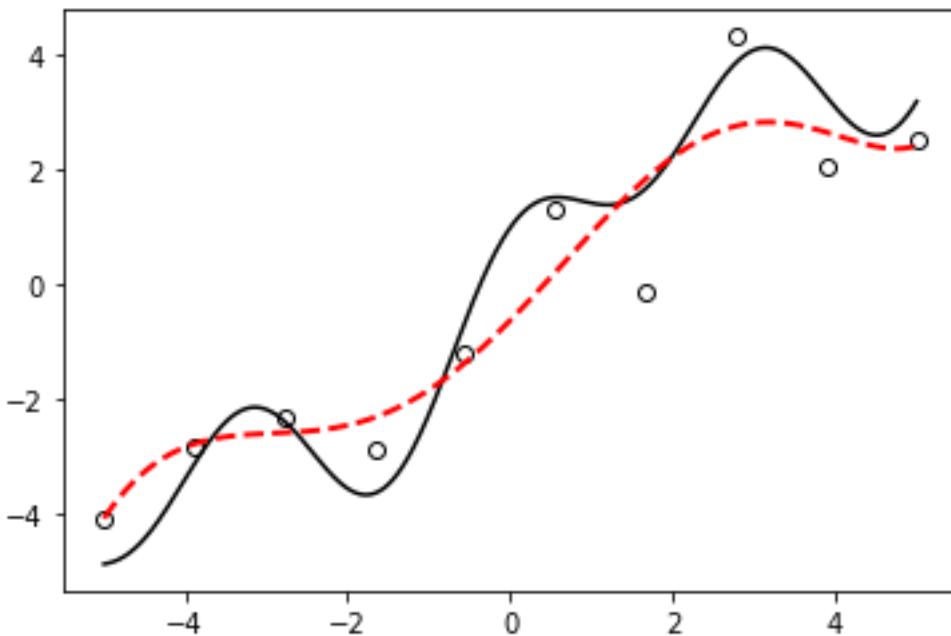
בעיה זו ניתן לפתור, למשל, **בעזרה רגרסיה פולינומיאלית**: נחפש פונקציה מהצורה

$$\hat{y}(x) = a_0 + a_1 x + \dots + a_n x^n$$

אשר עוברת "הכי קרובה" אל הנקודות הלבנות. לא נוכיח זאת כאן, אך תחת הנחות קלות קיימת בחירה יחידה של פרמטרים  $a_0, a_1, \dots, a_n$  אשר מזערת את ממוצע ריבועי המרחקים בין התוצאות  $y_k$  לבין נקודת הדגימה מסט האימון  $(x_k, y_k)$ . בקיצור, הווקטור  $(x_k, \hat{y}(x_k))$

$$(a'_0, \dots, a'_n) = \underset{(a_0, \dots, a_n) \in \mathbb{R}^n}{\arg \min} \frac{1}{N} \sum_{k=0}^N (\hat{y}(x_k) - y_k)^2$$

קיימים, והוא ייחיד. זאת ועוד, ניתן למצוא אותו אונלייטית באמצעות פתרון מערכת המשוואות לינארית. הפולינום ממעלה 6 המתkeletal מהפעלת תחילה זה על אוסף הנקודות שדגמנו לעיל מאויר להלן.



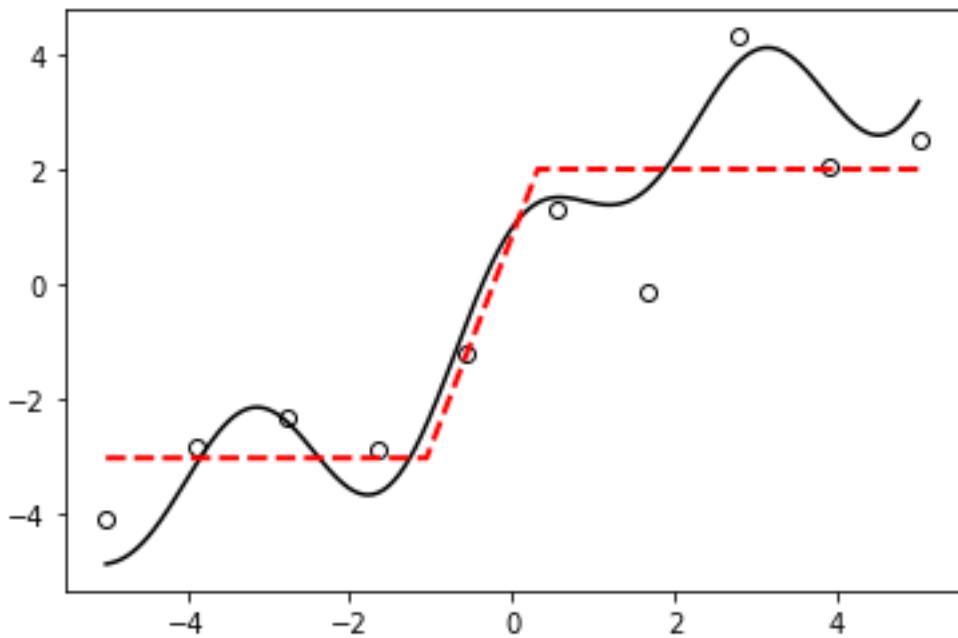
כעת, ברצונו לשאוב השראה מאלגוריתם זה ולאמן רשת נוירונים לפתור את אותה הבעיה. אם כן, פلت הרשת צריך להיות כפלט הפונקציה  $\hat{y}(x)$ , הערך החזוי עבור  $x$  נתון. כמו כן יש צורך בפונקציית מבחן חדשה, אך זו למעשה כבר כתובה לעיל – זהו ממוצע ריבועי השגיאות (Mean Squared Error),

$$MSE(\text{Model}) = \frac{1}{\#Data} \sum_{(x_k, y_k) \in Data} (\text{Model}(x_k) - y_k)^2$$

נשים לב שביטוי זה גזיר לפט הרשת  $\text{Model}(x_k)$ , ולכן ניתן להפעיל את האלגוריתם SGD למציאת פרמטרים של הרשת אשר ממזערים מחיר זה. נגידור אףוא רשות שהשכבה الأخيرة של היאנוירון לינארי ייחיד, ונתאים את הרשת לנתחים בקטע הקוד שלහלן.

```
model=nn.Sequential(nn.Linear(1,2),nn.ReLU(),
                    nn.Linear(2,2),nn.ReLU(),
                    nn.Linear(2,1))
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
loss = nn.MSELoss()
```

כל שנותר לעשות כעת הוא לאמן את הרשת בollowat אימון סטנדרטית. התוצאה מאוירת להלן.



ニיכר שאפשר לשפר את התוצאה, ואכן נעשה זאת בהמשך יחידת לימוד זו.

## שאלות לתרגול

- מנו את מספר הפרמטרים בראש ש hogradaה לעיל.
- הסבירו מדוע למספר פרמטרים הגדול יותר, ניכר שהרשות חוזה פונקציה פשוטה יותר מאשר גרסיה פולינומיאלית של פולינום ממעלה 6.

לעתים בעיית הרגרסיה מנוסחת תוך כדי שימוש בפונקציית המחיר **סכום הריבועים**:

$$S(\text{Model}) = \sum_{(x_k, y_k) \in Data} (\text{Model}(x_k) - y_k)^2$$

במקום **ממוצע הריבועים**:

$$MSE(\text{Model}) = \frac{1}{\#Data} \sum_{(x_k, y_k) \in Data} (\text{Model}(x_k) - y_k)^2.$$

ניסוח הבעיה בשתי הדרכים שקול, שהר ששתי הפונקציות נבדלות זו מזו רק בקבוע, ולכן מתקבלות מינימום באותו ערכיהם. עם זאת, באופן פרקטי קיים הבדל. בהנחה שברשותנו הרבה נקודות דוגמה, הסבירו את הביעיות של שימוש בסכום הריבועים לאימון הרשות. רמז: חשבו את הנזירות של שתי פונקציות המחיר לפי פلت המודל.

3. הניחו שבין שלושה משתנים מתקיים הקשר המתמטי:

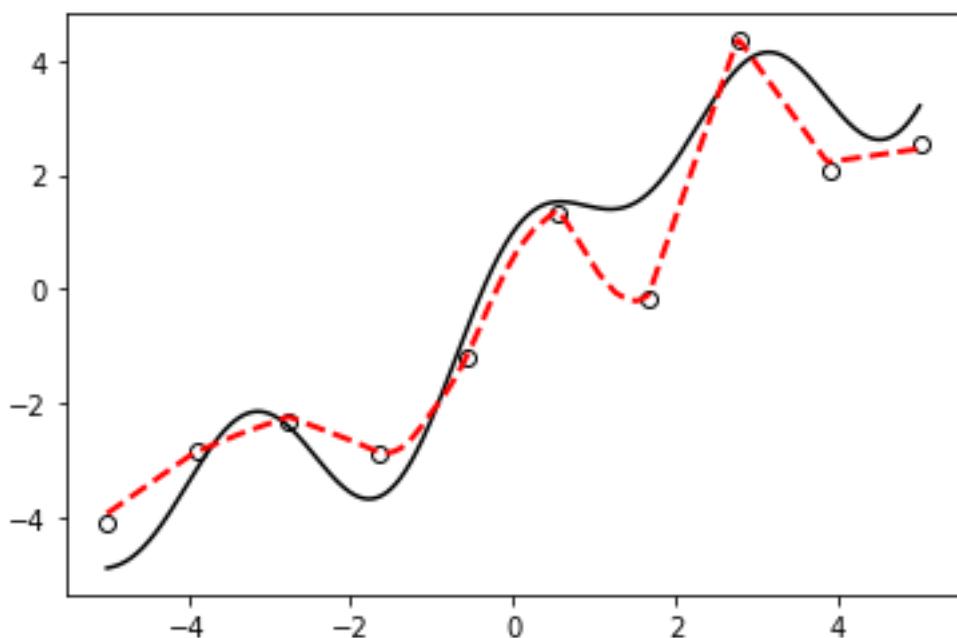
$$z = x + y + xy + \sin\left(\frac{x^2}{y}\right)$$

דגמו סט אימון "רועל" ובצעו גרסיה בעזרת רשת נוירונים למשתנה  $z$ .

## התאמת יתר

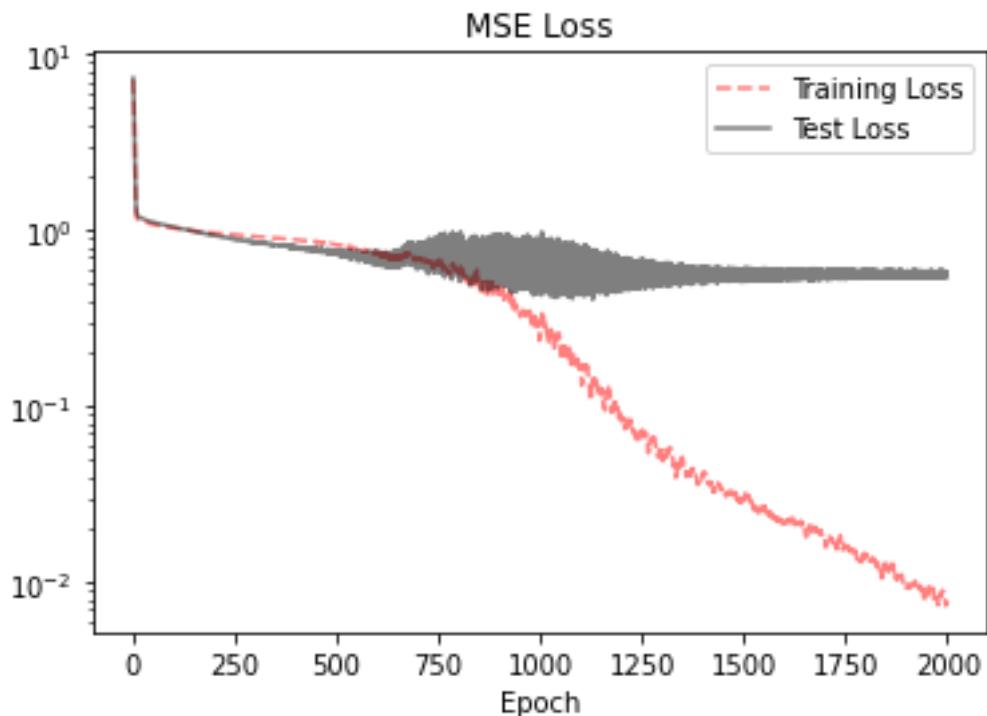
את הפרק הקודם סיימנו באימון רשת נוירוניים בסיסית לפתרון בעיית גראסיה פשוטה, וראינו שה透וצאה שהתקבלה שוחרה בקווים כליליים את הכלל האמייתי שמננו נוגם סט האימון, אך ודאי שלא את כל המרכיבות אשר באה לידי ביטוי בו. זהו בכך כלל איןו המצב כאשר אנו מאמנים רשותות נוירוניים מודרניות. בהיוטן רשותות עמוקות בעלות מספר פרמטרים עצום, הן יכולות לבטא מרכיבות כמעט בלתי מוגבלות. לצורך הדוגמה, נאמן את הרשות המוגדרת בקטע הקוד שלහן, שהיא רשות מרכיבת יותר, אך עדין קטנה לעומת רשותות מודרניות, לפתרון אותה הבעיה. נקבל כתוצאה פונקציה המתאימה לסט האימון באופן מושלם, כפי שניתן לראות באIOR למטה.

```
model=nn.Sequential(nn.Linear(1,100),
                    nn.ReLU(),
                    nn.Linear(100,100),
                    nn.ReLU(),
                    nn.Linear(100,100),
                    nn.ReLU(),
                    nn.Linear(100,1))
```



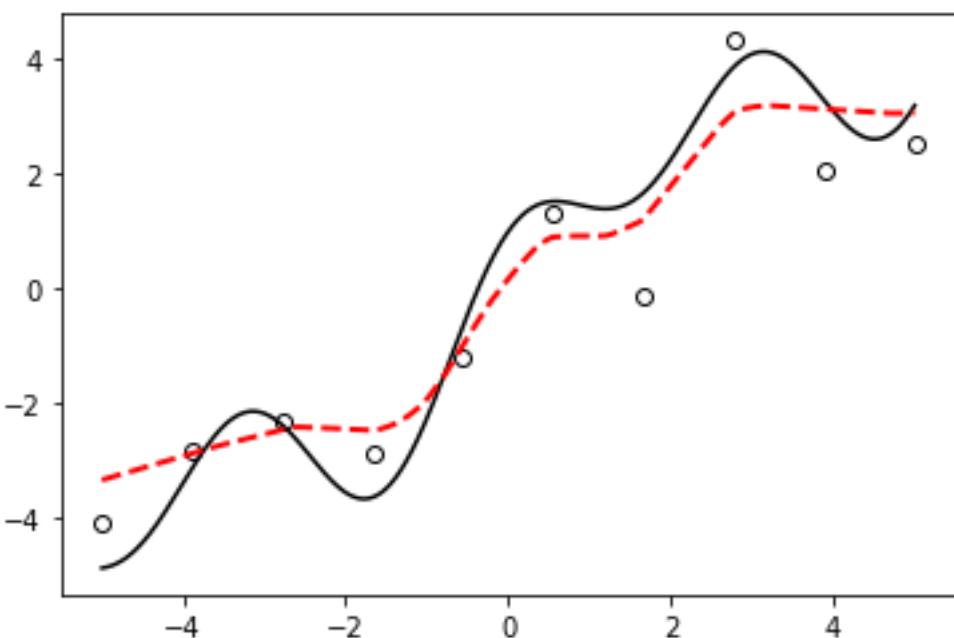
透וצאה זו לא הייתה בעייתית לו היה סט האימון מייצג באופן מושלם את הכלל האמייתי (לו היו הנקודות נמצאות בדיק על הקו השחור), אך זהו איןו המצב, שכן סט האימון נוגם עם רעש. כמו בדוגמה, כך גם במקרה, סט האימון שאנו משתמשים בו לאימון כל מודל מכיל מידע רב על הכלל שאנו רוצים ללמד את הרשות, שהוא כלל רלוונטי גם לדוגמאות מחוץ לסט האימון. ואולם, סט האימון מכיל גם מידע לא רלוונטי הנובע מתחילה איסוף הנתונים הרועש ומוגדל הסופי של הסט. אלגוריתם למידה מצליח ילמד את המידע הרלוונטי לכל הנתונים ויתעלם מהמידע הספציפי לסט האימון – בឋיאן הנקרא **הכללה** (generalization).

מטרתנו בפרק זה היא אפוא להוסיף לתהיליך של אימון הרשות כלים אשר יפחיתו את שגיאות ההכללה (generalization error), שאוთה נאמודה בעורמת סט הבדיקה של הנתונים – אוסף נתונים נוספים אשר לא השתמשנו בו בעת האימון. רשות מוצלחת אשר ביצעה הכללה ולא "שינהה" את נתוני האימון ופגינו ביצועים טובים גם על סט נתונים זה. המצב החפוץ, שבו הרשות משנה את סט האימון, בא לידי ביטוי בכך שבמהלך תהליכי האימון פונקציית המחיר של סט האימון ממשיכה לרדת, אך כאשר אנו מחשבים את המחיר של סט הבדיקה – השגיאה אינה משתנה, ואף עולה לאורך דורות האימון. מצב זה, הנקרא התאמת יתר (overfitting), ניכר בבירור בדוגמה הנ"ל: ראו את גרפ' שגיאת האימון ושגיאת הבדיקה המאויר להלן.



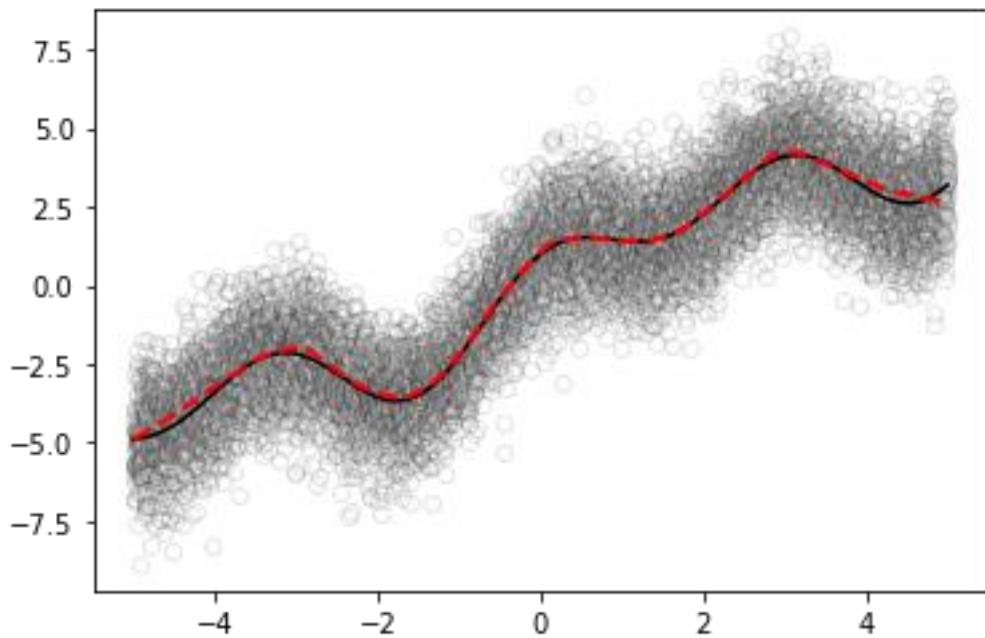
מайיר זה ניכר שהחל מהepoch ה-500 לערך, הרשות התחליה לשנן את סט הנתונים – היא המשיכה להתקrab אל הנקודות הלבנות, שכן שגיאת האימון המשיכה לרדת, בעוד השגיאה על סט הבדיקה לא ירדה. הקربה אל הנקודות הלבנות באה על חשבון הקربה למודל התיאורטי (הקו השחור) ומכך אנו רוצים להימנע.

השינוי של תהליכי אימון הרשות שמטרתו להקטין את שגיאות ההכללה נקרא רגולרייזציה (regularization), ובהמשמעות היחידה נלמד שיטות אחדות לבצע זאת. הראשונה שניתן להפעיל היא הבסיסית ביותר: אם החל מ-epoch מסוים שגיאת הבדיקה מתחילה לעלות, נאמנו את הרשות עד לepoch זה בלבד. באופן לא מפתיע שם השיטה הוא עצירה מוקדמת (early stopping). תוצאה אימון הרשות הנ"ל לאורך epochs 500 בלבד מאוירת להלן, וממנה ניכר בבירור שהרשות עוד לא הספיקה לשנן את הנתונים.



כאשר משתמשים בשיטה זו, יש לתת את הדעת לכך שטח הבדיקה שימוש בבחירה הדור האידיאלי לעצירת תהליכי האימון. על כן, שגיאת הבדיקה הנמדדת לפי סט זה אינה בהכרח מייצגת נאמנה את שגיאת הכללה – הרוי דור העצירה נבחר לבדוק כדי למנוע שגיאה זו. לפיכך, אם ברצונו לammo את שגיאת הכללה של הרשות יש להשתמש בסט נתונים נוסף, שלא השתמשו בו כלל. רשמית, השם "סט הבדיקה" שמור לאוסף נתונים נוסף זה, ואוסף הנתונים שהשתמשו בו עד עתה לבדיקת ביצוע המודל הוא "סט הולידייזיה" (validation set). עם זאת, בהמשך הלימוד, כאשר נשתמש רק בסט אימון וסט נתונים נוסף לבדיקה, נழיק לקרוא לו סט הבדיקה כמקובל בתחום הלמידה העמוקה, בזדענו שאין זה השם המדוייק.

השיטה הבסיסית השנייה לרגולרייזציה היא הטבעית ביותר – לאסוסף יותר נתונים מתוך ציפייה שלרעש הבא לידי ביטוי בכל דוגמה תהיה משמעות פחותה כאשר סט האימון גדול. לדוגמה, אם נאמן את רשות הרגרסיה הניל על סט אימון גדול הרבה יותר, אשר נדגם באותו תהליך בדיקת סט האימון הקודם, תוצאה הרשות תימצא בדיקת סט האימון המקורי, כפי שניכר באIOR המצורף.



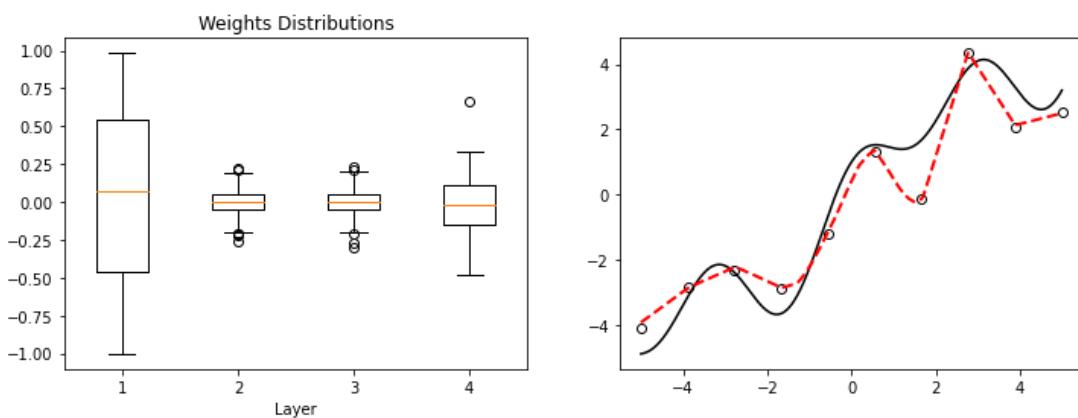
ביצועיהן העדיפים של רשתות נוירונים על פני אלגוריתמי למידה אחרים נובעים בmäßigה רבה מרעיון פשוט זה – בהיותו כה גמישות, דרוש סט אימון גדול מאוד כדי לאמן.

## שאלות לתרגול

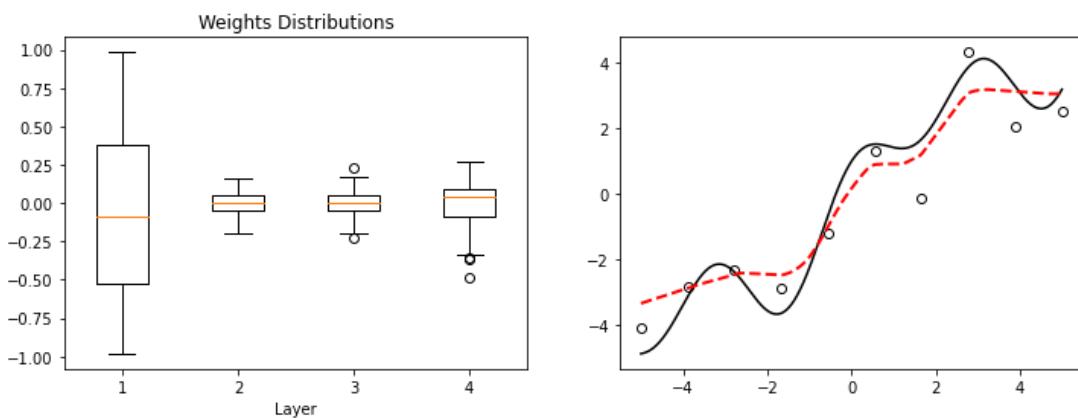
1. בשאלת זו תיוכחו בקיים תופעת התאמת היותר כפי שהיא באה לידי ביטוי ברשות לסייעת תמונות פריטי הלבוש בסט הנתונים Fashion-MNIST.  
א. חלקו את סט נתוני הבדיקה לשני חלקים : ולידציה (80% מסט הבדיקה המקורי) ובדיקה (20% מהנותרים).  
ב. אמנו רשת عمוקה לפתרון בעיית הסיווג עד אשר מתקבלת התאמת יתר לסט הולידייצה החדש.  
ג. מצאו את ה-epoch האידיאלי לעצרת תהליכי האימון, ואמנו את הרשת עד epoch זה.  
ד. מددזו את ביצועי הרשת לאורך הדורות על סט הבדיקה.  
ה. הסבירו את הפרע בין שגיאת הולידייצה לבין שגיאת הבדיקה.  
2. חזרו על שאלה 1 כאשר גודלו של סט האימון הוא 10% מגודלו של סט האימון המקורי. מה תוכלו להגיד על ה-epoch האידיאלי לעצרה מוקדמת במקרה זה, לעומת התוצאה הקודמת?  
3. בשאלת זו נבדוק את ההשפעה של אימון הרשת על סט נתונים אחד, והפעלתו על סט נתונים השונה בmphowto.  
א. חזרו על שאלה 1 כאשר בסט האימון רק חצי מהמחלקות (מחלקות 0–4), בסט הולידייצה רק המחלקות 0–6 ובסט הבדיקה המחלקות 5–9. **הערה:** את הדוגמאות הרלוונטיות תוכלו למצוא `בעזרת הפונקציה torch.where`.  
ב. האם איסוף נתונים נוספים לסט האימון יעזור להקטנת שגיאת ההכללה? הסבירו תשובהכם.

## דעת המשקלים

הפתרון הקלסטי לביעית התאמת היותר באלגוריתמים של למידת מכונה הוא העדפת מודלים "פשוטים" על פני " מורכבים": ככלו אשר אין גמישים דיים כדי לשנן את סט הלמידה. בהקשר של רשתות נוירוניים עיקרונו זה בא לידי ביטוי באמצעות העדפת רשת המפעילה פחות נוירוניים ובעצמה נhocה יותר מאשר רשת אחרת. על סמך הדוגמה מהפרק הקודם משורטט להלן גרף של התפלגיות המשקלים בכל שכבה בראשת אשר שינה את סט האימון,



לעומת התפלגיות המשקלים של הרשת אשר עיצרנו את אימוננו מוקדם.



באיורים אלו סיכמנו את התפלגיות המשקלים בצורת boxplot אשר תכונתיו הן :

- 75% מהדגימות נמצאות בתחום הקופסה המרכזית.

• החציון מסומן בקו באמצע הקופסה.

• לקופסה "שפמים" (whiskers) אשר מגיעים עד לדגימה הגדולה או הקטנה ביותר באוסף הנתונים,

אשר מרוחקת מהקופסה לכל היותר 1.5 פעמיים גודל הקופסה.

- דגימות רחוקות יותר מסומנות בנקודות לבנות.

ניתן לצייר boxplot ולשלוט בפרמטרים השונים בעזרת הפונקציה `boxplot` של הספרייה `matplotlib`.

מהאיורים ניכר בבירור הבדל בהתפלגות המשקלים בשתי הרשותות: ברשות בעלת התאמת היותר והתפלגות רחבה יותר: שפמי ה-*boxplot* רחבים במעט וכן יש מספר גדול יותר של ערכים חריגים הנמצאים מעבר לשפים.

בפרק זה נלמד שיטות רגולרייזציה נוספות אשר כופות על הרשות לכבות או להנמק את העוצמה של משקלים שאינם תורמים במידה רבה להקטנת השגיאה של הרשות: נוסיף לפונקציית המחיר של הרשות "קנס" על מרכיבות. אם בעבר פונקציית המחיר אמדה את ההתאמה של הרשות לאוסף הנתונים בלבד, למשל באמצעות חישוב ממוצע ריבועים במקרה של בעיית רgression,

$$C(\text{Model}) = \text{MSE}(\text{Model})$$

כעת לפונקציית המחיר אשר אלגוריתם האופטימיזציה ימזרע יתרוסף וርיב רגולרייזציה, המתגמל מודלים שבهم ערכי הפרמטרים קטנים יותר. לפיכך, מודלים אשר מתאימים במיוחד לסט האימון אך בעלי ערכי פרמטרים גדולים יידחו בתהליך האופטימיזציה. כך למעשה אנו מצמצמים את תחום החיפוש למודלים אשר ערכי הפרמטרים שלהם קרוביים בראשית בלבד. נוסחת פונקציית המחיר תהיה מהצורה

$$C(\text{Model}) = \text{MSE}(\text{Model}) + \lambda \cdot \text{Size}(\text{Weights})$$

כאשר הפרמטר  $\lambda$  (שהוא תמיד אי-שלילי) מażן בין שתי המטרות השונות של פונקציית המחיר החדשה – ההתאמה לנ נתונים והעדפת מודל פשוט. עבור  $\lambda = 0$  לא תבוצע רגולרייזציה כלל, ועבור  $\lambda > 0$  גדול איפוס הפרמטרים ישתלט על תהליך האופטימיזציה, והרשות לא תלמד כלל.

## רגולרייזציה $L_2$

אחד האפשרויות השכיחות למדד הגודל של הפרמטרים הוא נורמת  $L_2$  של המשקלים (למעשה אנו מעדיפים את מחצית ריבוע הנורמה, לשם פשטות החישובים):

$$\cdot \frac{1}{2} L_2(\text{Weights})^2 = \frac{1}{2} \sum_{w \in \text{Weights}} w^2$$

בשיטת רגולרייזציה זו יתווסף בכל צעד באלגוריתם האופטימיזציה וርיב הרגולרייזציה לחישוב הגרדיאנט:

$$\cdot \frac{\partial C}{\partial w} = \frac{\partial \text{MSE}}{\partial w} + \frac{\partial}{\partial w} \left( \frac{\lambda}{2} L_2 \right) = \frac{\partial \text{MSE}}{\partial w} + \lambda w$$

על כן הפרמטר  $w$  יעדכן באמצעות SGD כדלקמן:

$$w_{t+1} = w_t - \alpha \left( \frac{\partial \text{MSE}}{\partial w} + \lambda w_t \right) = (1 - \alpha \lambda) w_t - \alpha \frac{\partial \text{MSE}}{\partial w}$$

מנוסחה זו ניתן להבין את השם שניית לשיטה בקהילת הלמידה העמוקה – דעיכת המשקלים (weight decay): בכל איטרציה המשקל  $w$  מאבד שיעור מסוים מערכו, ודועך אל האפס. שימוש לב שילג משפייע על הגודל של העדכון של אלגוריתם SGD: עבור ערכי  $\lambda > 0$  גדולים מדי יתכן שהאלגוריתם כבר לא יתכנס עקב קבועות גבולות מדי בין האיטרציות, תופעה הדומה לבחירת צעד  $\alpha$  גדול מדי.

נמש שיטה זו בקוד. המתודה `named_parameters()` של המודל מחזירה איטרטור על כל הפרמטרים הקיימים בראש ייחד עם שמותיהם, ובמעבר על כולם נחלץ לשימה אחת רק את **המשקלים**. אלו הפרמטרים אשר עליהם נפעיל רגולרייזציה, שכן לרוב אין לנו מעוניינים לבצע רגולרייזציה על ערכי `bias` של הנוירונים – הסוג השני של פרמטרים בראש הרגרסיה שאנו עובדים איתה כעת.

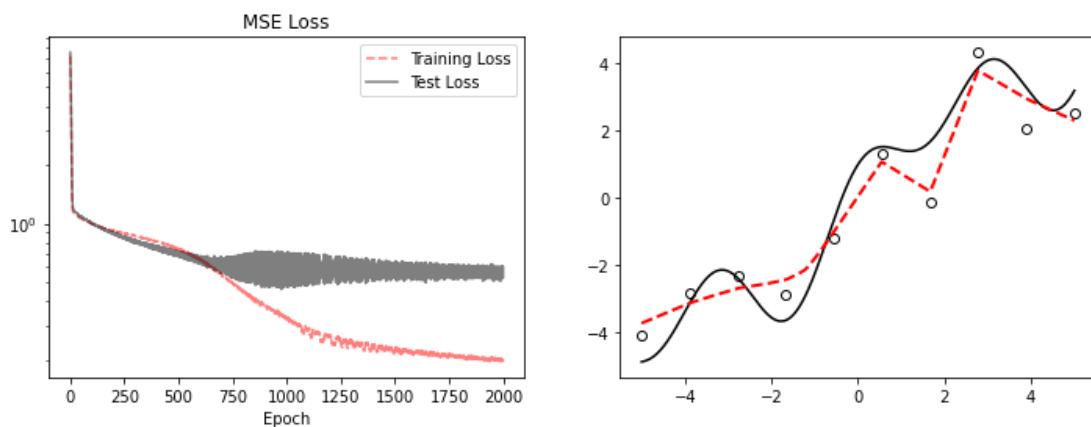
```
weights = [param for (name,param) in model.named_parameters()
           if "weight" in name]
```

אחרי כן, נוסיף לולאת האימון את חישוב רכיב הרגולרייזציה, ולבסוף נחשב את הגראדיאנט של פונקציית המחיר החדשה, המורכבת משני החלקים.

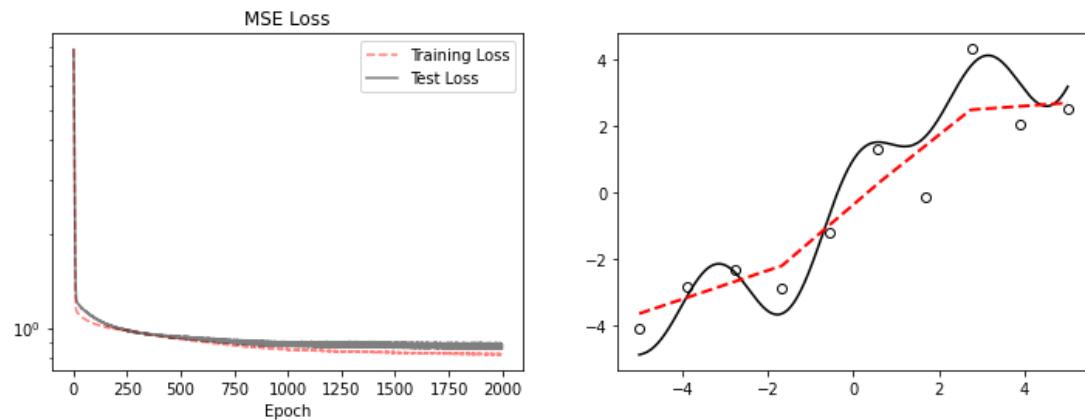
```
reg_loss = 0
for param in weights:
    reg_loss += (param**2).sum()
total_loss = MSE_loss + 1/2*lambd*reg_loss
total_loss.backward()
```

המשך תהליך אימון הרשות נשאר כשהיה.

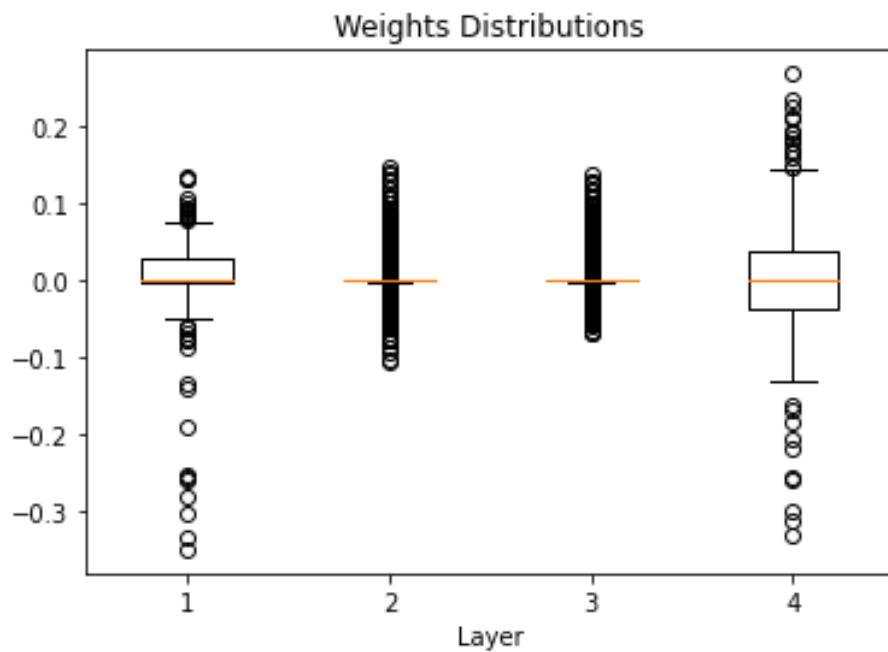
תוצאות תהליכי האימון עבור ערך  $\lambda$  קטן מאוד להלן, ונניתן לראות שעדין מתבצעת התאמת יתר לאורך דורות האימון, אך השפעתה פחותה מבעבר.



עבור ערך  $\lambda$  גדול יותר, תתקבל התוצאה שלහלן, וממנה ניכר שאין התאמת יתר ניכרת אך המודל המתתקבל פשוט יותר על המידה.



במבחן עמוק יותר על תוצאות הריצת האחורונה, ניכר שרוב המשקלים בשכבות הפנימיות בראשת כמעט התאפסו, שכן תרומות להורדת מחיר ההסתאמות של הרשות לנוטנים לא הייתה ניכרת דיה בהשוואה לגורם הדעיכה של הרגולרייזציה. ראו זאת באירוע ההתפלגות שלහן.



## רגולרייזציה $L_1$

שיטה שכיחה נוספת לרגולרייזציה היא שימוש בנורמה  $L_1$  של המשקלים, כך שפונקציית המחיר החדשנית שיש למזרע היא

$$C(\text{Model}) = \text{MSE}(\text{Model}) + \lambda \sum_{w \in \text{Weights}} |w|$$

בדומה לשיטה הקיימת, נתבונן בהשפעת השיטה על הגredient של פונקציית המחיר :

$$\frac{\partial C}{\partial w} = \frac{\partial \text{MSE}}{\partial w} + \lambda \text{sgn}(w)$$

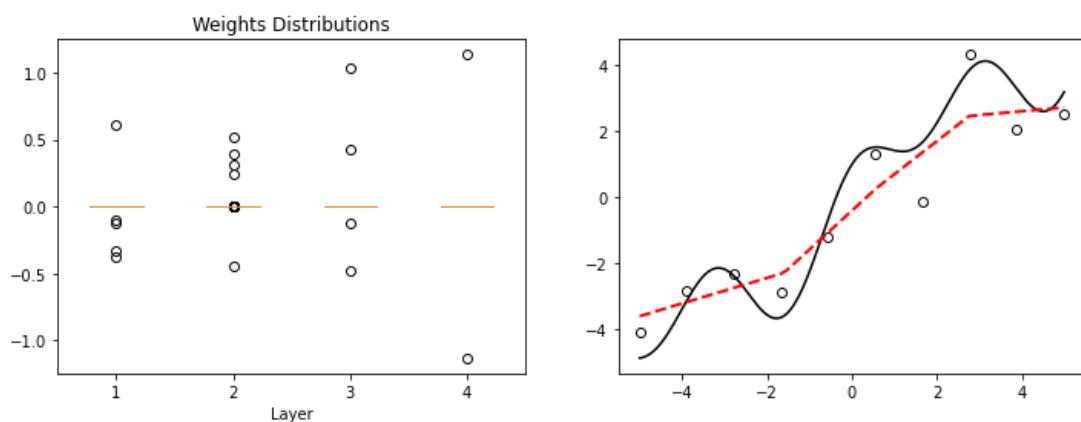


כאשר  $(w)$  מוגדר היא פונקציית הסימן המוחזירה תמיד את אחד הערכים  $1 \pm$ . שימוש לב ששוב התעלמנו מהבעיה בחישוב הנגזרת של  $|w|$  בנקודה 0, בדומה לאקטיבציה  $\text{ReLU}(x)$ . צעד העדכון ב-SGD הוא:

$$w_{t+1} = w_t - \alpha \left( \frac{\partial \text{MSE}}{\partial w} + \lambda \text{sgn}(w) \right) = w_t - \alpha \frac{\partial \text{MSE}}{\partial w} - \alpha \lambda \text{sgn}(w)$$

למרות הדמיון הרב בין שתי השיטות, יש לשים לב לכך שברגולרייזציה  $L_1$  דעיכת הפרמטר  $w$  לכיוון האפס אינה פרופורציונית לגודלו: בכל איטרציה מתבצע צעד קבוע בגודל  $\lambda$ . כמו במקרה הקודם, יש לשים לב לבחירה שcolaה של גודל הצעד  $\alpha$  ושל פרמטר הרגולרייזציה  $\lambda$  כך שמכפלתם לא תהיה גדולה מדי, אחרת הקפיצות הקבועות בגודל זה לכיוון האפס יובילו לתנועה מוחזרת ואלגוריתם האופטימייזציה לא יתכנס.

גודל הצעד הקבוע משפיע דרמטית על פרמטרים קטנים בגודלם, אך השפעתו זניחה על פרמטרים גדולים. התוצאה המתבקשת היא שהרשת תרכז את רוב המשקל במספר מועט של משקלים, והשאר יתאפסו לחולטיין. ראו דוגמה לכך בתוצאת אימון רשת הרגסינית יחד עם שיטת רגולרייזציה זו באירוע המصور.



בבואהו למש שיטה זו בקוד, כל שיש לעשות הוא לחשב את מחיר הרגולרייזציה בדרך אחרת:

```
reg_loss = 0
for param in weights:
    reg_loss += param.abs().sum()
total_loss = MSE loss + lambda*reg loss
```

את ההבדל בין התוצאות המתתקבלות משתי גישות אלו לרגולרייזציה ניתן להבין גם דרך דוגמה פשוטה מאוד, שבה הפרמטרים של המודל הם  $w_0, \dots, w_n$  ופונקציית המחיר היא:

$$\text{Loss}(w_0, \dots, w_n) = \frac{1}{2} \sum_{k=0}^n H_k (w_k - w_k^*)^2$$

כאשר  $H_k > 0$  לכל  $k$ . אומנם מקרה זה רחוק מאוד מפונקציות המחיר המורכבות של רשתות נוירוניים עמוקות, אך הוא אינו מנוטק מהמציאות: פונקציה כזו, או דומה לה, מופיעה בעקבות רגרסיה ליניארית.

נקודות המינימום (היחידה) של מחיר זה היא  $w_0^*, w_1^*, \dots, w_n^*$  – ובה ערך הפונקציה הוא אפס – אלו פרמטרי המודל האופטימליים. אלו מוצפים כי הרגולרייזציה תסייע את הערכות האופטימליים אל הראשית, אך החבדל בין שתי הגישות הוא בכך שבה הסטה זו מתרחשת כאשר הפרמטר  $\lambda$  גדול. ראשית נتبונן במקורה שבו לפונקציית המחיר נוספת גורם של רגולרייזציה  $L_2$ :

$$C_2(w_0, \dots, w_n) = Loss + \frac{\lambda}{2} \sum_{k=0}^n |w_k|^2$$

ניתן להוכיח (אך לא נעשה זאת כאן) כי נקודת המינימום החדשה, שנסמנה  $(\hat{w}_0, \dots, \hat{w}_n)$ , מתקבלת מהמקורית לפי הנוסחה:

$$\hat{w}_k = \frac{H_k}{H_k + \lambda} w_k^*$$

מכאן נסיק שככל משקל אכן כווץ לכיוון האפס, **באופן כפלי**, אך ככל בעלי מקדם  $H_k$  גדול לעומת גולן המשקלים בעלי השפעה רבה על פונקציית המחיר (מעט שלא כוווץ).

לעומת זאת, עבור רגולרייזציה  $L_1$  פונקציית המחיר החדשה היא

$$C_1(w_0, \dots, w_n) = Loss + \lambda \sum_{k=0}^n |w_k|$$

נקודות המינימום החדשה מתקבלת לפי הנוסחה

$$\hat{w}_k = \text{sgn}(w_k^*) \cdot \max\left(|w_k^*| - \frac{\lambda}{H_k}, 0\right)$$

נוסחה זו יש להבין כך: אם מרחקו של הפרמטר  $w_k^*$  מהאפס היה גדול מ-  $\frac{\lambda}{H_k}$ , הוא הוסט לכיוון האפס באמצעות החסרה או תוספת של גורם זה. אם ערכו המקורי היה קטן מדי – הוא פשוט אפס. כאן ניתן לראות בבירור כיצד גישת רגולרייזציה זו מביאה לאי-פוס פרמטרים, וכי-זד גודל הפרמטר המקורי אינו משפיע על עוצמתו השינוי בערכו.

## שאלות לתרגול

- הפעילו רגולרייזציה על כל פרמטרי רשות הרגסיה שבה עסקנו בפרק הקודם והנוכחי, כולל ערכי  $bias$ , וアイירו את התוצאות המתקבלות. האם דרך דוגמה זו תוכלו להסביר למה לא נהוג להפעיל עליהם רגולרייזציה?
- אמנו את רשות הסיווג של תמונות פרטי הלבוש בסט הנתונים Fashion-MNIST עם רגולרייזציה  $L_1$ . מצאו את ערך הפרמטר  $\lambda$  האידיאלי המאזן בין פשטות המודל לבין התאמת יתר לסט האימון. רמז: חפשו את הפרמטר  $\lambda$  שעבورو מתקבלת שגיאת ההכללה הנמוכה ביותר.
- הסבירו لماذا השאלה הקודמת יש צורך בשימוש בסט נתונים ולידציה וסט נתונים בדיקה להערכת ביצועי הרשות באופן אמין.

## אימון רשת בעזרת מאיצ' גרפי

חלק ניכר מഫולות החישוביות שאנו מבצעים בעת אימון רשת נוירונים ניתנות לביצוע ברזמנית. הראשונה שבהן, אך לא היחידה, היא כפל מטריצות, הבא לידי ביטוי בין השאר בשכבות לנאריות. נניח שוקטור הקלט לשכבה לנארית נתונה הוא  $(x_1, \dots, x_d) = X$ , אז בעת התפשטות פנים ברשת מתבצע החישוב שלහן (עבור דוגמה יחידה),

$$\begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix} = \begin{pmatrix} w_{11} & \cdots & w_{1d} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{md} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} = \begin{pmatrix} \sum_k w_{1k} x_k \\ \vdots \\ \sum_k w_{mk} x_k \end{pmatrix}$$

עבור minibatch (בגודל  $N$ ),

$$\begin{pmatrix} z_1^{[1]} & \cdots & z_1^{[N]} \\ \vdots & \ddots & \vdots \\ z_m^{[1]} & \cdots & z_m^{[N]} \end{pmatrix} = \begin{pmatrix} w_{11} & \cdots & w_{1d} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{md} \end{pmatrix} \begin{pmatrix} x_1^{[1]} & \cdots & x_1^{[N]} \\ \vdots & \ddots & \vdots \\ x_d^{[1]} & \cdots & x_d^{[N]} \end{pmatrix}$$

ראו כי עבור כל איבר במטריצה הפלט יש לחשב את הביטוי

$$z_a^{[b]} = \sum_k w_{ak} x_k^{[b]}$$

ולכן למעשה, כל ביטוי מהצורה  $w_{ak} x_k^{[b]}$  ניתן לחישוב ברזמנית בלבד בלבנה נפרדת ולבסוף אפשר לסקום את התוצאות לקבלת הפלט. נציין שקיימים אלגוריתמים מתקדמים לכפל המטריצות, וחבילות התוכנה שאנו משתמשים בהן מושמות אותם, אך גם הם ניתנים למימוש ברזמני, בדומה לחישוב הנאיבי.

אם כן, יש לנו צורך ברכיב חומרה בעל מספר רב של ליבות המסוגלות לבצע פעולות פשוטות (כגון כפל שני סקלרים). מאיצ' גרפי הוא בדיקת רכיב שכזה, ולכן הוא מועמד טבעי להאצת החישובים של מודלי למידה عمוקה. וההיסטוריה מלמדת לכך היה: רשתות הנוירונים הראשונות אשר ניצחו את המתחרים בתחרויות למדידת מכונה, כגון ImageNet, ובישרו את עלייתה של הלמידה העמוקה לבכורה בעשור השני לשנות האלפיים, אומנו על גבי מאיצים גרפיים. מאז פותחו רכיבים ייחודיים להאצת החישובים הנפוצים במימוש רשתות נוירונים, וכן ממשקי הפשטה לעובדה עם רכיבים אלו, כך שכiams תהליך האימון (ולעתים גם החיזוי) מבוצע כמעט חלוטין ברזמנית.

השימוש במאיצ' גרפי בעזרת PyTorch הוא פשוט. ראשית יש לבדוק שבסביבת העבודה מותקן וርיב CUDA, שהוא המשק שבו ממומשת יכולת זו ב-[PyTorch](#). ניתן לעשות זאת בעזרת פקודה `nvcc -smi shell` אשר תחזיר פירוט של המאיצים הזמינים, או שירות דרך PyTorch

```
torch.cuda.is_available()
```

פלט:

```
True
```

שירותויינו אחדים מספקים סביבות עבודה עם מייצים גרפיים, בinclusive Colab של גוגל: היכנסו לתפריט Edit, בחרו באפשרות Notebook settings וסמן GPU את מאייז החומרה המבוקש.

כעת, כשייש יותר מרכיב חומרה יחיד זמין, עלינו להיות מודעים למאפיין device הקיים בכל טנזור. מאפיין זה מצין את הרכיב שהטנזור ממוקם עליו. מקום ברירת המחדל הוא המעבד הראשי (CPU):

```
A = torch.empty(1)
print(A.device)
```

פלט:

```
cpu
```

ביכולתנו להעביר טנזורים מרכיב אחד לאחר בעזרת המתודה () .  
to, כלהלן.

```
device = torch.device('cuda:0')
A      = A.to(device)
print(A.device)
```

פלט:

```
cuda:0
```

שים לב שם המאייז הגרפי (הראשון או היחיד בסביבת העבודה) הוא 0 : cuda. כמו כן, דעו כי העברת המשתנה מהמעבד הראשי אל המאייז היא פעולה הכרוכה בהעתיקת נתוני המשטנה מזיכרונו המעבד אל זיכרונו המאייז, וכך התוצאה של המתודה () .  
to היא תמיד עותק של המשטנה המקורי.

ברירת מחדל, התוצאה של פעולה חיבור המבוצע על שני טנזורים הנמצאים באותו רכיב נשמרת גם היא ברכיב זה. למשל,

```
A = torch.zeros(1, device=device)
B = torch.ones(1, device=device)
C = A+B
print(C.device)
```

פלט:

```
cuda:0
```

ראו גם שניתן להגדיר את ה-device של הטנזור כבר בשלב האתחול.

מכיוון שהעתיקת נתונים מרכיב חומרה אחד לאחר היא פעולה איטית מאוד, היא אינה מתבצעת אוטומטית אלא רק לאחר הוראה מפורשת מהמשתמש. לכן, אם ננסה לבצע פעולה חיבור בין שני טנזורים הנמצאים על רכיבים שונים תתקבל שגיאה. ראו למשל:



```
A = torch.zeros(1, device=device)
B = torch.ones(1, device='cpu')
C = A+B
```

**פלט:**

`RuntimeError: Expected all tensors to be on the same device,  
but found at least two devices, cuda:0 and cpu!`

ניתן להבהיר אל המאיצ' הגרפי לא רק טנзорים, אלא גם מודלים שלמים (כגון אלו אשר יצרנו בעזרת הפקודה `nn.Sequential()`), שכן גם עבורם קיימת המתוודה (`.to()`).

הprt האחxon שצרכ' להביא בחשבון הוא שספריות פיתון אחרות, כגון NumPy או matplotlib, אין עובדות עם המאיצ' באופן מובנה. לכן קודם לשימוש בהן, למשל לצורך ציור גרף, علينا להעתיק את הקלט שלחן חורה ל-CPU. אפשר לבצע זאת בעזרת המתוודה (`.cpu().tensor()`). אם הטנזור כבר נמצא על המעבד הראשי, לא מתבצעת העתקה נוספת, ולכן ניתן להשתמש במתוודה זו בחופשיות.

לסיום, כדי לאמן מודל על גבי המאיצ' הגרפי כל שיש לעשות הוא להבהיר אליו את המודל לאחר הגדרתו, וכן להבהיר אליו כל `minibatch` של נתונים, לאחר טעינתו. שאר לולאות האימון אינה משתנה. שימו לב שכוחו של המאיצ' הגרפי בא לידי ביטוי כאשר פעולות החישובן הן צוואר הקבוק של תהליך האימון, ככלمر כאשר רשת הנוירונים גדולה דיה. עבור רשתות קטנות, יתכן שהשימוש במאיצ' **אף** יאט את תהליך האימון, עקב המחיר של העתקת הנתונים מרכיב אחד לאחר.

## שאלות לתרגול

1. שנו את הקוד לאימון רשת הנוירונים לסייע התמונות של אוסף הנתונים Fashion-MNIST שכתבתם בעבר, כך שלפני האימון תבדקו אם קיים בסביבה מאיצ' גרפי, ואם הוא קיים – הרשת תאומן על גבי המאיצ'. **הנחיות:** אין צורך לכתוב את לולאת האימון מחדש. די להגדיר בסביבה משתנה `device` בעל ערך המתאים לרכיבים חזניים, ולהשתמש במתוודה (`device.to()`).
2. השוו את זמני האימון של הרשת על גבי ה-CPU ועל גבי ה-GPU עבור ארכיטקטורות רשת שונות – רשת קטנה (מעט פרמטרים) ורשת גדולה (רבה פרמטרים).
3. נוסף על חישוב כפלי המטריצה בהתרששות לפנים דרך שכבה לינארית, פעולה זו מבוצעת גם בהתרששות לאחר. כתבו בפיירוט את המטריצות המשתתפות בפעולה זו. הניחו שగרדיאנט פונקציית המחיר לפי פלט השכבה, עבור כל דוגימה ב-`minibatch` הנתון, חושב זה מכבר.

## שכבות Dropout

בעוד רגולרייזציה דרך שינוי פונקציית המחיר, כפי שהציגנו בתחלת ייחודה לימוד זו, היא שיטה נפוצה במעטן מודלים של למידה מכונה, קיימת שיטת רגולרייזציה אפקטיבית במיוחד המביאה בחשבון את מבנה המודלים של למידה عمוקה, ולכן היא שימושית רק עבורה. נסוק בה בפרק זה. בשיטה זו, הנקראת dropout, אנו מנסים למנוע מהרשת לשன את סט האימון באמצעות הוספה רעש לתהליכי האימון: בכל איטרציה של אלגוריתם האופטימיזציה רק חלק אקרים מהנוירונים ישתתף בתהליכי הלימוד. רעיון זה מומחש בשני שלבים.

ראשית, נגידר שכבת dropout מקבלת קלט  $X$  ומפסת כל אחד מרכיביו באקראי, בהסתברות קבועה מראש ובאופן בלתי תלוי.

```
class Dropout(nn.Module):
    def __init__(self, drop_rate=0.5):
        super().__init__()
        assert(0 < drop_rate < 1)
        self.drop_rate = drop_rate
    def forward(self, X):
        mask = torch.rand(X.shape) > self.drop_rate
        return X*mask
```

בכל מעבר קדימה בשכבה מוגרל מחדש של משתנים מקריים אחידים סטנדרטיים, בעל אותו ממך כמו  $X$ , ובטעות זה אנו משתמשים לחשב את המשנה הבוליאני  $.mask$ . זכרו שפונקציית הצפיפות של מ"מ אחיד  $U[0,1]$  היא

$$f(x) = \begin{cases} 1 & x \in [0,1] \\ 0 & x \notin [0,1] \end{cases}$$

ולכן מתקיים

$$P(u > p) = \int_p^1 1 dx = 1 - p$$

מכאן נסיק שבמכפלה  $X * mask$  כל קואורדינטה מאופסת בהסתברות  $drop\_rate$ , ללא תלות בשאר הערכים. שאר הקואורדינטות מועברות הלאה ללא שינוי.

השלב השני בימוש השיטה הוא שילוב שכבות dropout בהגדרת הרשות, לאחר כל שכבת נוירונים. למשל, נוסיף שכבות לרשות של הכל,

```
model = nn.Sequential(nn.Flatten(),
                      nn.Linear(784, 100), nn.ReLU(),
                      nn.Linear(100, 10), nn.ReLU(),
                      nn.Linear(10, 10),
                      nn.LogSoftmax(dim=1))
```

ונקבל את הרשות

```
model_dropout = nn.Sequential(nn.Flatten(),
                             nn.Linear(784, 100), nn.ReLU(),
                             Dropout(),
                             nn.Linear(100, 10), nn.ReLU(),
                             Dropout(),
                             nn.Linear(10, 10),
                             nn.LogSoftmax(dim=1))
```

שימוש לב שלא הוספנו שכבת dropout מייד לאחר הקלט, שכן אנו מעוניינים לתת לרשות את מלאה המידע הקיים בנתונים, וכן לא לאחר השכבה האחורונה, שכן אם ברכזנו להשתמש ברשות לחיזוי סיוג לאחת מ-10 מחלקות, איפוס נוירונים בשכבה זו יפגע קשות ביכולת החיזוי של הרשות.

לאחר הגדרת הרשות מחדש מחדש, יש לבצע את תהליך האימון כבעבר. קיומו של שכבות ה-dropout יביא לרגולריזציה של המודל ללא שניוי נוסף, אך לאחר האימון התעורר בעיה נוספת: בדומה לשימוש בשכבות normalization, akaraiot המובנית בתוך שכבת dropout אינה רצויה בעת החיזוי. במימוש הנוכחי קיבל תוצאות שונות בהתאם להגדרת המ"מ בשכבות אלו. לכן נשנה את הגדרת המעבר קדימה כך שבמצב חיזוי לא יבוצע כל חישוב:

```
def forward(self, X):
    if self.training:
        mask = torch.rand(X.shape) > self.drop_rate
        return X*mask
    else:
        return X
```

זכרו שניתן להעביר את הרשות למצב אימון למצב חיזוי וחזרה באמצעות המתודות `model.eval()` ו-`model.train()`.

כפי שהמעבר קדימה מוגדר כעת, שכבות ה-dropout אין פועלות מוחז למצב אימון. שינוי זה אינו מספק, ואף יפוגם ביכולת החיזוי של הרשות לעומת המצב הקודם: לאורך כל תהליך האימון שכבות ה-dropout החלישו את הפלט מהשכבה הקודמת, ובעת החיזוי הפלט עבר להלאה במלואו, אך לא באופן המתאים לעוצמה שלפיה אומנו הפרמטרים של הרשות. שימוש לב שאמם כל איבר ב-  $X$  נפל בהסתברות  $p$  או השכבה הבאה אומנה לקבל אותן בעוצמה ממוצעת  $X(1-p)$ . לכן התקנון האחרון שנבצע לשכבת dropout הוא להפחית את עוצמתה האות העובר להלאה בעת החיזוי, כפי שניתן לראות בשורה الأخيرة בקטע הקוד המצורף.

```

class DropoutFinal(nn.Module):
    def __init__(self, drop_rate=0.5):
        super().__init__()
        assert(0 < drop_rate < 1)
        self.drop_rate = drop_rate
    def forward(self, X):
        if self.training:
            mask = torch.rand(X.shape) > self.drop_rate
            return X*mask
        else:
            return (1-self.drop_rate)*X

```

השימוש ב-`Dropout` נעשה נפוץ מאוד מאז הצגתו, וערכו רב. קיימים הסברים אפשריים אחדים להצלחת השיטה, ונפרט להלן את שניים מהנפוצים שבהם:

1. שיטה זו דומה לשיטות ensemble קלאסיות של למידת מוכנה: בשיטות אלו מאמנים כמה מודלים (בדרך כלל או זה אחר זה) על אוסף הנתונים, ולבסוף החיזוי שלensemble מתקבל בחילטה משותפת של כל המודלים. כמו ensemble, אשר מطبعו רגש פחות להתקامت יתר (שכן היא צריכה לבוא לידי ביטוי בהתקامت יתר זהה לכל תתי המודלים), כך ניתן לחשב על תהליכי האימון של רשות עם dropout כאימון של מספר רב של מודלים שונים: בכל איטרציה מוגרלת קישוריות שונה בין נוירוני הרשת, והרשת היחדית לאיתרציה זו משaira אוטותיה במשקל הרשות. לבסוף התקוזית המתקבלת לאחר האימון היא מעין ממוצע של התקוזיות הרשות השונות.
2. שיטה זו מונעת יצירת תלות קיצונית בין נוירונים, שכן כדי להצליח בתחזית בנסיבות dropout המודל צריך לפתח יתרות (redundancy): האקטיבציה של נוירון מסוים חייבות להיות מושפעת ממספר רב של נוירונים אחרים, כי שכבת dropout יכולה לאפס כל אחד מהם. בהתחשב בכך, סביר פחות שלאחר האימון יתקבלו משקלים גדולים במיוחד, תוצאה הדומה לרגולרייזציה  $L_2$ .

כמו רכיבים נוספים של רשותנו נוירונים, גם ללא הסבר מדויק או הוכחות תיאורטיות לאפקטיביות השיטה לא נחס למשתמש ב-`Dropout`, שכן ישן שפע הוכחות בשיטה לביצועה המוצלחים בנסיבות פרקטיות.

## שאלות לתרגול

1. א. ב-`PyTorch` שכבות `dropout` ממומשות באופן שונה במעט מהנ"ל: פلت המעביר קידימה בעט האימון הוא  $(1-drop\_rate) * X / (1-drop\_rate)$ , ואילו בעט החיזוי השכבה אינה מבצעת כל שינוי בקלט. הסבירו למה שינוי זה משמש אף הוא פתרון עבור הבעיה של עצמת אות שונה בעט אימון ובעת חיזוי. איזה יתרון חישובי יש למימוש זה על פני המימוש הנ"ל, בהנחה שהרשת המאומנת תשמש לחיזוי אוסף גדול מאוד של נתונים?
2. א. אמנו רשות עמוקה לחיזוי המחלקות של אוסף הנתונים Fashion-MNIST. בחרו רשות עמוקה דיה, כך שבבירור תהיה התקامت יתר.
- ב. הוסיפו לרשות שכבות `dropout` ואמנו אותהשוב. בחרו את הפרמטר `drop_rate` של כל שכבה, כך שמאך אחד לא תהיה התקامت יתר, ומצד שני דיקוק החיזוי לא ייפגע.
- ג. אמנו את הרשות עם שכבות `dropout` אשר אין מתקנות את ההבדל בעוצמת האות של הפלט במצב אימון ובמצב חיזוי.
- ד. השוו את הביצועים של שלוש הגרסאות של הרשות על גבי סט הבדיקה.

## יחידה 5: רשתות קונבולוציה (CNN)

### מוטיבציה לשימוש ברשתות קונבולוציה

ביחידות הלימוד הקודמת למדנו כיצד לאמן רשת נוירונים לסייע תמונות פריטי הלבוש באוסף הנתונים Fashion-MNIST, אך לא השענו מחשבה רבה בבחירה ארכיטקטורת הרשתות שאימנו. דבר זה לא היה בעיה שכן שכבות הרשת תמיד היו בעלות קישוריות מלאה (FC – Fully Connected), כלומר כל נוירון בוחן היה מחובר לכל נוירון בשכבה העוקבת. שכבות אלו בעלות כוח הבעה רב, ובחירה מתאימה של משקליהם מאפשרת אפשרה לחון ללמידה כללים המפרידים בין מחלקות פריטי הלבוש. עם זאת, לכוח הבעה פוטנציאלי זה יש מחיר: השימוש בשכבות אלו יקר מבחינה חישובית, וכן יש בוחן מספר רב של פרמטרים כך שדרוש סט אימון גדול במיוחד כדי לאמן. כל עוד הבעה שאנו עוסקים בה פשוטה דיה, כגון סיוג תמונות שחור-לבן בגודל  $28 \times 28$  פיקסלים ל-10 מחלקות, מחיר זה לא בא לידי ביטוי, אך מהר מאוד, עם העלייה במורכבות הבעה, המחריר נעשה בלתי ריאלי. חשבו לדוגמה על רשת אשר הקלט שלה הוא תמונות צבעוניות בגודל 1 מגה-פיקסל: ממד הקלט הוא 3 מיליון (три מיליון) פיקסלים לכל אחד משולשת ערוצי הצבע), ולכן **לכל נוירון** בשכבה השנייה יהיה 3 מיליון משקלים. אפילו אם השכבה השנייה תהיה קטנה והרשת לא עמוקה במיוחד, הצלות החישובית של מעבר קדימה יחיד תהיה גבוהה, ועל אחת כמה וכמה הצלות של תהליכי האופטימיזציה.

כוח הבעה הרב של שכבות FC מאפשר גמישות, אך לעיתים שלא לצורך: חשבו למשל על כך שבכל הרשתות אשר אימנו עד כה, מיקום הפיקסלים בתמונה לא הובא בחשבון. יתכן שעם תהליך האימון היה הקלט "מעורבב" לפני הזנתו, וזאת מפני שבתחלת תהליכי האימון שיטחנו את טנזור הקלט, וכל איבר של טנзор הקלט השפיע באותה במידה על החישוב העוקב.

לצורך דוגמה, נגיריל תמורה אקראית על שורות התמונות,

```
row_mix = torch.randperm(28)
print(row_mix)

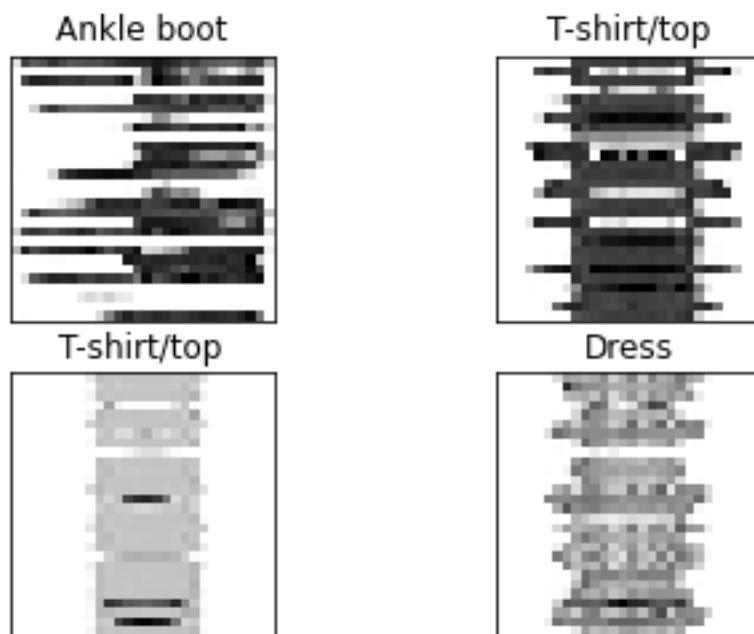
平淡:
```

```
tensor([21,  6, 18,  0, 11, 23,  4, 12, 27,  9,  7, 13, 24,
       3,  5, 16, 22,  8, 20, 26, 19, 14, 10, 17,  1, 25,  2, 15])
```

ונפעיל תמורה זו על השורות של כל התמונות בסט האימון כדלקמן.

```
imgs = imgs[:, :, row_mix, :]
```

זכרו שהמשתנה `img`, הנטען דרך ה-`dataLoader`, מכיל `batch` של תמונות ועל כן הוא טנזור 4-ימדי: הממד הראשון הוא ממד ה-`batch`, השישי והרביעי הם ממדים השורה והעומدة של התמונה (ועל כן גודל כל אחד מהם הוא 28). הממד השני הוא ממד העורץ (channel), אך התמונות באוסף נתונים זה הן בשחור-לבן ולכן בעוניות לרוב יכולו שלושה עורצים עבור עצמת האדום, הירוק והכחול בpixel הנutan. נפעיל את התמורה על 4 פריטי לבוש מקוריים ונאייר זאת להלן.



כאשר פרטיה הלבושים המקוריים הם :



מהאיוריםعلילונים כלל לא ברור לעין אනושית באילו פריטי לבוש מדובר, אולם רשותת הנוירונים שעסקנו בהן עד כה יפגינו ביצועים זהים עבור כל אחת מצורות הקלט, כל עוד נפעיל את אותה הטרנספורמציה על כל התמונות באוסף הנתוני האימון והבדיקה.

ביחידה הנוכחית נותר על עצמה זו, אך בתמורה נקבל שכבות עיליות הרבה יותר, המותאמות ספציפית למשימות העוסקות בעיבוד תמונה וזיהוי פרטיים. עיליות זו תאפשר לנו לאמן רשותות עמוקות יותר, אשר לבסוף, תמורת אותו מחיר חישובי, יבצע עבודה טובה הרבה יותר מרשותות בעלות קישוריות מלאה. הרשותות החדשות יורכבו משכבות קונבולוציה (convolution), השואבות השראה ממערכת הראייה האנושית ומשיטות קלאסיות של שימוש תמונה דיגיטלי. שכבות קונבולוציה מסכמות את השכבה הקודמת באמצעות חילוץ מאפיינים פשוטים (כגון קיום פינה במיקום מסוים בתמונה). באמצעות שרשרת היררכי של השכבות זו לזו, מחלצים לעומק הרשת מאפיינים מורכבים יותר של התמונה המקורית (כגון קיומם של שרול חולצה או שרוכי נעליים). לבסוף, על בסיס המאפיינים שהרשת לחדך מהתמונה, יתבצע הסיווג למחלקות השונות.

בהמשך ייחידה זו תלמדו את הפרטims המדויקים של פעולת רשות כזו, את הרכיבים השונים שהיא מורכבת מהם, וכמוון את מימושם בקוד וシילובם באלגוריתם למידה מבוסס גרדיאנט.

## שאלות לתרגול

1. א. אמוני רשות عمוקה בעלת קישוריות מלאה לחיזוי המחלקות של אוסף הנתונים Fashion-MNIST לאחר שהגרלתם תמורה אקראית  **לכל הפיקסלים** בתמונה והפעלתם אותה על כל תמונה לפני הזנתה.  
ב. בדקו את ביצועי הרשות על סט הבדיקה לאחר שעלה כל תמונה בו הופעלה אותה תמורה.  
ג. בדקו את ביצועי הרשות על סט הבדיקה כאשר לא הופעלה על התמונות בו כל תמורה.
2. אמוני שוב את הרשות כפי שעשיתם בשאלת 1, אך בעת הגירילו מחדש תמורה אקראית עבור כל minibatch. הסבירו את התוצאה שהתקבלה.

## פעולת הקונבולוציה עם גרעין

הפעולה הבסיסית המשמשת לבניית שכבות קונבולוציה אינה המצאה חדשה של מומחי הלמידה העמוקה, אלא פונקציה ידועה בתחום עיבוד התמונה – מדובר בקונבולוציה של תומונות הקלט עם גרעין (kernel). בתחום עיבוד התמונה פעולה זו נקראת לעיתים הפעלת פילטר, וליה מגוון שימושים שונים, כגון טשטוש תמונה, ניקוי רעש או זיהוי קצוטות של אובייקטים.

גרעין הקונבולוציה הוא מטריצה קטנה, אשר ערכיה קבועים בהתאם לאפקט הרצוי על תומנת הפלט.  
למשל השימוש במטריצה

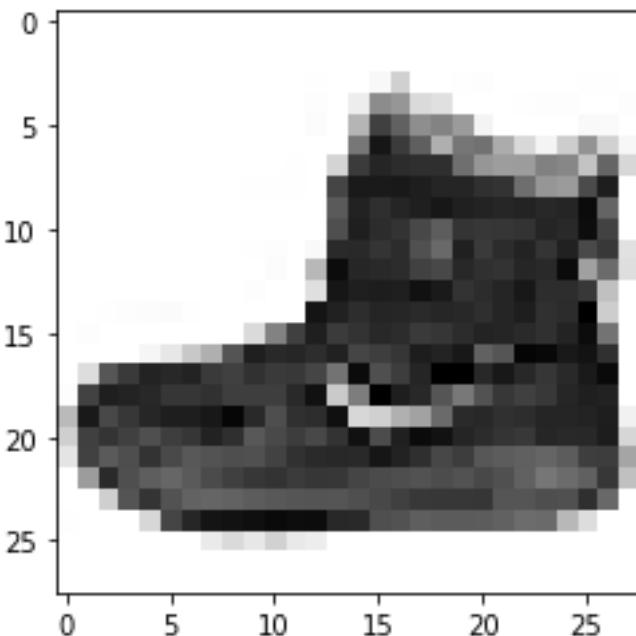
$$K = \begin{pmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{pmatrix}$$

יוביל לטשטוש התמונה, כפי שנראה מייד. להשלמת הדוגמה נתבונן באחת התמונות מהאוסף-Fashion MNIST – תמונה בגודל  $28 \times 28$  פיקסלים אשר כל אחד מהם מיוצג באמצעות מספר טבעי בטוחה 0 (לבן) עד 255 (שחור), כמפורט בקטע הקוד שלהן.

```



```



נגיד בזיכרנו את גרעין הקונволוציה הניל, ולאחר כך נחשב את הקונבולוציה של תמונה המג' עם גרעין זה.

```

kern = torch.tensor([[1/16,1/8,1/16],
                    [1/8,1/4,1/8],
                    [1/16,1/8,1/16]])

```

כעת נסrok את המטריצה של תמונה הקלט משמאלי לימין ומלמعلاה למיטה, ועבור כל תת-מטריצה מסדר  $3 \times 3$  נחשב פיקסל יחיד של תמונה הפלט באמצעות:

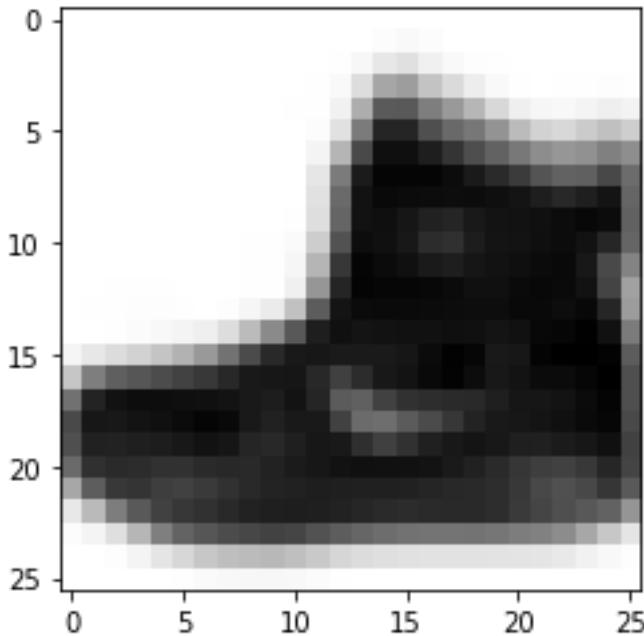
1. כפל חלק התמונה הרלוונטי בגרעין, איבר-איבר.
2. סכימת התוצאה.

```

output = torch.empty(26,26)
for i in range(26):
    for j in range(26):
        sub_img      = img[i:i+3,j:j+3]
        output[i,j] = (sub_img*kern).sum()
plt.imshow(output, cmap='Greys');

```

פלט:



שיםו לב שב קוד הנ"ל המשתנים  $i$ ,  $j$  מסמנים את הפינה השמאלית העליונה של תת-המטריצה  $\text{sub\_img}$ . לפיכך, ערכם המרבי הוא 25, בעוד אינדקס השורה או העמודה המרבי של תמונה הקלט הוא 27. אין זו טעות, שכן כדי ליצור את תת-המטריצה  $\text{sub\_img}$  יש צורך בnockותם של עוד שני איברים מימין ו מתחת ל- $i$ ,  $j$  במטריצת הקלט. ניתן לראות אפוא שפלט פועלות הקונבולוציה הוא תמונה קטנה יותר, בגודל  $26 \times 26$ . לעיתים הקטנת הממד תהיה שימושית עבורנו, ולעתים נרצה לשמור על גודל התמונה המקורי. בהמשך היחידה נדון בשיטות לשיליטה בממד הפלט.

אין זה הכרחי שגרעינו הפעולה יהיה ריבועי, ראו למשל דוגמה נוספת שבה נבצע קונבולוציה של התמונה עם הגרעין :

$$K = \begin{pmatrix} -1 & 1 \end{pmatrix}$$

```

kern    = torch.tensor([-1,1])
output = torch.empty(28,27)
for i in range(28):
    for j in range(27):
        sub_img      = img[i,j:j+2]
        output[i,j] = (sub_img*kern).sum()
print(output.size())
plt.imshow(output,cmap='Greys');
plt.colorbar();

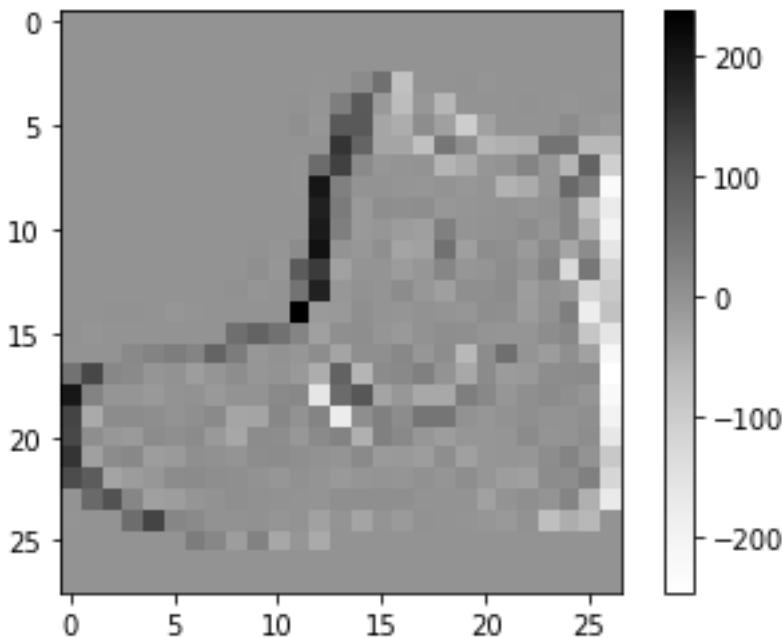
```

פלט:

```

torch.Size([28, 27])

```



לעיל ניכר כי רוב מטריצת הפלט מלאה בערכים סביבה האפס (אפור), אך יש ערכים גבוהים במקומות שבתמונה המקורית היה מעבר לבן לשחור (בתנועה משמאל לימין) וערכים נמוכים מעבר הפוך. לאור זאת ניתן להגיד שהגרעין הנ"ל הוא למעשה מזהה קצוטות אופקי. ראו גם כי פלט פועלות הקונבולוציה הפעם הוא מממד  $28 \times 27$ , שכן לא בוצע כל חישוב עבור העמודה الأخيرة בתמונה המקורית: מימין לעמודה זו אין פיקסלים שבזורתם ניתן להגיד את תת-המטריצה  $img\_sub$ .

مدוגמאות אלו נקיש על הכלל. בהינתן מטריצת קלט מסדר  $H \times W$  (המייצגת תמונה בעלת עroz צבע ייחיד: שחורי-לבן),

$$, X = \begin{pmatrix} x_{1,1} & \cdots & x_{1,W} \\ \vdots & \ddots & \vdots \\ x_{H,1} & \cdots & x_{H,W} \end{pmatrix}$$

ומטריצה קטנה מסדר  $q \times p$ , אשר תשמש כגרעין,

$$, K = \begin{pmatrix} k_{1,1} & \cdots & k_{1,q} \\ \vdots & \ddots & \vdots \\ k_{p,1} & \cdots & k_{p,q} \end{pmatrix}$$

תוצאת פועלות הקונבולוציה של  $X$  עם  $K$  היא מטריצה  $Y$  מסדר  $(H - p + 1) \times (W - q + 1)$  בעלת הערכים הבאים:

$$y_{r,s} = \sum_{i=1}^p \sum_{j=1}^q x_{r+i-1, s+j-1} k_{i,j}$$

שימוש לב שלפי נוסחה זו, כדי לחשב את  $y_{r,s}$ , יש:

1. לחותוך מ- $X$  תת-מטריצה מסדר  $q \times p$  אשר הפינה השמאלית العليا שלה היא  $x_{r,s}$ ,
2. לכפול תת-מטריצה זו איבר-אייבר במטריצת הגרעין,
3. לסכום את התוצאה.

מהגדרת הפעולה נבון גם את ממד הפלט: ניתן לחשב את  $y_{r,s}$  כל עוד מתחת ל- $x_{r,s}$  יש עוד  $p - 1$  שורות וימין לו  $1 - q$  עמודות.

לסיום פרק זה נזכיר שבוד בתחומי עיבוד התמונה והלמידה העמוקה הפעולה המתוארת לעיל קרוייה קונבולוציה, שמה המתמטי הוא קורלציה צולבת (cross-correlation), והפעולה המתמטית הקרוייה קונבולוציה שונה כמעט לחלוטין. למרות זאת ממשיך, מקובל בתחום, להשתמש בשם קונבולוציה עבור הפעולה הרלוונטי לשימושנו.

## שאלות לתרגול

1. כתבו פונקציית פיתון המקבלת מטריצת קלט וגרעין כלשהו ומחשבת את הקונבולוציה שלהם. על הפונקציה לבדוק את תקינות הקלט, כולל שבסך הכול ניתן לבצע את הקונבולוציה (יש לבדוק שגרעין הקונבולוציה אינו גדול מדי).
2. תכנו גרעין  $3 \times 3$  המזהה קצוטים אלכסוניים והפעילו אותו על התמונה של המגף הניל.

## מבנה שכבה קונבולוציה ברשתות נירוניים

את פועלות הקונבולוציה אשר הכרנו בפרק הקודם נרטום בעבור רשתות נירוניים בעליות קלט ויזואלי. לשם כך, הצעד הראשון שננקוט הוא הפיכת גרעין הקונבולוציה **לפרמטרים** של רשף הנירוניים: בעודו מטריצת גרעין מוגדרים את גרעין הקונבולוציה מראש עבור שימוש ידוע, כאשר בשלב השני זו במלידה עמוקה נניח לערכיו מטריצת הגרעין להיקבע לפי אלגוריתם האופטימיזציה. משמעות הדבר היא שהרשף תלמד בעצמה גרעינים המחלצים מאפיינים רלוונטיים למטרותיה. אם כן, נתחיל כמנהגנו בהגדירה ראשונית של שכבה קונבולוציה, ונשפר אותה בהמשך היחידה.

נניח כי הקלט לשכבה הוא minibatch בעל  $N$  דוגמאות של תמונות בגודל  $W \times H$ , שלחן ערוץ קבוע יחיד. לפיכך, גודל טנזור הקלט יהיה  $W \times H \times N$ , כאשר הקונבולוציה תופעל בנפרד על כל תמונה ב-minibatch.

```
class ConvLayer_1(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.kern = nn.Parameter(torch.randn(kernel_size))
        self.p, self.q = kernel_size
    def forward(self, X):
        output = torch.empty(X.size(0),
                             X.size(1)-self.p+1,
                             X.size(2)-self.q+1)
        for i in range(output.size(1)):
            for j in range(output.size(2)):
                sub_img = X[:, i:(i+self.p), j:(j+self.q)]
                output[:, i, j] = (sub_img * self.kern).sum(dim=(1, 2))
        return output
```

במימוש זה בקוד מופיעים כמה פרטים מעניינים:

- ראשית שימושו לב שגרעין הקונבולוציה מאותחל באקראי ומוגדר כאובייקט `nn.Parameter` שכן הוא ישתתף בתחום האימון, ובעת ביצוע צעד של אובייקט האופטימיזציה נרצה שאגן הוא יעודכן.
- שנית, ראו שבעת חישוב האיבר  $i, j$  של הפלט, הגרעין משודר לאורך ממד ה-`batch` של התמונה, ואחרי כן מתוזמת הסכום מבצעת הפקתת על ממד הגובה והרוחב של התמונה, כך שנשאר סקלר ייחיד עבור כל דוגמה ב-`batch`.
- לבסוף, שימושו לב לגמישות המובנית בתוך השכבה: בNetworking לשכבות לינאריות, במקרה שכבה זו לא דורש את ממד הקלט והפלט מראש. בעת המעבר קדימה לשכבה ייקבע ממד הפלט לפי חוקיות הקונבולוציה. משמעות הדבר היא שנייתן להשתמש בשכבה מסווג זה עבור קלט המשתנה בגודלו.

הפרט המעניין ביותר בשכבה הקונבולוציה הוא שבעת המעבר קדימה אנו מבצעים רק פעולות כפל וחיבור בין הגרעין לבין איברי מטריצת הקלט, וכך בעת התפשטות הגרדיינט לאחר לא תתעורר בעיה – שהרי אלו פעולות גזירות (הגם שמערכת `autograd` תטפל בכך עבורנו).

נדגים זאת באמצעות אתחול שכבת קונבולוציה ואימוננה באמצעות SGD לזיהוי קצוטות אנכיתים : נגדיר פילטר זיהוי קצוטות בעל הגרעין

$$K = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

בעזרת המחלקה שכתבנו זה עתה :

```
edge_detector = ConvLayer_1((3,3))
edge_detector.kern.requires_grad = False
edge_detector.kern[:] = torch.tensor([[-1.,-1.,-1.],
                                      [0,0,0],
                                      [1,1,1]])
```

ראו כי כיבינו את מעקב מערכת `autograd` על גרעין מזזה הקצוטות, שכן הוא יהיה המטרה שאליה שכבת הקונבולוציה שנגידיר להלן תתאים עצמה לאורך תהליכי האופטימיזציה.

```
my_conv = ConvLayer_1((3,3));
optimizer = torch.optim.SGD(my_conv.parameters(), lr=0.1)
```

כל שנותר לעשות הוא לכתוב את LOLאט האימון, שבה נטען של תמונות מהאוסף-Fashion-MNIST, נורמל אותן למספרים ממשיים בטוחה 0 עד 1 ונעביר אותן דרך פילטר זיהוי הקצוטות ושכבת הקונבולוציה שלנו. המרחק בין שתי התוצאות ישמש כפונקציית ההפסד של האופטימיזציה, וכן גרעין השכבה יעדכן לאורך התהליכי ליצר פלט דומה לזה של `edge_detector`. לאחר טעינת התמונות לתוך המשתנה `imgs` והמרתן לטנзорים מנורמליים, ליבת LOLאט האימון תיראה כך :

```
optimizer.zero_grad()
my_transformed_imgs = my_conv(imgs)
target_imgs       = edge_detector(imgs)
loss = ((my_transformed_imgs-target_imgs)**2).mean()
loss.backward()
optimizer.step()
```

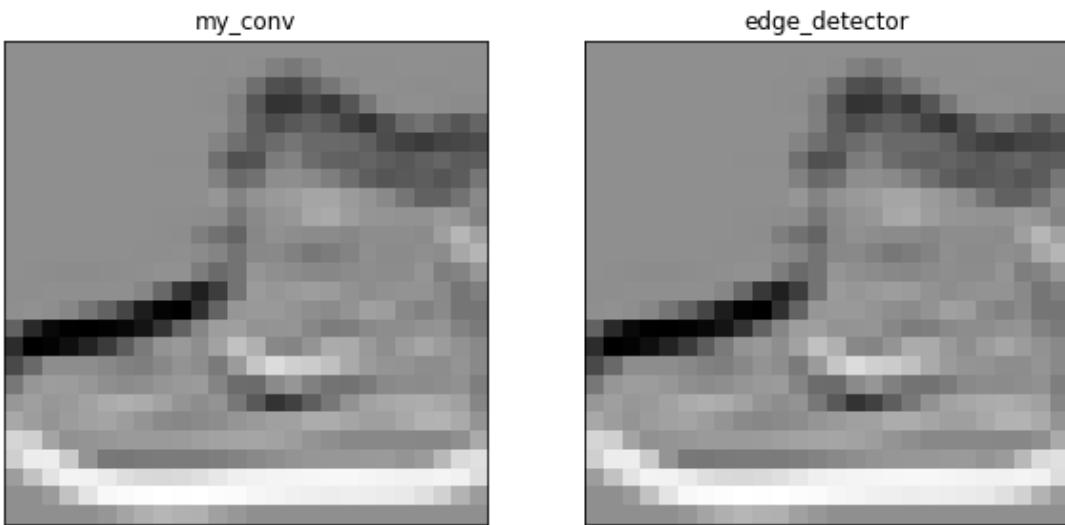
יחיד של אימון מניב את גרעין הקונבולוציה שלhalten, וניתן לראות בו את הדמיון לגרעין מזזה הקצוטות.

```
print(my_conv.kern)
```

**פלט:**

```
Parameter containing:
tensor([[-0.8108, -0.7242, -0.6472],
       [-0.0661,  0.0861, -0.0626],
       [ 0.5860,  1.0309,  0.5985]], requires_grad=True)
```

באյור המצורף נראה שהפעלת הפילטרים על אותה תמונה מניבה תוצאה כמעט זהה. אם כן, תהליך הלמידה הסתיים בהצלחה.



- כבר בשלב זה אנו יכולים לבדוק ב יתרונות של שכבות אלו עבור קלט ויזואלי :
- ראשית, מספר הפרמטרים עבור שכבה נתונה קטן במידה ניכרת מאשר בשכבה בעלת קשריות מלאה – יש למודד את הגרעין בלבד.
  - שנית, רק מספר מועט של פיקסלים הממוקמים בסמיכות זה לזה בתמונה הקלט משפיעים על חישוב פיקסל של הפלט. כך אנו כופים על הרשות ללמידה כללים המבאים בחשבון אינטראקציה מקומית בלבד, כללים אשר לצופה האנושי ברור שקיים במידע ויזואלי : קיומו של שרול חולצה **במקום מסויים בתמונה**, למשל, מעיד על כך שזו תמונה של חולצה.
  - נוסף על כך, השימוש החזר באוטו גרעין עבור ת特-מטריצות שונות של תמונה הקלט מוביל את הרשות ללמידה כללים שאינם תלויים במיקום ספציפי למרחב (אין זה משנה היכן בתמונה נמצא השROL הני'ל).
  - לבסוף, פועלות הקונבולוציה זולה הרבה יותר מחישוב התפשטות לפנים בשכבה לינארית מלאה, שוב עקב לכך שכל פיקסל בפלט מחושב על בסיס מעט פועלות חשבון.

## שאלות לתרגול

- תמונה הקלט בצורתן המקורית מכילות ערכים בטוחה 0 עד 255. נסו לאמן את שכבת הקונבולוציה ללמידה את גרעין זיהוי הקצוות ללא נרמול לטוחה 0 עד 1. אם האימון נכשל, הסבירו מדוע.
- אסטרטגיית אתחול פרמטרים מוצלחתتبיא בחשבון את גודל הגרעין. שנו את בנייה שכבת הקונבולוציה כך שאיברי הגרעין יאותחלו באמצעות דגימה מההתפלגות  $\mathcal{U}\left[-\frac{1}{\sqrt{K}}, \frac{1}{\sqrt{K}}\right]$  כאשר  $K$  הוא מספר האיברים בגרעין. רמז: אם  $u \sim \mathcal{U}[0,1]$  אז  $au + b \sim \mathcal{U}[b, a+b]$ .
- חשבו באופן אנלטי את ההתפשטות לאחרו של השגיאה דרך שכבת קונבולוציה בעלת גרעין בגודל  $1 \times 2$ , כאשר ידוע שתמונה הקלט היא בגודל  $3 \times 2$ . כרגע, הניחו שגרדיאנט המחיר לפוי פלט השכבה,  $\frac{\partial C}{\partial Y}$ , חושב זה מכבר, ועליכם לחשב רק :

- את הגרדיאנט של הפלט לפי הפרמטרים (גרעין הקונבולוציה),  $\frac{\partial Y}{\partial K}$ , לצורך עדכוןם בעת האופטימיזציה,

- ואת הגרדיאנט לפיקסל השכבה,  $\frac{\partial Y}{\partial X}$ , לצורך המשך התפשטות השגיאה.

רמז: הפרידו למקרים לפי מספר הפעמים שפיקסל מסוים של הקלט משתתף בחישוב פיקסל בפלט.

## מבנה שכבת קונבולוציה: ריבוי ערוצים

לשכבה הקונבולוציה אשר כתבנו בפרק הקודם יש להוסיף כמה רכיבים, לפני שנוכל להשתמש בה למשימות מיידת מורכבות. בפרק זה נתחיל לטkor רכיבים אלו, ונשלבם בהגדרת השכבה.

### ריבוי ערוצי פלט

שכבת הקונבולוציה שלנו בשלב זה תלמד גרעין יחיד בהינתן תמונה הקלט. עם זאת, ייתכן שיש לנו מאפיינים חשובים אחדים שצדאי לרשף לחץ מההתמונה לצורך המשך הלמידה (זיהוי קצוטות אופקיים, זיהוי קצוטות אנכיים, זיהוי פינות וכו'). על כן, נרצה לאפשר לשכבה אחת ללמידה **כמה גרעינים שונים** באופן בלתי תלוי זה זהה, כאשר כל אחד מהם יחשב קונבולוציה נפרדת של הקלט, והפלט שלו ישמר **ערוצי פלט נפרדים**. אם כן, עבור טנזור קלט בגודל  $N \times H \times W$  בגודל  $N$  תמונות מסדר  $W \times H$ , פלט השכבה יהיה טנзор בממדים  $(H - p + 1) \times (W - q + 1) \times C \times N$ , כאשר הממד השני יכול את תוצאות הקונבולוציות של התמונות עם הגרעינים השונים, כולם מסדר  $q \times p$ . קוד השכבה החדש הוא

```
class ConvLayer_2(nn.Module):
    def __init__(self, out_channels=1, kernel_size=(1,1)):
        super().__init__()
        self.kern = nn.Parameter(
            torch.rand((out_channels,*kernel_size)))
        self.p , self.q    = kernel_size
        self.out_channels = out_channels
    def forward(self, X):
        output = torch.empty(X.size(0),
                             self.out_channels,
                             X.size(1)-self.p+1,
                             X.size(2)-self.q+1)
        for i in range(output.size(2)):
            for j in range(output.size(3)):
                sub_img = X[:,i:(i+self.p),j:(j+self.q)]
                sub_img = sub_img.unsqueeze(1)
                output[:, :, i, j] = (sub_img * self.kern).sum(dim=(2,3))
        return output
```

כאן יש לשים לב לכמה פרטים חדשים:

- ראשית, בעת אתחול השכבה יש לציין את מספר ערוצי הפלט המבוקשים. פרט זה הכרחי שכן בעת ייצרת אובייקט ממחלקה זו יש לאותל מספר גרעינים שונים השווה לפרמטר זה.
- שנית, בעת דגימות המ"מ לאותל גרעיני השכבה, אנו "פותחים" את הזוג הסדר `kernel_size` לשני ערכים נפרדים בעורת אופרטורו `h*` ומימד מאגדים אותו לשירה סדורה עם הפרמטר `out_channels`. אנו עושים זאת לאחר שגודל הטנзор הנדגם מועבר ל' `torch.rand` בתוך `kernel_size`. מילה סדורה ייחידה.
- לבסוף, חישוב כל פיקסל בכל ערוצי הפלט מתבצע בשורת הקוד הקודמת לסוף, וזאת באמצעות שידור תתהתמונה `sub_img` לאורך ממד הערוצים ושידור הגרעינים לאורך ממד `batch`. מכיוון

שלתת-התמונה עוד אין ממד המתאים לעורçi הפלט החדש (גודלה הוא  $q \times p \times q$ ), קודם לשידור אנו מכנים ממד נוסף לאחר הממד הראשון, בעזרת המתוודה (1) `unsqueeze` (גודל תמתת-המונה אחריו שימוש במתוודה זו הוא  $q \times p \times 1 \times N$ ). ללא חוספת ממד מלאכותי זה קיבל תוצאה שגויות: חוקיות השידור תשווה את ממד העורץ של הגרעינים (טנוור בגודל  $C \times p \times q$  לממד ה-`batch` של תמתת-המונה, דבר אשר יוביל לרוב לשגיאה, ובמקרים נדירים לשילוב לא רצוי של רכיבים מממדים שונים).

לצורך הדוגמה ניצור שכבה קונבולוציה בעלת שני עור齊 פלט, נגידר במפורש את הגרעינים שלה ונפעילה על קלט לדוגמה.

```
edge_detector2 = ConvLayer_2(2, (3, 3))
filters = torch.tensor(
    [[[[-1., -1, -1],
      [0, 0, 0],
      [1, 1, 1]],

     [[[1, 0, 1],
       [-1, 0, 1],
       [-1, 0, 1]]]])
edge_detector2.kern.requires_grad = False
edge_detector2.kern[:] = filters
```

בקטע קוד זה הגדרנו את גרעיני הקונבולוציה של השכבה להיות

$$\text{kern}[0,...] = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \quad \text{kern}[1,...] = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

כך שהעורץ הראשון יזהה קצוות אופקיים והשני קצוות אנכיים.

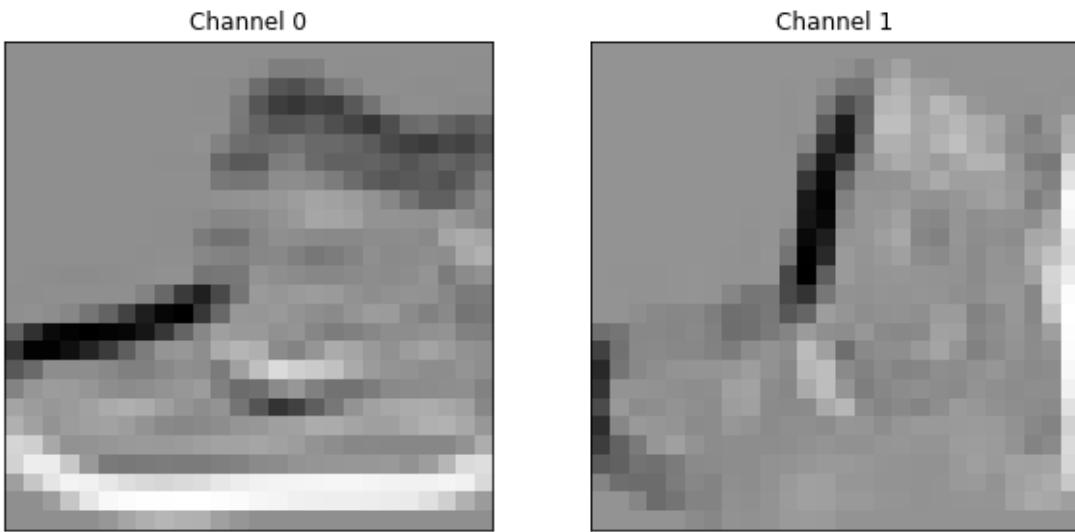
בהנחה שלtopic המשתנה `imgs` טענו כבר 512 תМОונות, הפעלת השכבה עליו תניב את הטנוור שלහן:

```
edges_detected = edge_detector2(imgs)
print(edges_detected.size())
torch.Size([512, 2, 26, 26])
```

**פלט:**

בטנוור זה נמצאת התוצאה של שתי הקונבולוציות עבור כל תמונה ב-`batch`. באյור המצורף ניתן לראות את שני העורצים עבור התמונה הראשונה:





## ריבוי ערכזים קלט

מטרתנו היא להשתמש בשכבות קונבולוציה בראשת עמוקה, שבה פלט שכבה אחת מועבר לשכבה הבאה בתור. לפיכך, علينا לשנות את חישוב הקונבולוציה כך שריבוי הערכזים בקלט יבוא בחשבון, שכן אף אם קלט הרשות היה תמונה בעלת ערך צבע יחיד, בפלט השכבה הראשונה כבר יהיו כמה ערכזים שונים, והחל מהשכבה השנייה לא יהיה די בפעולת הקונבולוציה שהשתמשנו בה עד כה. מובן שהפונקציונליות שנosis' לשכבה CUT האפשר לנו גם להזין לרשות תמונות צבעוניות.

את המעבר לריבוי ערכזים הפלט עשינו באמצעות הגדלת מספר הגרעינים וחישוב קונבולוציות אחדות בויזמנית, אולם שינוי זה אינו מתאים עבור ריבוי ערכזים הקלט. ערכזים הקלט השונים של פיקסל כלשהו בתמונה נתונה מכילים מידע רלוונטי לפיקסל זה וסביבתו: קיומה של פינה באזורי מסוימים בתמונה יכול לבוא לידי ביטוי בשינויים בשלושת ערכזים הצבע של תמונה צבעונית, למשל. לפיכך, נרצה עתה שפעולות הקונבולוציה תשלב את המידע הקיים בערכזים הקלט השונים, כך שעבור תמונה יחידה בעלת  $C$  ערכזים, פלט פעולה הקונבולוציה (עם גרעין נתון) תהיה בעלת ערך יחיד.

על כן علينا לשנות את הגדרתה של הקונבולוציה עצמה, וכן להגדיל את ממד הגרעין: עבור טנזור קלט  $X$  בגודל  $C \times H \times W$  (המייצג תמונה בעלת  $C$  ערכזים באורך  $H$  ורוחב  $W$ ), וגרעין קונבולוציה  $K$  בגודל  $C \times p \times q$  (כאשר, כמובן,  $p$  ו- $q$  הם פרמטרים קבועים), פלט פעולה הקונבולוציה של  $X$  עם  $K$  יהיה שוב מטריצה  $Y$  בגודל  $(H - p + 1) \times (W - q + 1)$ , בעלת הערכים

האלה:

$$\cdot y_{r,s} = \sum_{c=1}^C \sum_{i=1}^p \sum_{j=1}^q x_{c,r+i-1,s+j-1} k_{c,i,j}$$

- זו הכללה טبيعית של הקונבולוציה עם ערך יחיד, שכן כדי לחשב את  $y_{r,s}$  יש:
1. לחותוך מי  $X$  תת-טנзор בגודל  $C \times p \times q$  אשר הפינה השמאלית העליונה שלו במדדי האורך והרוחב היא בעלת האינדקסים  $r, s$ .
  2. לכפול כל ערך של תת-טנзор זה, איברא-איבר, **ערך המתאים בטנзор הגרעין**,
  3. לבסוף לסכום את התוצאה.

בקטע הקוד שלහלן נרחיב את הגדרת שכבת הקונבולוציה המקורית, כך שתתקבל טנוור קלט בגודל  $N \times C \times H \times W$  תמונות בעלות  $N$  minibatch :  $N \times C \times H \times W$  ערוצים. פلت השכבה יהיה טנוור תלת-ממדי : תוצאה הקונבולוציה עבור כל תמונה ב-`batch`.

```
class ConvLayer_3(nn.Module):
    def __init__(self,in_channels=1, kernel_size=(1,1)):
        super().__init__()
        self.kern = nn.Parameter(
            torch.rand((in_channels,*kernel_size)))
        self.p , self.q = kernel_size
        self.in_channels = in_channels
    def forward(self, X):
        output = torch.empty(X.size(0),
                             X.size(2)-self.p+1,
                             X.size(3)-self.q+1)
        for i in range(output.size(1)):
            for j in range(output.size(2)):
                sub_img      = X[:, :, i:(i+self.p), j:(j+self.q)]
                output[:, i, j] = (sub_img * self.kern).sum(dim=(1,2,3))
        return output
```

ראו כי שכבה זו בעלת גרעין קונבולוציה יחיד (תלת-ממדי), וכן שהפחתת הסכום מתבצעת על כל הממדים, מלבד ממד `batch`.

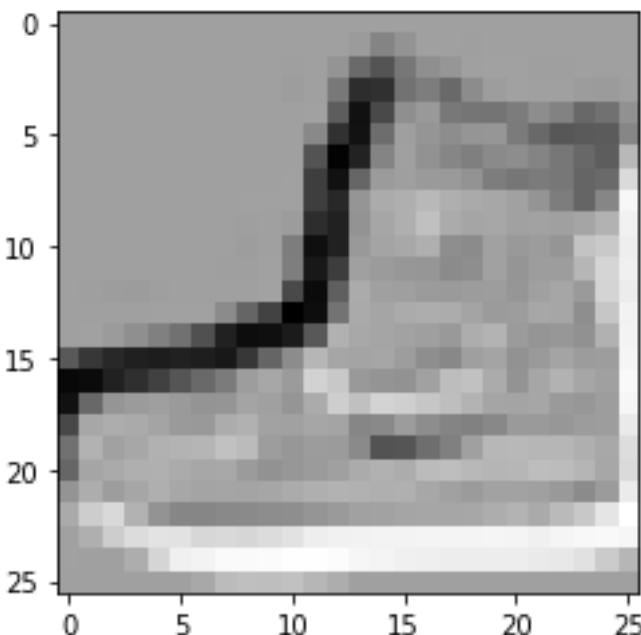
המשך עם הדוגמה הקודמת שבה יצרנו במשתנה `edges_detected` טנוור בעל שני ערוצים : מזזה קצוטות אופקי ומזזה קצוטות אנכי. נשלב את המידע מהערוצים השונים בעזרת קונבולוציה עם גרעין  $1 \times 1$  ושני ערוצי קלט. לאחר הפעלת גרעין זה רוחב ואורך התמונה לא ישתנו, אך בכל פיקסל נקבל צירוף לינארי של ערוצי הקלט. נבחר למשל גרעין המחשב ממוצע של ערכי הערוצים השונים ונקבל את התוצאה הזו.

```

aggregate      = ConvLayer_3(2, (1,1))
with torch.no_grad():
    aggregate.kern[:] = torch.tensor(0.5).expand(2,1,1)
edges_aggregated = aggregate(edges_detected)

print(aggregate.kern.size())
print(edges_aggregated.size())
plt.imshow(edges_aggregated[0,...].detach(), cmap='Greys');
    פלט:
torch.Size([2, 1, 1])
torch.Size([512, 26, 26])

```



ראו כיצד פלט השכבה מזוהה קצווות אופקיים או אנכיים.

### שאלה לתרגול

כתבו שכבה קונבולוציה בעלת מספר ערוצי קלט ופלט: הקלט לשכבה יהיה טנזור בגודל  $C_{out} \times N \times C_{in} \times H \times W$  והפלט – טנзор בגודל  $N \times C_{out} \times (H - p + 1) \times (W - q + 1)$ . השכבה תחשב קונבולוציות שונות (עם גרעינים שונים) על כל אחת מ- $N$  תמונות הקלט. חתימת בנייה השכבה תהיה

```
__init__(self, in_channels=1, out_channels=1, kernel_size=(1,1))
```

## רכיבים נוספים של שכבות קונבולוציה

לאחר שהוספנו ערוצי פלט וקלט רבים לשכבות הקונבולוציה בפרק הקודם, נסקור כעת את הרכיבים האחרונים הנדרושים כדי להשתמש בהן במשימות למידה.

### ריפוד (padding)

בפרקים הקודמים פגשנו בתוכנה יסודית של פועלות הקונבולוציה: גודל תמונה הפלט קטן מהתמונה המקורי, וכך גם גודל הפלט המקורי הקטן ממד זו אוינה רצiosa, וכך ניתן לשנות על גודל הפלט אפשרות להגדיל באופן מלאכותי את הפלט של השכבה באמצעות הוספה שורות ועמודות בקצות התמונה. שיטה זו נקראת ריפוד (padding) וישנם כמה שימושים שלה PyTorch, בהתאם לערכיהם המוצבים בעמודות או בשורות חדשות. הבחירה הנפוצה ביותר היא ריפוד באפסים. ראו בדוגמה שלහן כיצד אנו מRADIMים טזוזר קלט בשורת אפסים אחת ובעמודות אפסים אחת נוספת מכל צד.

```
pad_layer = nn.ZeroPad2d(1)
A = torch.arange(1,10).reshape(3,3)
print(A, pad_layer(A), sep='\n')

tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
tensor([[0, 0, 0, 0, 0],
        [0, 1, 2, 3, 0],
        [0, 4, 5, 6, 0],
        [0, 7, 8, 9, 0],
        [0, 0, 0, 0, 0]])
```

**פלט:**

טרנספורמציה זו מומשת גם בתוך שכבת הקונבולוציה הסטנדרטית של PyTorch: בשורת הקוד שלහן אנו מגדירים שכבת קונבולוציה המקבלת minibatch של תמונות קלט בעלות ערוץ אחד, טזוזר בגודל  $W \times H \times 1$ , ומהזירה פלט באותו גודל בדיק. בהתאם לגודל הגרעין, השכבה בוחרת אוטומטית את מספר השורות והעמודות של הריפוד.

```
conv_layer = nn.Conv2d(in_channels=1, out_channels=1,
                      kernel_size=(3, 5), padding="same")
```

### פרמטר bias

לשכבת הקונבולוציה המובנית ב-Pytorch שני רכיבים נוספים שעוז לא דנו בהם. הראשון נגוע שווים כדי התבוננות ברשימה הפרמטרים בשכבה שזה עתה הגדרנו.



```

for params in conv_layer.named_parameters():
    print(params)

平淡:
('weight', Parameter containing:
tensor([[[[-0.2093, -0.2208,  0.0425,  0.1462, -0.1257],
          [ 0.0535, -0.0787, -0.0518,  0.1329,  0.2383],
          [ 0.0557,   0.1856, -0.1075, -0.2001,  0.1463]]],,
       requires_grad=True))
('bias', Parameter containing:
tensor([-0.2201], requires_grad=True))

```

ראו שהפרמטר `weight` מכיל את גרעין הקונבולוציה בגודל 5X3 (עם שני ממדים נוספיםinos – עבור ערכיו הקלט והפלט), כפוי. נוסף על כך, יש לשכבה פרמטר בשם `bias`, המכיל ערך נפרד לכל ערוץ פלט (אחד במקרה שלנו), ומוסיפים אותו לתוצאת הקונבולוציה בערוץ זה. הצורך בפרמטר זה נובע ממבנה הרשת שבה שכבת הקונבולוציה תמצא שימוש: לרוב, פלט השכבה יעבור מיידית לאקטיבציה לא לינארית, כגון ReLU, שבעוראה דרוש ה-`bias` כדי לווסת את הסף שבו הנירון מופעל. אין לנו מחרירים שכבות קונבולוציה זו לזו ללא אקטיבציה לא לינארית ביןיהן, שהרי הקונבולוציה היא פולה לינארית בסיסה, ועל כן שרוור קונבולוציות זו לאחר זו שקול לביצוע פולה לינארית ייחידה.

## גודל פסיעה (stride)

הרכיב הנוסף השני של שכבות קונבולוציה שנדרשו בו כתה הוא פרמטר גודל הפסיעה. השתמשנו בריפוד הפלט למטרת הגדלת הפלט, אולם לעיתים נרצה דוקא את התוצאה ההפוכה – לצמצם את גודל התמונה לאחר המעבר בשכבה, למשל כאשר גודל הפלט גדול מדי לאימון הרשת ביעילות, או כאשר מעוניינים לקבל לאחר מספר מועט של שכבות מאפיינים היוצרים קשר בין אזורים מרוחקים בתמונות הפלט.

בעבר עברנו על **כל פיקסל** בפלט (אשר מימיינו ומתחתיו ישנים די איברים, כגודל הגרעין) וחתכנו מהפלט מטריצה בגודל הגרעין לטובת חישוב פיקסל יחיד בפלט, ואילו כתה נעשה זאת רק עבור חלק מהפיקסלים לפי חוקיות פשוטה, שנבין מהדוגמה שלהלן:

```

downsample = nn.Conv2d(in_channels=1, out_channels=1,
                     bias=False, kernel_size=(1, 1),
                     stride=(2, 3))
with torch.no_grad():
    downsample.weight[:] = torch.tensor(1.).reshape(1, 1, 1, 1)
A = torch.arange(25, dtype=torch.float).reshape(1, 1, 5, 5)
print(A, downsample(A), sep='\n')
פלט:
tensor([[[[ 0.,  1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.,  9.],
          [10., 11., 12., 13., 14.],
          [15., 16., 17., 18., 19.],
          [20., 21., 22., 23., 24.]]]])
tensor([[[[ 0.,  3.],
          [10., 13.],
          [20., 23.]]]], grad_fn=<ThnnConv2DBackward0>)

```

בדוגמה זו הגדכנו שכבת קונבולוציה ללא bias עם גרעין בגודל  $1 \times 1$ , ואת ערך הגרעין קבענו להיות 1. קונבולוציה עם גרעין זהה, עברו קלט בעל ערך יחיד, אמורה להניב את פונקציית הזוזות, אך שימו לב שהעברנו לבניית השכבה גם פרמטר `stride`, גודל הפסיעה. משמעות הזוג הסדור שהועבר בפרמטר זה היא שבעת חישוב הקונבולוציה יש לחשב ולשמור את ערך מכפלת תחתה מטריצה בגרעין עבור אחד מכל שלושה איברים בשורה (כל עוד ניתן לבצע חישוב זה, כאמור), ואחת מכל שתי שורות. התוצאה המתקבלת עבור הגרעין המסוים שבחרנו היא שرك האיברים מהעומודה הראשונה או הריבועית הנמצאים בשורות האיזוגיות נשמרים בפלט.

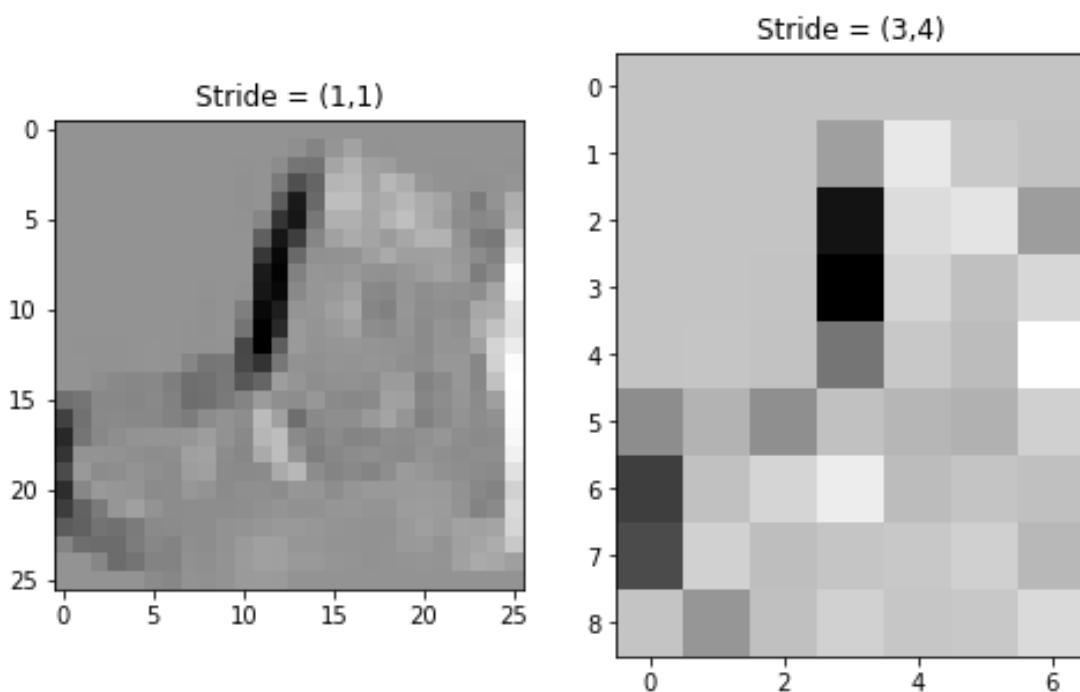
ובן שניינו לשלב את פרמטר הפסיעה עם כל אחד מרכיבים האחרים של שכבת הקונבולוציה. למשל, נגיד מחדר את מזזה הקצויות האופקיים ששימש כדוגמה בפרק הקודם עם פסיות של שורה אחת מתוך כל שלוש ועמודה אחת מתוך כל ארבע בקטע הקוד שלහן, ונשווה את תוצאות הפילטר החדש **למקורו באירור העוקב**.

```

edge_detector2 = nn.Conv2d(in_channels=1, out_channels=1,
                           bias=False, kernel_size=(3, 3),
                           stride=(3, 4))
with torch.no_grad():
    edge_detector1.weight[:] = torch.tensor(
        [[-1., 0, 1], [-1, 0, 1], [-1, 0, 1]]).reshape(1, 1, 3, 3)

```





## איגום (pooling)

שיטה נוספת לצמצום ממד הפלט היא העברת הפלט של שכבת הקונולוציה (לאחר אקטיבציה) לשכבה איגום (pooling layer). שכבה זו תשקל את הנתונים של כמה פיקסלים סמוכים זה לזה, למשל באמצעות חישוב הממוצע שלהם, לייצור מאפיין יחיד בפלט. כתוצאה יתקבלו מאפיינים אשר אינם רגילים לתוצאות קטנות של הקלט, דבר הרצוי לרוב: מעניין אותנו אם קיימים בתמונה הקלט שרוכי נעלים או לא, ולאו דווקא המקום המדוקדק שלהם.

ראו בדוגמה המצורפת כיצד אנו מגדירים שכבת איגום אשר מחלקת את הקלט לחתימות רוחות מסדר  $2 \times 3$  ומחזירה מאפיין יחיד לכל חתימת – הערך המרבי בה.

```
pool = nn.MaxPool2d(kernel_size=(2, 3))
A = torch.arange(40, dtype=torch.float).reshape(1, 1, 5, 8)
print(A, pool(A), sep='\n')

tensor([[[[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11., 12., 13., 14., 15.],
          [16., 17., 18., 19., 20., 21., 22., 23.],
          [24., 25., 26., 27., 28., 29., 30., 31.],
          [32., 33., 34., 35., 36., 37., 38., 39.]]])
tensor([[[[10., 13.],
          [26., 29.]]]]
```

פלט:

בדוגמה זו שתי העמודות האחוריות והשורה האחורונה בפלט "אבדו" – הן לא השתתפו בתהליכי האיגום, שכן ממדיהם הפלט לא תואמים בדיקות את גודל המטריצה המשמשת לאיגום. כדי להימנע מאובדן זה ניתן להעביר לשכבה פרמטר הדורש להשתמש בכל המאפיינים בפלט, במחיר של מטריצה

איגום קטן יותר בקצוט. למשל בקטע הקוד שלහלן, המאפיין השלישי בשורה הראשונה בפלט מחושב כערך המרבי של מטריצה מסדר  $2 \times 2$ .

```
pool = nn.MaxPool2d(kernel_size=(2, 3), ceil_mode=True)
print(A, pool(A), sep='\n')

פלט:
tensor([[[[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11., 12., 13., 14., 15.],
          [16., 17., 18., 19., 20., 21., 22., 23.],
          [24., 25., 26., 27., 28., 29., 30., 31.],
          [32., 33., 34., 35., 36., 37., 38., 39.]]]])
tensor([[[[10., 13., 15.],
          [26., 29., 31.],
          [34., 37., 39.]]]])
```

בדומה לשכבה קונבולוציה, גם לשכבות איגום יש פרמטר `stride`, המאפשר לקבוע ברזולוציה גבוהה את החפיפה בין תנתית מטריצות אשר ימשו לצירוף מאפייני הפלט. ביריתת המחדל משתמש במטריצות זרות, כדי הווער הפרמטר `stride=kernel_size`. לבסוף, נציין כי עבור קלט רב ערכזים שכבת האיגום מחשבת פלט לכל ערוץ בנפרד.

לסיום, ראו בקטע הקוד המצורף כיצד אנו מגדירים בлок של שלוש שכבות, המשרשר את פעולה הקונבולוציה, האקטיבציה והאיגום. זהו הרץ שנשתמש בו לרוב בתחום רשת נוירונים, ביציפייה שהרשת תלמד בתהlik האימון את גרעין הקונבולוציה ואת פרמטר `bias`. בדוגמה זו אנו מגדירים ידנית את הגרעין ואת מזזה הקצוטות האופקיים והאנכיים שהשתמשנו בו בעבר.

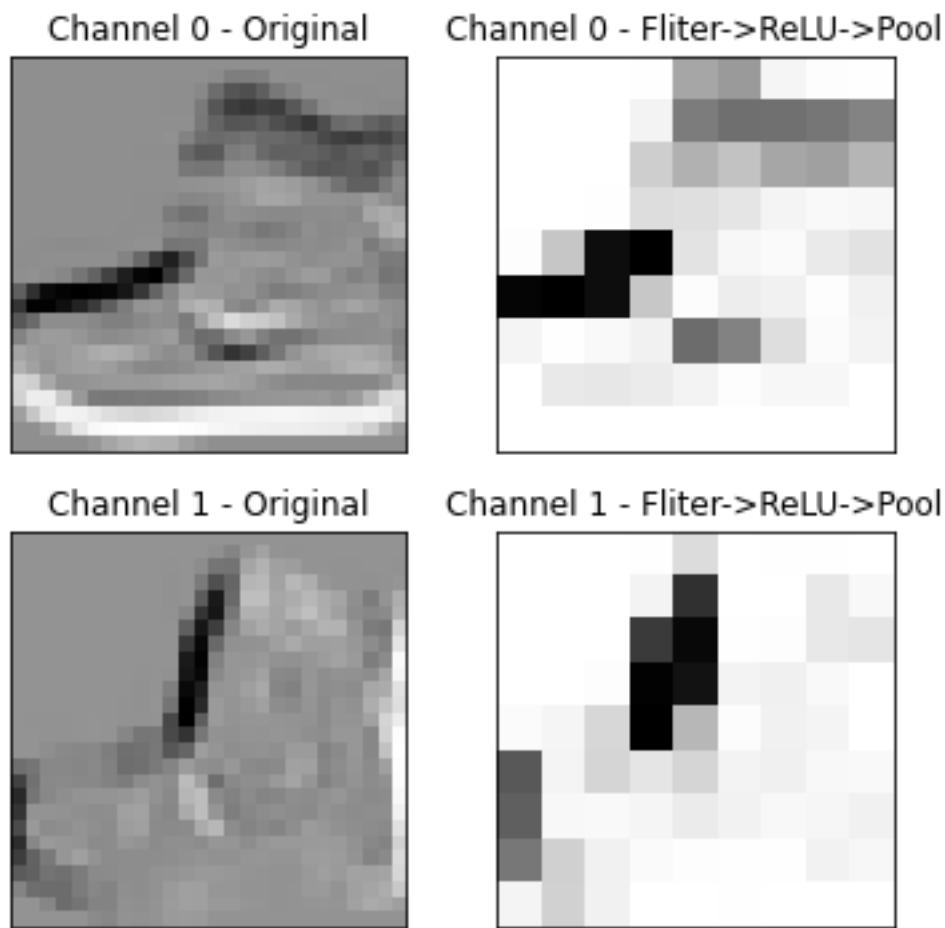
```
edge_detector2 = nn.Conv2d(in_channels=1, out_channels=2,
                          bias=False, kernel_size=(3, 3),
                          stride=(1,1))

with torch.no_grad():
    edge_detector2.weight[:]=torch.tensor(
        [[[ -1., -1., -1.],
          [ 0., 0., 0.],
          [ 1., 1., 1.]],

         [[ -1., 0., 1.],
          [ -1., 0., 1.],
          [ -1., 0., 1.]]]).reshape(2,1,3,3))

block = nn.Sequential(edge_detector2,
                      nn.ReLU(),
                      nn.MaxPool2d(kernel_size=(3, 3),
                                   ceil_mode=True))
```

פלט הבלוק (בגודל  $9 \times 9$  פיקסלים) עבר תמונה אחת מאייר להלן, לצד הפלט המקורי של מזזה הקצוטות לבחדו.



## שאלות לתרגול

1. עברו גרעין קונבולוציה בגודל  $q \times q$ , בכמה שורות ועמודות ריפוד יש להשתמש כדי לשמר על ממדיה המקוריים של תמונה הקלט לאחר הפעלת הקונבולוציה?
2. א. הגדרו שכבה קונבולוציה עם גרעין בגודל  $5 \times 5$  ללא פרמטר bias. בשכבה יהיה עורך קלט יחיד וערוך פלט יחיד.  
ב. בחרו תמונה קלט אקראיית מהאוסף Fashion-MNIST והעבירו אותה דרך השכבה.  
ג. רפדו את תמונה הקלט בשורות ובעמודות אפסים מלמעלה ומשמאלו בלבד, כך שהפלט יהיה מאותו ממד כמו הקלט. העבירו את התוצאה דרך שכבה הקונבולוציה והדפיסו או איררו את התוצאה.  
ד. האס תוכלו לשער מודיע נהוג למקם את האפסים הנוספים מסביב לתמונה המקורי, ולא רק בצד אחד שלה?
3. הציגו את הקונבולוציה של תמונה קלט  $X$  בגודל  $5 \times 5 \times 1$  (בעל ערך יחיד) עם גרעין בגודל  $2 \times 2$  כהעתקה לינארית.

**הנחיות:**

- ראשית מצאו את הממד של פלט הקונבולוציה.
- שטחו את תמונה הקלט והציגו אותה כווקטור  $\hat{X}$  בגודל  $1 \times HW$ .
- כתת מצאו מטריצה  $A$  כך ש-  $A\hat{X}$  היא תוצאה הקונבולוציה, אף היא לאחר שיטוח.
- מצא ראשית את ממד  $A$  בעזרת ההנחיות הקודומות.

- o רוב ערכיה של  $A$  יהיו אפסים.
4. העבירו לבניי של `Conv2d` בזמן ניתן פרמטר גודל פסיעה גדול מ-1 ואת הפרמטר `padding="same"`. הסבירו מדוע מתקבלת שגיאה.
5. צרו שכבת קונבולוציה בעזרת המחלקה `Conv2d` וגדירו את הגרעין שלה ידנית, כך שהיא תבצע פעולה זהה לשכבת איגום הנוצרת כך:

```
pool = nn.AvgPool2d(kernel_size=(2, 2))
```

הערה: שימושו לבשכבת איגום זו מחשבת את **הממוצע** של כל תת-מטריצה מסדר  $2 \times 2$ .

## רשתות קונבולוציה מודרניות

רשתות קונבולוציה נמצאות בשימוש החל משנות ה-90 של המאה ה-20: אחת הרשות הנפוצות המוקדמות היא LeNet-5, אשר אומנה לזהות ספרות בכתב יד מתונות בגודל 28X28 בעלות ערך צבע יחיד – תמונות בעלות ממדים זהים לאלו של אוסף הנתונים MNIST. רשת זו הראתה ביצועים שווים ערך לאלגוריתמי זיהוי תמונה אלטרנטיביים המשמשים בחילוץ מאפיינים מהונדס מראש למטרה הנדרשת והזנת מאפיינים אלו לאלגוריתם מיידית. זאת כאשר הקלט לרשת היה תמונות מקוריות, ללא כל עיבוד מקדים.

עם זאת, הגידול העצום בפופולריות של רשתות קונבולוציה, ושל למידה عمוקה בכלל, התרחש עקב הביצועים יוצאי הדופן של הרשת AlexNet בתחרות ILSVRC-2012: תחרויות זיהוי תמונה המבוססת על אוסף הנתונים הענק ImageNet. בפרק הנוח כי נסקור את הרכיבים אשר הובילו לניצחונה בתחרות, רכיבים אשר הפכו סטנדרטיים בתכנון המבנה ותהליכי האימון של רשתות נוירונייםعمוקות כיוום.

### אוסף הנתונים ImageNet

בשנת 2009 פורסם לראשונה מסד הנתונים ImageNet ובו מיליון תמונות שנאספו ממגוון חיפוש וממאגרי תמונות מקוונים, כגון Flickr. תמונות אלו סוגו בידי משתמשים אנושיים ליותר מ-20,000 קטגוריות שונות, חלקן חופפות (למשל כלב באופן כללי וככל מגזע מסוים), ולפיכך תמונה נתונה סוגה לעיתים בויזמנית לקטגוריות אחדות.

מהזק מסד זה בחרו מארגני התחרויות ILSVRC אלף קטגוריות זרות זו לזו, וסיפקו למשתתפי התחרותות כ-1,000 דגימות מכל קטgorיה, וב███ הכל כ-1.2 מיליון תמונות לאימון. תחרויות פופולריות קודמות השתמשו בסט נתונים של عشرות אלפי תמונות המסוגות לעשרות אחדות של קטגוריות, ולכן זמינותו של סט נתונים בסדר גודל שכזה הייתה קרקע פורייה להצלחתן של רשתות נוירוניים, אשר כזכור מצטיינות כאשר סט נתונים האימון גדול במיוחד.

גודל סט נתונים זה אינו מאפשר סיוג אנושי מדויק ומלא של כל התמונות: ניתן למשל שקיימת באוסף נתונים תמונה שהשicket לשתי קטגוריות בויזמנית, אך סוגה רק לאחת מהן. כדי להתמודד עם אידיעקים אלו, המدد להצלחה בתחרות נבחר להיות דיוק של 1 מזווית 5 (top-5 accuracy), שבו האלגוריתם הנמדד הציע חמיש קטגוריות אפשריות עבור כל תמונה מסט הבדיקה. אם הקטגוריה הנכונה היא אחת מהמש אלו, נזקפה לזכות האלגוריתם הצלחה בסיווג התמונה הנתונה. במדד זה הגיעו הצעה לשיעור סיוג נכון של 84.7%, בעוד האלגוריתם הבא בתור בתחרות זו, אשר השתמש בגישה קלאסית של חילוץ מאפיינים ומסוגים לינאריים, הגיע לשיעור סיוג של 73.8% בלבד. בשנת 2015 הגיעו הצלחה של האלגוריתם המנצח (גם הוא רשת קונבולוציה, ונלמד עליה בפרק הבא) כבר היה מעל ל-95%.

### עיבוד מקדים והעשרה אוסף הנתונים

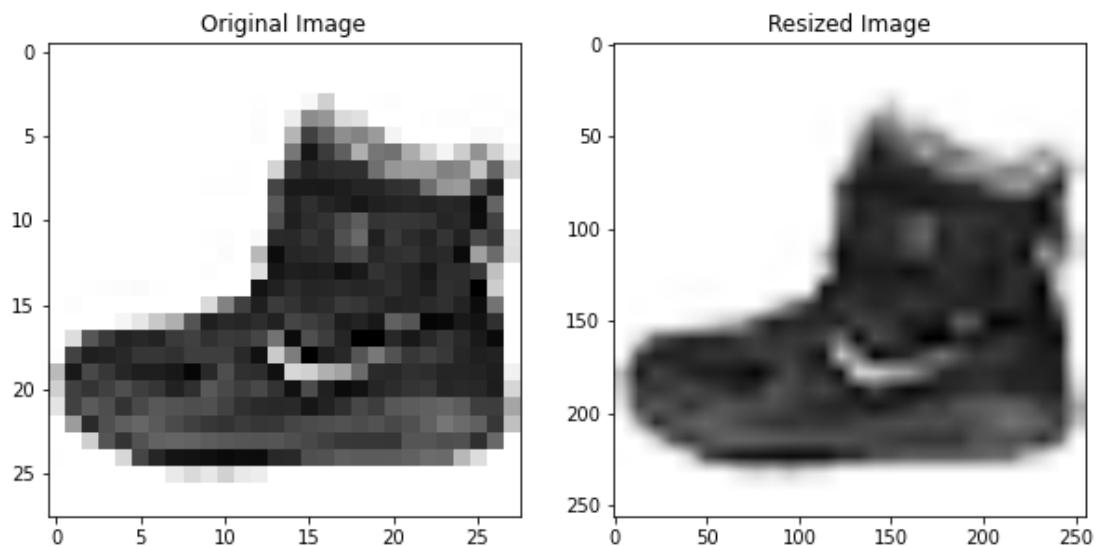
התמונות ב-ImageNet נאספו מרחבי האינטרנט ולבן לרוב הרזולוציה שלهن גבואה מדי כדי לשמש כקלט/APIilo עבור רשת קונבולוציה. לפיכך, לפני הזנתן לרשת הן הוקטו לגודל 256X256. בתחום תוצאות הימון תוצאה זו לא הוזנה ישירות אל הרשות אלא שימושה מוקור להעשרה אוסף הנתונים: בעזרת טרנספורמציות פשוטות, כגון שיקופים, סיבובים, שינוי צבע קלים וכו', ניתן ליצור מתמונה מקורית

אחתalfי תמונות נוספות מבלי לשנות את הקטגוריה שהיא שייכת אליה, ובכך להגדיל את סט הנתונים באופן מלאכותי ולהימנע מההתאמות יתר.

עבור נתונים ויזואליים קיימת PyTorch חבילת פונקציות נרחבת לעיבוד מקדים זה, ולפניהם שמשיך את הדיון ב-AlexNet, נדגים את השימוש שלו להלן.

```
import torchvision.transforms as T
resize = T.Resize(256)
new_img = resize(img)
```

לאחר הרצת קטע הקוד הניל, קיבל במשתנה `new_img` תמונה אשר הוגדלה או הוקטנה ל- $256 \times 256$ .  
ראו באירוע שלහן את השפעת הפונקציה על תמונה מהאוסף Fashion-MNIST.



לרוב נרצה להשתמש בכמה פונקציות של עיבוד מקדים, ולצורך זה ניתן לשרשון כדלהלן.

```
transforms = T.Compose([T.Resize(256),
                      T.RandomCrop(224),
                      T.RandomHorizontalFlip(0.5),
                      T.RandomRotation(30),
                      T.RandomPerspective(distortion_scale=0.3, p=0.5)])
new_img = transforms(img)
```

להלן מאורעות ותוצאות אחדות של הפעלת רצף טרנספורמציות זה, אשר מגדיל, חוטך, משקף סביב האמצע, ומסובב בדרכים שונות ובאופן את תמונה הקלט:

Original Image



בכל התמונות ניכר כי זהו מגף ושימוש בסט אימון המועשר בתמונות אלו יניב רשות אשר למדה לזהות מגף גם דרך טרנספורמציות אלו, ובבחירה מותאמת פחותה לסט הנתונים המקורי.

לבסוף, ניתן לשלב טרנספורמציות אלו ישירות בתהליך טעינת הנתונים לזיכרון, באמצעות הגדלתן בפרמטר `transform` של בניית אובייקט `Dataset`. בczורה זו, הטרנספורמציות יופעלו פעם אחת על כל תמונה בעת טעינת כל `minibatch`, וכך הרשת תיחשף לתמונות "חדשנות" בכל `epoch`.

נקודה עדינה שדורשת תשומת לב כאשר אנו מעשירים את סט הנתונים היא שאט סט הבדיקה יש להעביר אל הרשת דרך טרנספורמציות דומות, כדי להתאיםו לקלט שהרשות אומנה עליו. עם זאת, ברצוננו להימנע מהפעלת טרנספורמציות אקרואיות על סט הבדיקה, כדי שהחיזוי לא יהיה תלוי בהగרת המשתנים המקוריים. אם כן, סדרת הטרנספורמציות עבור סט הבדיקה המתאימה לשט אימון שהועשר כנ"ל מופיעה בקטע הקוד שלהלן, ואותה יש להעביר להגדרת אובייקט `Dataset` של סט הבדיקה. ראו כי הגזירה האקרואית הוחלפה בערך המומוצע שלה: גזירה במרכז התמונה.

```
test_transforms = T.Compose([T.ToTensor(),
                           T.Resize(256),
                           T.CenterCrop(224)])
```

## ארQUITטורת הרשת AlexNet

בקטע הקוד המצורף מופיעה ארQUITטורת הרשת AlexNet, כמחלקה של PyTorch. שימוש לב שזו הפעם הראשונה שבה אנו מגדירים רשת שלמה כמחלקה היורשת מ-`nn.Module`, אך אין זה שונה מהגדרת שכבה יחידה בczורה זו.

```

class AlexNet(nn.Module):
    def __init__(self, dropout=0.5):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, 11, stride=4, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, 5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, 1000))
    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x.flatten(start_dim=1))
        return x

```

מעיוון בארכיטקטורה שלמים כמה פרטיטים מעניינים. ראשית, נשים לב ששלט הרשות הוא שכבה לינארית – אלו הצוינים הגולמיים אשר יש להעביר לפונקציית softmax ולקבל את ההסתברויות השינויים לכל אחת מ-1,000 המחלקות. כזכור, בעורת הסתברויות אלו אנו מחשבים את פונקציית המחיר המתאימה למשימת סיווג, היא האנטרופיה הצלובה. ב-PyTorch ניתן להעביר ציוינים אלו לשירות לפונקציית המחיר המבצעת חישוב זה ביעילות. על כן, כדי לאמן רשות זו יש להגדיר את פונקציית המחיר כך :

```
CE_loss = nn.CrossEntropyLoss()
```

ולא בעורת הפונקציה `NLLoss`, כפי שעשינו בעבר, כאשר בסוף רשות הייתה שכבת `.nn.LogSoftmax`

שנייה, גרעין הקונבולוציה הגדל בשכבה הראשונה והשימוש בשכבות איגום מפחיתים את גודל הקלט ל $13 \times 13$  באמצע רשות, ואך ל $6 \times 6$  לאחר שכבת האיגום الأخيرة. עם זאת, מספר העරוצים גדל עד ערך המרבי 384. הרגע הניצב מאחוריו טרנספורמציות אלו הוא לאפשר לרשות ללמידה מסpter רב של

מחלצי מאפיינים המתייחסים לחלקים נרחבים ברשות, שהרי עקב הפחיתה הממד, פיקסל יחיד במאפיין מאמצע הרשות מחושב על בסיס מספר רב של פיקסלים בתמונה המקורית.

פרט שלישי מעניין הוא השימוש בשכבות dropout ובפונקציית האקטיבציה ReLU, שאמנם שגורות בשימוש המודרני, אך היו בתחום דרכן בעת פרסום המאמר, ואין ספק שהצלחתה של AlexNet תרמה לפופולריות של רכיבים אלו.

לבסוף, ניתן לראות שהחישוב ברשות מחולק לשניים: רשות קוונבולוציה המחלצת מאפיינים ומעבירה אותן למסוג סטנדרטי בעל קישוריות מלאה – זהו "ראש הסיווג" אשר ניתן להחלפה אם ברצונו לבצע משימה אחרת, כגון רגרסיה, על בסיס רשות זו.

שני פרטים חשובים אשר לא באים לידי ביטוי במימוש הנ"ל, אך היו חלק בלתי נפרד מהצלחתה הם השימוש בדנומול, בדומה לbatch normalization, על בסיס ממ"ד העורוץ במקומות ממ"ד ה-*batch*, וכן אימון הרשות על גבי מאיצ' גрафי. אפילו תוך כדי שימוש בשני מאיצים בויזמנית – נדרש זמן ריצה של כשבוע עבור אימון הרשות המקורית.

## שאלות לתרגול

1. א. הגדרו מחלקת הירשת Module. nn ובה ממומשת ארכיטקטורת הרשות LeNet-5 (מצאו איוור של ארכיטקטורה זו במאמר המקורי הנמצא באתר הקורס). ניתן להחליף רכיבים כגון פונקציית האקטיבציה  $\tanh$  בחלופות מודרניות, כגון ReLU.

ב. אמנו את הרשות לזהות את פריטי הלבוש מאוסף הנתונים Fashion-MNIST, עד אשר מתקבלת התאמת יתר ברווחה.

ג. לפני הזנת התמונות ברשות ביצעו טרנספורמציות שונות על תמונות הקלט למטרת העשרה נוספת הנתונים ואמנו את הרשות שוב. האם התקבלה התאמת יתר?

2. העבירו דרך הרשות AlexNet דוגמה אקראית של קלט במדדים הצפויים (תמונה צבע 224X224X3), תוך כדי הדפסת גודל הטנוור לאחר מעבר בכל שכבה.

**רמז:** השתמשו במתודה `( )` המחזירה גנרטור של תת-השכבות במודול.

3. מהו מספר הפרמטרים ברשות AlexNet?

**הנחיות:**

- מצאו את גודל גרעין הקוונבולוציה עבור כל שכבה, תוך כדי התחשבות במספר ערווצי הקלט והפלט. זכרו שלכל ערוץ פلت יש גם פרמטר `bias`.
- זכרו שבשכבה לינארית כל נוירון פلت מחובר לכל נוירון קלט, ולכל חיבור כזו קיים פרמטר.
- תוכלו לעשות זאת אוטומטית באמצעות מעבר על כל השכבות ברשות וצבירת הגודל של הפרמטרים שלחן.

4. על סמך התשובה לשאלת הקודמת, שערו מדוע בארכיטקטורת AlexNet השתמשו בשכבות dropout במסוג בלבד.

## רשותות שיוריות

בפרק זה נסקור את מבנה הרשת ResNet, אשר שימושה כבסיס לאלגוריתם המנץ' בתחרות ImageNet בשנת 2015, בשיעור דיווק top 5 accuracy של 96.43%. הסוד להצלחתה של ResNet הוא שימוש ברשותות עמוקות במיוחד: בעוד הרשותות הזרוכות בתחרות שנים קודמות היו בעלות 20–30 שכבות, מפתחי ResNet השתמשו ברשותות בעלות עד 150 שכבות.

באותן שנים היה ידוע בקהלת ציורי התמונה שרשותות קונבולוציה עמוקות מצליחות להקליל טוב יותר משיטות אחרות: המבנה ההיררכי של השכבות מאפשר לרשת לחץ מאפיינים מורכבים מתמונת המקור, מאפיינים השימושיים לשימוש הסיגוג העוקבת. עם זאת, רשותות עמוקות מצליחותאתגר קשה יותר לאמון, עקב הצורך לחשב את הגרדיאנט דרך מספר רב של שכבות. חישוב גראדיאנט זה עשוי להיות בעייתי, שכן התפשטות הגרדיאנט לאחרור דרך שכבה נתונה מבוצעת באמצעות **כפל** הגרדיאנט הקודם בגרדיאנט של שכבה זו. תוצאה הכפל של מספר רב של ערכים נוטה להתאפס (אם כופלים לרוב ערכים קטנים), או לשאוף לאינסוף (אם כופלים לרוב ערכים גדולים) – זהה התופעה של הגרדיאנט המתאפס או המתפוץ, אשר מובילת לאיתחכנסות אלגוריתם האופטימיזציה.

הוספת שכבות נורמליזציה כגון Batch-Norm ותחול נכוון של ערכי הפרמטר הפכו את תהליך האימון של רשותות קונבולוציה עמוקות, עם עשרות שכבות, לאפשרי, אך רשותות עמוקות יותר עדין הפגינו ביצועים פחותים, הן על סט הבדיקה כמוון והן על סט האימון עצמו, כך שלא היה מדובר בהתאם יתר. דבר זה עומד בסתייה לאינטואיציה, שכן על פניו, רשות עמוקה יכולה ללמוד לחץ מאפיינים זהים לאלו של רשות בעלת פחות שכבות ולהעביר אותם להלה מבלי לבצע כל חישוב נוספת בשכבות הנוספות, וכך להגיע לביצועים אלהים.

הבדיקות אלו הובילו את צוות החוקרים האחראי לפיתוח ResNet להסיק כי פתרון אפשרי לבעה יהיה לאפשר לשכבות ברשות ללמידה את פונקציית הזזהות בצורה קלה יותר. פתרון זה בא לידי ביטוי ברכיב פשוט אשר נター בפירוט מידי, המכונה **חיבור דילוג** (skip connection). חיבור כזה עוקף כמה שכבות, ונועד להפוך את פונקציית הזזהות ל"בירורת המחדל" של שכבות אלו. תוך כדי שימוש ברכיב זה, על השכבות המוקפות בחיבור דילוג ללמידה רק את **התווסף** לפונקציית הזזהות שיש לחץ, ואם אין כזו – איפוס הפרמטרים של השכבות יוביל להתפשטות פנימית של פונקציית הזזהות.

### בלוקים שיוריים

חיבור דילוג באים לידי ביטוי בבלוקים שיוריים (residual blocks), אשר הרשת ResNet מורכבת משרשור שלהם ברצף. תחילת נגדיר בלוק שיורי עבור רשות בעלת קישוריות מלאה, כדי לדון בתוכנותיו, ואחרי כן נעboro להגדרת הבלוק המקורי, עבור רשותות קונבולוציה.

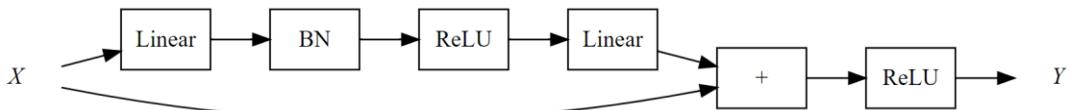


```

class ResBlockMLP(nn.Module):
    def __init__(self, in_features):
        super().__init__()
        self.relu = nn.ReLU()
        self.Z1 = nn.Linear(in_features, in_features)
        self.bn = nn.BatchNorm1d(in_features)
        self.Z2 = nn.Linear(in_features, in_features)
    def forward(self, X):
        Y1 = self.Z1(X)
        Y1 = self.bn(Y1)
        Y1 = self.relu(Y1)
        Y1 = self.Z2(Y1)
        Y2 = Y1 + X           #skip connection
        Y = self.relu(Y2)
        return Y

```

ראו בהגדרת השכבות הлиינאריות בעת אתחול הבלוק שמספר נוירוני הפלט (הפרמטר השני המועבר לבניית השכבה `nn.Linear`) שווה למספר נוירוני הקלט. בחירה זו אינה אקראיית, שכן בהתרפשותות פנימית דרך הבלוק יש לסקום את פלט שכבות אלו, איבר-איבר, עם הקלט המקורי, ועל כן הממדים חייבים להיות זהים. חישוב זה, נשמר במשתנה `Y2` הופך הבלוק זה לשינויי: לפני האקטיבציה האחרונה מוסיפים פלט הבלוק את הקלט המקורי ללא עיבוד. פעולה החישוב המבוצעת בבלוק מאורית להלן.



מכיוון שפעולות הסכום היא גזירה, ניתן לחשב את הגראדיינט לאחרור דרך הבלוק. לא רק שחישוב זה אפשרי, הוא אף יכול להועיל לתהליכי הלמידה: חיבור הדילוג מאפשר גראדיינט להתרפשת לאחרור ללא הפרעה במקרים שבהם הגראדיינט של שאר הבלוק מתאפס. נראה זאת דרך חישוב מפורש, ולצורך כך נסמן את טזוז הפלט על וביביו ב'  $(y_0, \dots, y_n)$ , את טזוז הקלט ב'  $(x_0, \dots, x_n) = X$  ובודומה את **חישובי הבניינים בבלוק ב'**.

$$Y_1 = (y_{1,0}, \dots, y_{1,n}),$$

$$Y_2 = (y_{2,0}, \dots, y_{2,n})$$

כעת, נניח שהגראדיינט פונקציית המחיר לפי פלט הבלוק חושב זה מכבר ויש לחשב אותו לאחרור. קלומר הערכים  $\frac{\partial C}{\partial y_k}$  ידועים לכל  $n, k = 0, \dots, n$  ויש לחשב את  $\frac{\partial C}{\partial x_m}$  לכל  $n, m = 0, \dots, n$ . נתחילה בשימוש ראשון בכל השרשרת,

$$\frac{\partial C}{\partial x_m} = \sum_{k=0}^n \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial x_m}$$

וחשב את  $\frac{\partial y_k}{\partial x_m}$  בתווו כך:

$$\frac{\partial y_k}{\partial x_m} = \frac{\partial \text{ReLU}(y_{2,k})}{\partial x_m} = \frac{\partial \text{ReLU}(y_{2,k})}{\partial y_{2,k}} \frac{\partial y_{2,k}}{\partial x_m} = 1\{y_{2,k} \geq 0\} \frac{\partial y_{2,k}}{\partial x_m}$$

ונזכיר ש-

$$1\{x \geq 0\} = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

היא הנגזרת של  $\text{ReLU}(x)$  אשר חישבנו בעבר. בשלב זה יש לשים לב ש-

$$y_{2,k} = y_{1,k} + x_k$$

לפי הגדרת הבלוק, ולכן

$$\frac{\partial y_{2,k}}{\partial x_m} = \frac{\partial(y_{1,k} + x_k)}{\partial x_m} = \frac{\partial y_{1,k}}{\partial x_m} + \frac{\partial x_k}{\partial x_m} = \begin{cases} \frac{\partial y_{1,k}}{\partial x_m} & k \neq m \\ \frac{\partial y_{1,k}}{\partial x_m} + 1 & k = m \end{cases}.$$

מכאן שגם אם הנגזרות דרך מסלול החישוב העליון בבלוק (ראו האיור לעיל) מתאפסות, קרי לכל  $k$  ו-  $m$

$$\frac{\partial y_{1,k}}{\partial x_m} = 0$$

עדין מתקיים

$$\frac{\partial y_{2,k}}{\partial x_m} = \begin{cases} 0 & k \neq m \\ 1 & k = m \end{cases}$$

ובהצבה בחזרה בנוסחאות הקודמות מתקבל

$$\frac{\partial C}{\partial x_m} = \sum_{k=0}^n \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial x_m} = \sum_{k=0}^n \frac{\partial C}{\partial y_k} \cdot \frac{\partial y_{2,k}}{\partial x_m} \cdot 1\{y_{2,k} \geq 0\} = \frac{\partial C}{\partial y_m} \cdot 1\{y_{2,m} \geq 0\}$$

הчисוב בתוצאה זו אינו ערך עצמאי, אלא שהוא אינו מתאפס: ללא חיבור הדילוג, התאפסות הגרדיינט דרך שאר הבלוק הייתה גוררת  $\frac{\partial C}{\partial x_m} = 0$ , ולפיכך לכל פרמטר השיעיך לשכבה הקומתית בבלוק זה בראשת הירינו מקבלים, שוב לפि כלל שרשרת,

$$\frac{\partial C}{\partial w} = \sum_{m=0}^n \frac{\partial C}{\partial x_m} \frac{\partial x_m}{\partial w} = 0$$

משמעות הדבר היא שהלמידה בעזרת אלגוריתם מבוסס גרדיינט עברו כל הפרמטרים הקודמים לכל הבלוק הייתה עוזרת.

## בלוקים שיוריים ברשותות קונבולוציה

בעוד הדיון שלעיל מציבע על הרלוונטיות הכללית של חיבור דילוג, ResNet עצמה היא עודנה רשות קונבולוציה, ולפיכך בבלוק השיורי המתאים לה יש שימוש בפעולות הקונבולוציה במקומות הארגנזה הילינארית. ראו זאת בקטע הקוד שלහן.

```

class ResBlockConv(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.relu = nn.ReLU()
        self.conv1 = nn.Conv2d(in_channels,in_channels,3,
                            padding="same",bias=False)
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.conv2 = nn.Conv2d(in_channels,in_channels,3,
                            padding="same",bias=False)
        self.bn2 = nn.BatchNorm2d(in_channels)
    def forward(self, X):
        Y1 = self.conv1(X)
        Y1 = self.bn1(Y1)
        Y1 = self.relu(Y1)
        Y1 = self.conv2(Y1)
        Y1 = self.bn2(Y1)
        Y2 = Y1+X
        Y = self.relu(Y2)
        return Y

```

מלבד החלפת השכבות הлиニアריות בפעולות קונבולוציה עם גרעין  $3 \times 3$  וריפורד השומר על ממך תמונה הפלט כממך תמונה הקלט (עומדה אחת ושורה אחת של אפסים בכל צד במקרה זה), נוספו שני שינויים:

קטנים הנוגעים לשכבות ה-*batch normalization*:

1. בבלוק זה יש שימוש בשכבות נורמליזציה מיוחדות עבור קונבולוציות, המנормלות **כל ערוץ** בנפרד (במקום כל פיקסל בשכבה BN קלאסית).
2. מכיוון ששכבות ה-*BatchNorm2d* לומדות פרמטר bias משלهن עבור כל ערוץ ומוסיפות אותו לכל הפיקסלים בערוץ לאחר הנרמול, אין צורך בפרמטר המבצע אותו חישוב בשכבה הקונבולוציה. לכן מועבר הפרמטר `bias=False` לבניית השכבה.

באמצעות שימוש בריפורד ושמירת מספר ערוצי הפלט כמספר ערוצי הפלט, גודל הטנזור  $1 \times 1$  במעבר קדים מה בלוק זה זהה לגודל הטנзор המקורי  $X$ , וכך ניתן לסקום אותם בחיבור הדילוג. עם זאת, יש לזכור שמטרתן של שכבות הקונבולוציה היא ללמידה מספר הולך וגדל של מחלצי מאפיינים (גרעיניים הקונבולוציה), תוך כדי הקטנת רזולוציית תמונה הפלט. לפיכך, בתכנון הרשות המלאה יש צורך בבלוק שירוי נוסף המאפשר גמישות זו. ראו את מימושו בקטע הקוד שלחלו.

```

class ResBlockDownSamp(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        out_channels = in_channels*2 # 

        self.relu      = nn.ReLU()
        self.conv1     = nn.Conv2d(in_channels,out_channels,3,
                               padding=1,stride=2,bias=False) #

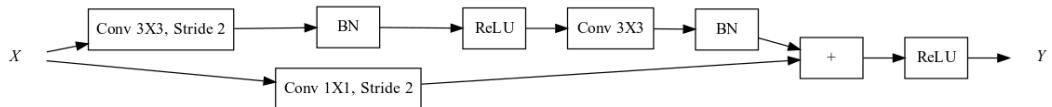
        self.bn1       = nn.BatchNorm2d(out_channels)
        self.conv2     = nn.Conv2d(out_channels,out_channels,3,
                               padding="same",bias=False)
        self.bn2       = nn.BatchNorm2d(out_channels)

        self.downsampX=nn.Conv2d(in_channels,out_channels,1 # 
                               ,stride=2,bias=False)

    def forward(self, X):
        Y1 = self.conv1(X)
        Y1 = self.bn1(Y1)
        Y1 = self.relu(Y1)
        Y1 = self.conv2(Y1)
        Y1 = self.bn2(Y1)
        Y2 = Y1 + self.downsampX(X) #
        Y  = self.relu(Y2)
        return Y

```

השינויים בין בLOC זה לבLOC הקודם מסוימים ב', והם נועד להקטין את ממ"ד האורך והרוחב פי חצי, בעוד מספר העורוצים גדל פי שניים, וזאת באמצעות שימוש בגודל פסיעה של 2. באIOR,



נתבונן בממדי הטעזורים של חישובי הביניים בLOC זה כדי להבין את השימוש בשכבות הקומבולוציה השלישי,  $X$  : עבור טנזור קלט  $X$  בגודל  $C \times H \times W \times N$ , לאחר הקומבולוציה הראשונה גודל המשתנה  $Y$  יהיה  $\frac{H}{2} \times \frac{W}{2} \times 2C \times N$  (בהנחה שממדי האורך והרוחב היו זוגיים), וכך גם לאחר הקומבולוציה השנייה, אשר לשמורת את הממדים. כתע ברכינו לסקום את תוצאת הביניים עם הקלט כדי להוסיף את חיבור הדילוג, אך ממדיהם שונים. לכן טנзор הקלט מוזן לתוך קומבולוציה עם גרעין  $1 \times 1$ , לצורך הכפלת מספר העורוצים וחציית ממ"ד האורך והרוחב. שאר החישוב דומה לזה המבוצע בLOC הקודם.

## רשתות שייריות (ResNets)

בעוד לעיל דנו ב-ResNet כאילו מדובר בראשת אחת, למעשה זהו מתכוון לתכנון כמה רשתות בעלות עומק משתנה. לנוחותנו ניתן לטעון אותן שיירות דרך דרך הספרייה PyTorch :

```
import torchvision.models as models
resnet18 = models.resnet18()
print(resnet18)
```

בהרצת קטע קוד זה יוגדר אובייקט רשת ResNet בעלת 18 שכבות ומבנה השכבות שלה יודפס. מהפלט ניתן לראות שהיא מורכבת מסדרה של בלוקים שיוררים רגילים המחווררים לבlokים שיוררים המגדילים את מספר הערווצים. ההבדל בין רשת זו לבין רשתות ResNet אחרות, כגון ResNet50, הוא מספר הבלוקים בסדרה בלבד.

משמעותו להתבונן בשכבות האחרונות בראשת, שאליהן מזון פلت מחלץ המאפיינים המבוסס על שכבות הkonvolוציה:

```
last_layers = list(resnet18.children()) [-2:]
print(*last_layers, sep="\n")
```

פלט:

```
AdaptiveAvgPool2d(output_size=(1, 1))
Linear(in_features=512, out_features=1000, bias=True)
```

בשכבה الأخيرة 1,000 נוירונים, כפוי מרשת אשר מיועדת לסווג את תמונות ImageNet ל-1,000 מחלקות שונות. פلت זה יזון לפונקציית softmax לייצירת הסטבריות השיכוכת למחלקות. השכבה הקודמת לה מחשבת ממוצע עבור כל אחד מ-512 הערווצים בפלט סדרת הקונבולוציות על פני ממד האורך והגובה. בעזרת שכבה זו אפשר להזין לרשת קלט אשר אינו בדוק הגודל הצפוי (תמונה  $224 \times 224$ ): בהתאם לגודל התמונה המזונה בראשת, אורך ורוחב הפלט של מחלץ המאפיינים השתנה, אך מספר הערווצים יישאר 512 והשכבה האדפטיבית תתאים את הממדים לאלו הדרושים לשכבת הסיווג. כך יהיה ניתן להשתמש בראשת המאוימת גם עבור תמונות ברזולוציה שונה מהמקורית.

## שאלות לתרגול

- הסבירו מדוע בירית המחלל של בלוק שיורי היא לחשב את פונקציית הזזהות, כאשר הוא לא תורם להורדת פונקציית המחיר. התייחסו לתהיליך אימון שבו משתמשים ברגולרייזציה של הפרמטרים.
- מנו את הפרמטרים ב"ראש הסיווג" המופיע בסוף ResNet18. השוו מספר זה למספר הפרמטרים במجموع של AlexNet

# יחידה 6: רשתות נשנות (RNN)

## שימוש מוקדים של נתונים טקסט

ביחידה זו נלמד על ארכיטקטורות רשת מיוחדות עבור נתונים סדרתיים, כגון מדידות של אוסף משתנים בפרק זמן קבועים (למשל מדידה יומית של טמפרטורה ולחות), או משפטים בשפה טבעית, שבהם בסדר הופעת המילים השפיעה על משמעות המשפט. כדוגמה חשוב על ההבדל בין שני המשפטים שלහן, המשמשים באותו המילים, רק בסדר שונה:

"טוב מאוד, לא רע"

"רע מאוד, לא טוב"

אם ברצוננו לאמן אלגוריתם לזהות את הרגש הבא לידי ביטוי בכל אחד מהמשפטים הנ"ל (חיובי או שלילי), הוא יהיה חייב למדוד בסדר המילים במשפט משפיע על הרגש המובע. רשת בעלת קשריות מלאה מקבלת כקלט את המשפט כולו ויכול למדוד את החשיבות של הסדר, אך יידרש לכך דוגמאות רבות ותהליך אימון סבוך. על כן, כמו ביחידת הלימוד הקודמת שבה תיכננו מראש רשות המיעודות לשימושות עיבוד תמונה, ביחידה זו נתקנן רשתות המביאות בחשבון באופן מובנה את ממד הסדר של הנתונים. רשתות אלו מכונות רשתות נשנות (Recurrent Neural Networks – RNN). הדוגמה אשר תלואה אותנו היא אוסף משפטיים בשפה האנגלית המבוקעים רגש שלילי או חיובי, ובפרק הנוכחי נלמד על עבודות ההכנה הדורשה לפני הזנתם כקלט לרשות ניירונים.

## אוסף הנתונים

מאגר המשפטים GLUE מורכב מתשעה אוספי נתונים שונים בתחום שימוש השפה הטבעית המשמשים לבדיקת ביצועיהם של אלגוריתמים לעיבוד שפה טבעית (Natural Language Processing – NLP) במגוון שימושות שונות. אחת המשימות היא זיהוי הרגש (שלילי או חיובי) המובע במשפט נתון, ואוסף הנתונים הנבחר לשימוש זה הוא SST2: אוסף של כ-70,000 משפטיים המתויגים לשתי מחלקות: 0 – רגש שלילי, 1 – רגש חיובי.

אוסף נתונים זה (ורבים אחרים), ניתן לטעון בעזרת הספרייה Datasets, אך ראשית יש להתקין אותה בסביבת העבודה, שכן היא לא מותקנת כברירת מחדל בהפצות הסטנדרטיות של פיתון. לשם כך, הריצו את הפקודה זו.

```
pip install datasets
```

לאחר ההתקנה ניתן לייבא את הספרייה, ובשורט קוד אחד לטעון את הנתונים הרצויים.

```
import datasets as ds
dataset = ds.load_dataset("glue", "sst2")
```

מתוך אוסף הנתונים ניקח את 67,000 המשפטים המשמשים לאימון, ונטען לזכור את המשפטים סיוגם כרשימות סטנדרטיות של פיתון.

```
sentence_list = dataset["train"]["sentence"]
labels_list = dataset["train"]["label"]
```

כעת נדפיס משפט אחד וסיווגו, כדי לוודא שטיענת הנתונים בוצעה כהלכה.

```
print(sentence_list[1], labels_list[1], sep="\n")
contains no wit , only labored gags
0
```

**פלט:**

## עיבוד מקדים

לפני הזנת הנתונים לאלגוריתם הלמידה, علينا לעורך בהם עיבוד מקדים. בעוד יכולנו להזין נתונים ויזואליים, כגון תמונות, ישירות לרשותות קובולוציה, אין זה המצב עבור רשותות נשנות: علينا להמיר את המשפטים לטזוריים מספריים בצורה שיטתיות, כאשר השלב הראשון בתהליך הוא טוקניזציה (tokenization) של המשפט, כלומר חלוקתו לרכיבים קטנים. הגישה הבסיסית ביותר למשימה זו היא חלוקת המשפט למילים לפי סימן הרוח המופיע ביניהם, כלהלן

```
sentence = sentence_list[1]
print(sentence, type(sentence))
print(sentence.split())
contains no wit , only labored gags <class 'str'>
['contains', 'no', 'wit', ',', 'only', 'labored', 'gags']
```

**פלט:**

שימוש בMETHOD split על משתנה מטיבוס str יניב רישימה של מושגים מסוימים מאותו טיפוס – אלו הם הטוקנים (tokens) של המשפט המקורי.

השלב הבא בתהליך העיבוד הוא יצירת אוצר המילים: איסוף כל הטוקנים באוסף הנתונים, ומיפוים למספרים סידוריים באמצעות חד-חד-ערכי. לצורך זה נשתמש בספרייה .torchtext

```

from torchtext.vocab import build_vocab_from_iterator
tokenizer = lambda x: x.split()
tokenized = list(map(tokenizer, sentence_list))

vocab = build_vocab_from_iterator(tokenized)
print(vocab(sentence.split()))
print(vocab("one two three".split()))

```

פלט:

```
[2923, 60, 329, 1, 88, 1992, 548]
[28, 128, 356]
```

ראו כיצד בעזרת הפונקציה `map` ביצענו טוקנייזציה לכל המשפטים בשורת קוד אחת. המשפטים הועברו לבנאי אוצר המילים המובנה, אשר החזיר אובייקט אוצר מילים המקודד טוקנים למספרים סידוריים. כך ניתן לראות למשל שהמספר 28 מייצג את המילה "one".

מכיוון שככל מספר שייך למילה אחת בלבד, ניתן לפרש קידוד נתון כזורה לטוקנים המתאימים. בדוגמה שלහלן אנחנו מפרשים את סדרת המספרים הראשוניים כזורה לטוקנים, אלו הם הטוקנים הנפוצים ביותר באוסף הנתונים.

```

print(vocab.get_itos()[0:10]) # integer to string

```

פלט:

```
['the', ',', 'a', 'and', 'of', '.', 'to', "'s", 'is', 'that']
```

אוצר המילים נבנה על בסיס סט נתונים האימון, אך לאחר אימון האלגוריתם נרצה להפעיל אותו על קלט כלשהו, לאו דוגמא כזה המורכב מאוצר המילים המקורי. במצב הנוכחי, אם ננסה לבצע טוקנייזציה והמרה למספרים סידוריים לשפט שמכיל מילים חדשות נקבל שגיאה.

```

vocab("hello world".split())

```

פלט:

```
RuneTimeError: Token hello not found and default index is not set
```

על כן נוסיף טוקן מיוחד עבור מילים מחוץ לאוצר המילים, ובאותה הזדמנות נמחק ממנו מילים נדירות, שכן אוצר מילים גדול מאוד יכיביד על תהליכי הלמידה.

```

vocab = build_vocab_from_iterator(tokenized,
                                  specials=["<UNK>"], min_freq=5)
vocab.set_default_index(0)
vocab("hello world".split())

```

פלט:

```
[0, 152]
```

כעת אנו רואים כי אף שהמילה "hello" לא נמצאת באוצר המילים, לא התקבלה שגיאה – מילה זו מופתча לטוקן של המילים הלא ידועות.

לבסוף, כל שנותר לעשות הוא לחלק את כל המשפטים באוסף הנתונים לטוקנים ולהמירם למספרים הסידוריים שלהם באוצר המילים.

```
stoi = lambda x: torch.tensor(vocab(x)) # string to integer
integer_tokens = list(map(stoi, tokenized))
print(integer_tokens[1])
```

פלט:

```
tensor([2924, 61, 330, 2, 89, 1993, 549])
```

במשתנה `integer_tokens` התקבלה רשימה פיתונית של טזוריים המכילים מספרים טבעיים: המספרים הסידוריים באוצר המילים של טוקני המשפטים המקוריים.

רשתות נוירוניים פועלות מטבען על מספרים  **ממשיים**  ולא טבעיים. על כן, השלב האחרון בעיבוד הנתונים יהיה להמיר את המספרים הסידוריים לייצוג ממשי: לכל מילה נתאים וקטור ממשי במרחב**רב-ממדי**, זהו שלב **שיכון המילה** (word embedding).

```
embedding_layer = nn.Embedding(len(vocab), 3)
integer         = stoi(["word"])
print(integer)
print(embedding_layer(integer))
```

פלט:

```
tensor([1234])
tensor([[ 0.4685, 1.1884, -0.3000]], grad_fn=<EmbeddingBackward0>)
```

בקוד זה הגדרנו שכבה המבצעת **שיכון מילים** למרחב תלמידי, המרנו את המילה "word" למספר הסידורי שלו במלון: 1234 וחשבנו את השיכון שלו. כאשר במשפט נתון **טופיע המילה** "word", טזoor תלת-ממדי זה הוא שיוון לרשות הנוירוניים במקומה.

שימוש לב Ci מערכת הגירה האוטומטית עוקבת אחרי השיכונים – אלו הם פרמטרים אשר יילמדו יחד עם אימונו הרשות, מותוך ציפייה שהאלגוריתם ילמד לייצג את המילים בצורה מועילה לניטוח הרגש של משפט נתון. בקטע הקוד שלහן נדגים כיצד השיכונים שמורים בשכבה: כמטריצה אשר השורה **ה-*k*-ית** שלה מכילה את השיכון של המילה **ה-*k*-ית** באוצר המילים.

```
W = embedding_layer.weight
print(W.size())
print(W[integer,:])
```

פלט:

```
torch.Size([11222, 3])
tensor([[ 0.4685, 1.1884, -0.3000]], grad_fn=<IndexBackward0>)
```

למידת שיכונים מוצלחים היא.Meshינה מأتגרת לא פחות מאשר מאימון רשות הנוירוניים עצמה: בדרך כלל השיכון יהיה מרחב מממד גבוה, ובאוצר המילים קיימות عشرות אלפי מילים. על כן, רק בשכבה השיכון יהיה יותר פרמטרים מכל שאר הרשות. כדי להתמודד עם אתגר זה, ניתן להשתמש בשיכונים מאומנים מראש (pretrained embeddings) על אוסף נתונים גדול. בקטע הקוד שלහן נדגים

כיצד לטען את הגרסה הקטנה ביותר של GloVe: שיכון מילים אשר אומנו על כל ערבי וקייפה בשנת 2014.

```
from torchtext.vocab import GloVe
glove_embedder = GloVe(name='6B', dim=50)
embedded = glove_embedder.get_vecs_by_tokens(["It's", "cool"])
print(embedded.size(), embedded.dtype, embedded.requires_grad)

平淡:
```

`torch.Size([2, 50]) torch.float32 False`

העברה משפט לאחר טוקנייזציה ל-GloVe תניב טזוזר המכיל את השיכונים של המילים במרחב 50-ממדי, שאוטם נזין לרשות הנוירונים במקום פלט שכבת השיכון שהגדנו בעבר. שימוש לב שמערכת autograd אינה עוקבת אחורי שיכון המילים, שכן ערכי השיכון כבר מאומנים ואין לנו רצון מהם השתנו בתהיליך אימונו הרשות אשר עתיד לבוא.

במבחן עמוק יותר לתוצאה נגלה כי השיכון של המילה "It's" הוא טזוזר האפס: זהו השיכון של מילים מחוץ לאוצר המילים של GloVe. בעת שימוש בשיכון מאומן מראש עלינו לתת את הדעת גם לאוצר המילים ולתהליך הטוקנייזציה שבו השתמשו בעת למידת השיכון, לאחר שנרצה להתאים אליהם את תהליך עיבוד הנתונים הגלמים שלו ככל האפשר.

נוסף על כך, יש להביא בחשבון גם את העובדה ששיכונים אלו לא אומנו על אוסף הנתונים שאיתו אנו עוסדים, וכן לא למטרת סיוג הוגש המובע המשפט, כך שייתכן שאין להם אידיאלים למטרה זו. זאת ועוד, אוסף הנתונים הגדל של שיכונים אלו אומנו לעליון לא מתאים ללמידה מפוקחת, שכן אין זה סביר לティיג את הרגש המובע בכל משפט בוויקיפדיה: עלות התיאוג באופן אנושי תהיה קירה להחריד. עקב לכך, שיכון GloVe נלמדו בפיקוח עצמי (SSL – Self Supervised Learning): כשלב הכנה לאלגוריתם האימון חושבו מספר המופעים המשותפים של כל זוג מילים באוסף המשפטים. מטריצת המופעים המשותפים בתורה שמשה לצירוף פונקציית המטרה של אלגוריתם הלמידה, כאשר התוצאה היא שכל הלמידה התרחשה על אוסף נתונים לא מתויג, מבליל להידרש להתרבות אונושית ביצירת אוסף הנתונים. עקב לכך, מילים המופיעות לרוב ייחדיו באוסף הנתונים הן בעלות שיכונים קרוביים זה לזה.

## שאלה לתרגול

המילים המשפט "It's cool" נפוצות בשימוש ועל כן לא סביר שהן לא נלמדו בתהיליך האימון של GloVe. נסו לבצע טוקנייזציה שונה למשפט, ידנית, ובדקו אם בדרך זו אפשר למצוא שיכון בעל ערך לכל טזוזן. **הערה:** טעינת השיכונים של GloVe כרוכה בהורדת נתונים בנפח של של כ-1GB.

## סיכום נתוני טקסט

בפרק זה נאמן רשות פשוטה מאוד להרגש הבא לידי ביטוי במשפט מסוון מסוון SST2. בפרקים הבאים נשלב בתוכה את רכיבי הארכיטקטורה המתקדמים, תוך כדי פיתוחם. בדומה לעיבית סיווג הנקודות השחורות והלבנות לשתי מחלקות המכברת לנו מיחידה 2, גם במקרה זה אנו מעוניינים לסווג לשתי מחלקות. על כן, בנוסף של הרשות יהיה עליו למקם ראש סיווג המניב פלט בעל שני איברים ומעביר אותו לפונקציית softmax. לפיכך על ראש הסיווג להיות מוגדר כדלהלן.

```
class ClassificationHead(nn.Module):
    def __init__(self, in_features):
        super().__init__()
        self.linear = nn.Linear(in_features, 2)
        self.logsoftmax = nn.LogSoftmax(dim=0)

    def forward(self, feature_extractor_output):
        class_scores = self.linear(feature_extractor_output)
        logprobs = self.logsoftmax(class_scores)
        return logprobs
```

ראו כי פלט ראש הסיווג הוא לוגריתם הסתברויות הסיווג, וזכור כי השתמש בו עבור פונקציית המחיר המתאימה לעיבית סיווג: האנטרופיה הצלבת.

כעת עליינו להגדיר את שאר הרשות, אשר יחלץ מאפיינים מהמשפט, ונרצה לעשות זאת בצורה הפשטota ביוטר: נזין ישירות את פלט שכבת השיכון אל ראש הסיווג. אם כן, מחלץ המאפיינים יורכב מהשיכון בלבד ובפרקים הבאים נהפוך אותו למתחכם יותר.

```
class FeatureExtractor(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.embedding = nn.Embedding(len(vocab), embed_dim)

    def forward(self, sentence_tokens):
        embedded = self.embedding(sentence_tokens)
        return embedded
```

בහינת משפט לאחר טוקנייזציה והמרת הטוקנים למספרים הסידוריים שלהם, מחלץ המאפיינים יחזיר לנו בעל שני ממדים: ממדו הראשון יציין את מיקום הטוקן במשפט, וממדיו השני את מרחב השיכון. ראו לדוגמה,

```

print(example_sentence)

preprocess = lambda x: torch.tensor(vocab(x.split()))
tokens    = preprocess(example_sentence)
print(tokens)

extractor = FeatureExtractor(2)
features  = extractor(tokens)
print(features.size(), sep="\n")

```

פלט:

```

contains no wit , only labored gags
tensor([2924,      61,     330,       2,      89, 1993,      549])
tensor([[ 0.9569, -0.6598],
        [ 0.9742, -1.0970],
        [-0.3451,  0.7615],
        [-0.8656,  1.8823],
        [ 0.6175,  0.2272],
        [ 0.9894, -0.8952],
        [ 1.0751, -1.2522]], grad_fn=<EmbeddingBackward0>)
torch.Size([7, 2])

```

פלט זה, המשתנה בגודלו בהתאם לאורך המשפט, עתיד להתגש עם הקלט הצפוי לראש הסיווג: בעת אתחול ראש הסיווג יש להגדיר את מספר נוירוני הקלט בשכבה הילינארית, בפרמטר `in_features` ובכך לקבוע את גודל הקלט. כדי להתגבר על קושי זה, נסכם את כל שיכוני הtokנים במשפט נתון לייצרת טנזור אשר ממדו קבוע, והוא כממד מרחב השיכון. השינוי הדורש במתודת `forward` של המחלץ מופיע בקטע הקוד שלහן.

```

def forward(self, sentence_tokens):
    embedded      = self.embedding(sentence_tokens)
    feature_extractor_output = embedded.sum(dim=0)
    return feature_extractor_output

```

כעת, כאשר מمد הפלט של מחלץ המאפיינים ידוע מראש, ניתן לחברו לראש הסיווג ולקבל את הרשות השלמה.

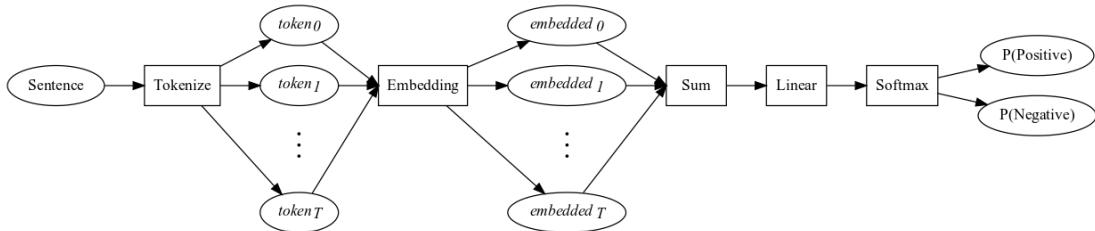
```

class EmbedSumClassify(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.extractor = FeatureExtractor(embed_dim)
        self.classifier = ClassificationHead(embed_dim)

    def forward(self, sentence_tokens):
        extracted_features = self.extractor(sentence_tokens)
        logprobs      = self.classifier(extracted_features)
        return logprobs

```

תהליך עיבוד הנתונים והזונת לרשף לצורך קבלת הסתבריות הסיווג מאויר להלן.



רשף זו אינה ערוכה לקבל batch של משפטים באורכים שונים, אך תוך כדי הזנת המשפטים זה אחר זה, נוכל לאמנה כרגע.

רשף בסיסית כנ"ל מסוגלת ללמד לסוג את המשפטים באופן סביר, אולם היא סובלת ממגבלה קריטית: פועלות הטעום מבטלת את משמעות סדר הופעת המילים במשפט. וראו בדוגמה שלחן כיצד שני משפטיים הנבדלים זה מזה רק בסדר הופעת המילים מסווגים באופן זהה.

```

example_sentences = ["very good , not bad",
                     "very bad , not good"]
with torch.no_grad():
    for sent in example_sentences:
        print(preprocess(sent))
        print(torch.exp(model(preprocess(sent))))
```

פלט:

```

tensor([77, 46, 2, 33, 74])
tensor([1.0000e+00, 4.7996e-07])
tensor([77, 74, 2, 33, 46])
tensor([1.0000e+00, 4.7996e-07])
```

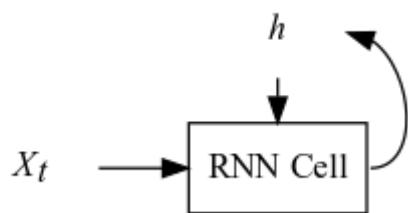
אין זה מפתיע, שכן הטוקנים של שני המשפטים זהים, ועל כן הקלט בראש הסיווג עברו שניהם זהה, ויישאר כך גם לאחר האימון.

## שאלות לתרגול

1. הגדרו אובייקט `EmbedSumClassify`, הינו את מיפוי סט האימון לרשות והעבירו את הפלט המתקבל לפונקציית המחיר של האנטרופיה הצלובת. אמנו את הרשות לסיווג הרגש המובע במשפטים, תוך כדי הזנת המשפטים זה אחר זה. בדקו את ביצועי הרשות המאומנת עבור ממדים שכון שונים.
2. החליפו את שכבת השיכוך ופעולות הסכום בשכבה `EmbeddingBag`. המבצעת שתי פעולות אלו זו אחרי זו ביעילות רבה יותר ואמנו את המודל מחדש. השוו את זמני הריצה. שימו לב שהקלט הצפוי של שכבה זו הוא batch של משפטיים, ולכן בעת הזנת משפט יחיד עליהם להוסיף ממך עצמו באמצעות `unsqueeze`, ולהוריד אותו לאחר מכן.
3. החליפו את שכבת השיכוך בשיכוני `GloVe` והשו את התוצאות לרשות עם שיכוניים נלמדים. השוו גם את זמני הריצה של תהליך האימון והסבירו את הפער שמצאתם.

## רשתות נשנות

בפרק זה נזכיר את רכיב הרשות אשר מאפשר לרשת הנוירונים להביא בחשבון את סדר הופעת המילים במשפט, זהו תא הרשות הנשנית (RNN cell) הידוע גם תא אלמן (Elman cell), על שם החוקר הראשון שפרט רשתות המבוססות עליו. בהמשך הדיון נניח כי משפט הקלט עבר טוקנייזציה ושיכון, וכן  $X_t$  הוא השיכון של הטוקן ה $t$ -י במשפט. הטוקנים מזומנים לפי הסדר אל תא הרשות הנשנית, שבו ישנו מצב חבוי (hidden state)  $h$  אשר מתעדכן עם הזנת כל טוקן. מלבד הטוקן, גם המצב החבוי הקודם משפיע על החישוב המתבצע בתא, וכך טוקנים המופיעים בתחילת המשפט משפיעים על המשך החישוב דרך השארת חומרם על המצב החבוי. להלן מצורף היוזן חזר זה באופן סכמטי,



ב יתר פירוט, בעת הזנת  $X_t$  לתא, המצב החבוי הבא מוחושב ונשמר לפי הנוסחה

$$h_{t+1} = \tanh(W_{input}X_t + b_{input} + W_{hidden}h_t + b_{hidden})$$

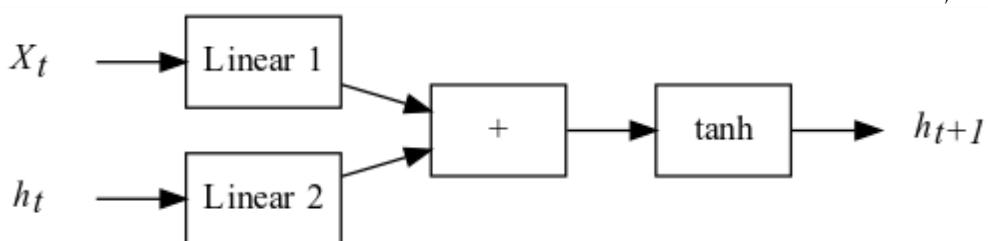
כאשר  $W_{input}, b_{input}, W_{hidden}, b_{hidden}$  הם הפרמטרים הנלמדים של התא, וכן

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

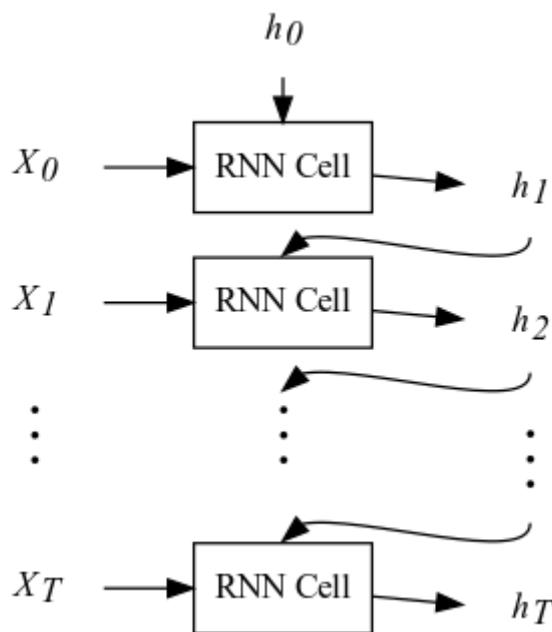
היא פונקציית הטגנס הhiperbolic: פונקציית אקטיבציה דמוית סיגמוואיד אשר מחזירה ערכים בטוחה  $(-1,1)$ . ניתן לפרש חישוב זה לשלבים:

1. הזנת טוקן הקלט (לאחר שיכון),  $X_t$ , לאגרגציה לינארית בעלת הפרמטרים  $W_{input}, b_{input}$ .
2. הזנת המצב החבוי הקודם לאגרגציה לינארית **אחרת** בעלת הפרמטרים  $W_{hidden}, b_{hidden}$ .
3. סכימת פלט שכבות האגרגציה.
4. הפעלת האקטיבציה על הסכום.
5. שמירת המצב החבוי החדש.

ובאיור,



בעת מעבר על דוגמה אחת, משפט באורך  $T$  טוקנים, החישוב הנ"ל מתבצע בטור  $T$  פעמים, ואם נרצה לאירר את זרימת המידע ברשות בפирוט, علينا להביאו זאת בחשבון. ראו באIOR שלහן כיצד אנו פורסים (unfold) את אIOR התא המקורי לאורך ממד הזמן, ומביאים לידי ביטוי מפורש את החישוב החזר על עצמו. ודאו כי אתם מבינים שמדובר בתא נשנה יחיד, בעל אותן פרמטרים.



כעת נಮש תא אלמן בקוד כמודול של PyTorch, ולאחר מכן נראה כיצד כלבבו ברשות נוירונים לחיזוי הרגש המובע במשפט.

```

class MyRNNCell(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_state = torch.zeros(hidden_dim)
        self.hidden_linear= nn.Linear(in_features = hidden_dim,
                                      out_features = hidden_dim)
        self.input_linear = nn.Linear(in_features = embed_dim,
                                      out_features = hidden_dim)
        self.activation = nn.Tanh()
    def forward(self, one_embedded_token):
        Z1 = self.input_linear(one_embedded_token)
        Z2 = self.hidden_linear(self.hidden_state)
        Y = Z1+Z2
        new_state = self.activation(Y)
        self.hidden_state = new_state
        #return
  
```

שימושו לב לכך שהמצב החבוי מוגדר מראש להיות בעל `hidden_dim` מאפיינים, שהוא מאותחל באפסים, וכן שבעת מעבר קדימה יחיד מזון לתא TOKEN משוכן בגודל `embed_dim`. בשונה משבوتות שראינו עד כה – לא מוחזר כל ערך בסוף החישוב, אלא המצב החבוי עצמו מעודכן ונשמר בתא. ראו גם כיצד מוגדרות השבותות הליניאריות, כך שפעולות הסכום מתבצע בין טנзорים מאותו ממד, וכן שמדובר המצב החבוי החדש יישאר זהה לממד המצב החבוי הקודם.

בבואהנו למש את הרשות הנשנית לשיווג המשפטים, נעשה שינויים אחדים בתוך הרשות שכתבו בפרק הקודם: נוותר על פעולות סכום השיכונים אשר ביטלה את חשיבות סדר הופעתם במשפט, ובמקומה נזין את שיכוני הטוקנים לפי הסדר בתוך התא. על המצב החבוי האחרון, לאחר הזנת כל הטוקנים, נחשב בתור פلت מחלק המאפיינים, ונזין אותו כרגע לשכבה ליניארית ולפונקציית `softmax`.

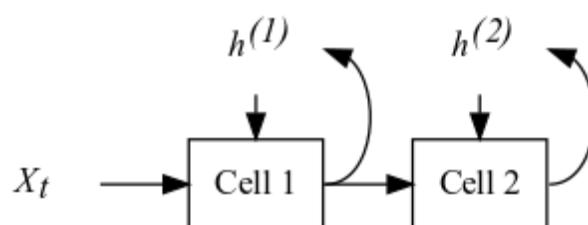
```
class RNNClassifier(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.embedding = nn.Embedding(len(vocab), embed_dim)
        self.rnn = MyRNNCell(embed_dim, hidden_dim)
        self.linear = nn.Linear(hidden_dim, 2)
        self.logsoftmax = nn.LogSoftmax(dim=0)

    def forward(self, sentence_tokens):
        self.rnn.hidden_state = torch.zeros(self.hidden_dim)
        for one_token in sentence_tokens:
            one_embedded_token = self.embedding(one_token)
            self.rnn(one_embedded_token)

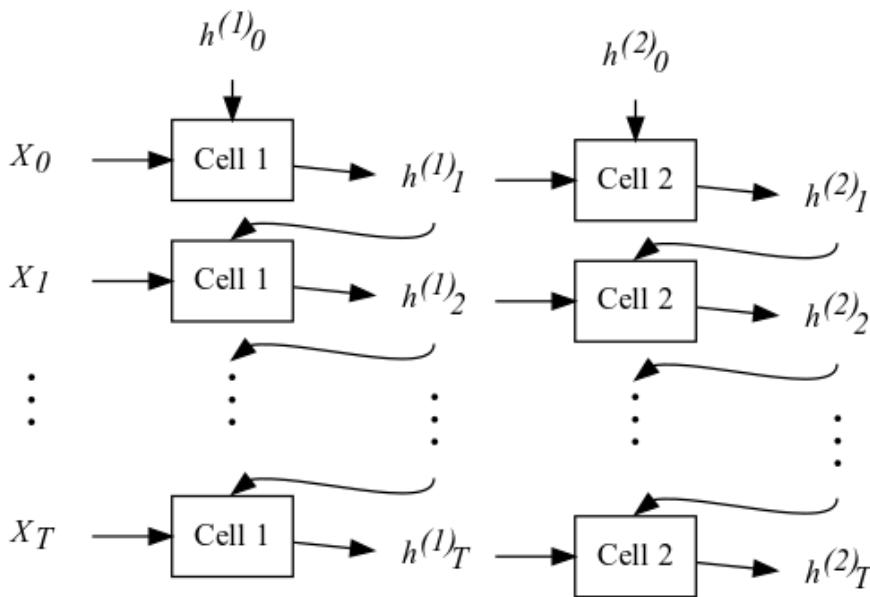
        feature_extractor_output = self.rnn.hidden_state
        class_scores = self.linear(feature_extractor_output)
        logprobs = self.logsoftmax(class_scores)
        return logprobs
```

ראו כיצד הוצרך בחישוב המצב החבוי החדש על סמך המצב החבוי הקודם מוביל לשימוש בלולה בעת מעבר קדימה ברשות: לפני חישוב המצב החבוי הסופי יש לחשב את כל הקודמים לו בסדרה.

כמו בשרותות בעלות קישוריות מלאה ושרותות קונבולוציה, ריבוי שכבות ויעבוד המידע לעומק השבותות מועיל גם עבור רשותות נשנות, אך במימוש הניל-Ano משתמשים רק בתא יחיד. כדי להרחיב את הרשות הנשנית לרשת עמוקה, علينا לחבר תא אלמן זה, כך שהמצב החבוי של התא הקודם יועבר כקלט לתא הבא, במקום הטוקן המשוכן. ובאיור סכמטי,



כמו כן, נאייר את החישוב המתבצע בהתאם לאחר פריסה לאורך ממד הזמן,



כעת פلت מחלץ המאפיינים יהיה פلت התא השני, לאחר הזנת כל הtokנים, המסומן ב-  $h_t^{(2)}$  באյור הנ"ל. נמשך רשות בעלת שתי שכבות נשנות באמצעות שינוי קוד המודול הקודם במעט, כאשר השינויים מסומנים ב- #.

```
class DeepRNNClassifier(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.embedding = nn.Embedding(len(vocab), embed_dim)
        self.rnn1      = MyRNNCell(embed_dim, hidden_dim)
        self.rnn2      = MyRNNCell(hidden_dim, hidden_dim)      #
        self.linear    = nn.Linear(hidden_dim, 2)
        self.logsoftmax = nn.LogSoftmax(dim=0)

    def forward(self, sentence_tokens):
        self.rnn1.hidden_state = torch.zeros(self.hidden_dim)
        self.rnn2.hidden_state = torch.zeros(self.hidden_dim)      #
        for one_token in sentence_tokens:
            one_embedded_token = self.embedding(one_token)
            self.rnn1(one_embedded_token)
            self.rnn2(self.rnn1.hidden_state)                      #

        feature_extractor_output = self.rnn2.hidden_state      #
        class_scores       = self.linear(feature_extractor_output)
        logprobs          = self.logsoftmax(class_scores)
        return logprobs
```

שימוש לב כך שמדובר הקלט של התא הנשנה השני כמדד המצב החבוי, שכן המצב החבוי הראשון משמש כקלט עבورو.

במימוש הרשתות שליל הדגשנו את בהירות החישוב המבוצע ואת זרימות המידע על פני ייעילות זמן הרצאה. על כן, לפני אימון הרשת, נחליף את השכבות הנשנות שלנו בעירימה (stack) של RNN מוגנות של PyTorch, המבצעת חישוב זהה, אך בדרך עיליה יותר.

```
class FasterDeepRNNClassifier(nn.Module):
    def __init__(self, embed_dim, hidden_dim, RNNlayers):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.embedding = nn.Embedding(len(vocab), embed_dim)
        self.rnn_stack = nn.RNN(embed_dim, # hidden_dim,
                               hidden_dim,
                               RNNlayers)
        self.linear = nn.Linear(hidden_dim, 2)
        self.logsoftmax = nn.LogSoftmax(dim=0)

    def forward(self, sentence_tokens):
        all_embeddings = self.embedding(sentence_tokens)
        all_embeddings = all_embeddings.unsqueeze(1)
        hidden_state_history, _ = self.rnn_stack(all_embeddings)

        feature_extractor_output = hidden_state_history[-1, 0, :]
        class_scores = self.linear(feature_extractor_output)
        logprobs = self.logsoftmax(class_scores)
        return logprobs
```

ראו כי אנו מעבירים את כל הטוקנים של המשפט בבת אחת לעירימת RNN, וברקע מתבצע החישוב בטור אך לא בלולאת פיטון. העירימה מחזירה כפלט את היסטוריית המצביעים החובויים של התא האחרון בערימה, התא השני במימוש הקודם שלו, וכן שולפים מטנזור זה את האיבר האחרון כפלט מחלץ המאפיינים.

השכבות המוגנות של PyTorch ערוכות לקבל minibatch של משפטים, כאשר בדרך כלל זהו הממד השני (המד הראשון נשאר ממך הזמן), ולכן לפני הזנת המשפט אל שכבת RNN המוגנת הוספנו לטנזור השיכונים ממד נוסף בעזרת המתוודה unsqueeze. הזנת הנתונים ב-batch דרושת עיבוד מקדים נוספים עקב אורך המשטנה של המשפטים, ולאណדו בו בקורס זה. עם זאת, נוכל עדין לאמן רשתות אלו כרגע באמצעות הזנת המשפטים זה אחר זה, במחair של תהליך אימון איטי יותר.

רשתות נשנות יש יכולת לחזות רגשות שונים עבור משפטים בעלי אותם טוקנים כאשר אלו מופיעים במשפטים בסדר שונה, וזאת בשונה מהרשת המבוססת על סכום שיכוני הטוקנים. עם זאת, כדי להשיג מטריה זו علينا לאמין בהצלחה. זהו אתגר לא פשוט ונדון בסיבות לכך בפרק הבא.

## שאלות לתרגול

1. אמנו RNN לחיזוי הרגש המובע במשפטים מוסף הנתונים SST2 תוך כדי שימוש בסט אימון קטן מאוד, עד שתתקבל בבירור התאמת יתר. הראו זאת באמצעות ציור גרף של שניות האימון ושגיאת הבדיקה.
2. א. מחקו את השורה הראשונה במתודה `forward()` של רשות מסוג `RNNClassifier`, שבה מאפסים את המצב החבוי לפני הזנת המשפט לרשות הנשנית. כתעת חזרו על השאלה הקודמת והשוו את התוצאות.  
ב. הסבירו את נחיצות האיפוס של המצב החבוי.
3. אמנו RNN عمוקה ותוך כדי האימון, עברו כל פרמטר ברשות, שמרו בנפרד את נורמת אינסוף של הגרדיאנט שלו לאחר כל איטרציה של לולאת האימון. ציירו גרף של ממוצע הנורמות כפונקציה של `epoch`.

## התפשטות לאחור דרך הזמן

בעת הפעלת אלגוריתם אופטימיזציה מבוסס גרדיאנט על רשות נשנית, לא ניכר על פניו שהתעורר בעיה: כל פעולות החשבון שננו משתמשים בהן לחישוב המצב הנוכחי האחרון חן גזירות, ולכן ניתן לחשב דרכן לאחור את גרדיאנט פונקציית השגיאה. ואכן כך פועלת מערכת autograd של PyTorch. עם זאת, המבנה המסוים של רשות נשנית, שבו אנו משתמשים באותו פרמטרים שוב ושוב בשלבים שונים בחישוב, מוביל לביעות של יציבות נומרית. נדון בהן接下來。

זכור כי נוסחת עדכון המצב החבוי של תא נשנה פשוט היא

$$, h_{t+1} = \tanh\left(W_{input}X_t + b_{input} + W_{hidden}h_t + b_{hidden}\right)$$

אך接下來 כתוב על כך שעבור משפט קלט באורך  $T$  טוקנים אנו מפעילים נוסחה זו  $T$  פעמים, עד ליצירת המצב החבוי הסופי, כפי שהופיע באירוע הרשות הפרוסה בפרק הקודם.

נתחיל את הדיוון במקורה הפשטוני שבו המצב החבוי הוא חד-ממדי:  $h_t$  הוא סקלר, ולפיכך  $W_{hidden}$ , מטריצת משקלים שכבה הילינארית החבוייה, מתנוונת לסקלר גם היא. נסמן אפוא  $w$ . עתה נרצה לחשב את נזורת פונקציית המחיר לפי פרמטר זה,  $\frac{\partial C}{\partial w}$ , לצורך ביצוע צעד העדכון של SGD.

בהתפשטות לפנים,  $w$  משפייע על החישוב המבוצע בראש בפעם האחרונה בעת חישוב המצב החבוי האחרון,  $h_T$ , וכן אם נניח כי הנזורת  $\frac{\partial C}{\partial h_T}$  כבר חושבה, נקבל מכלל השרשרת:

$$\cdot \frac{\partial C}{\partial w} = \frac{\partial C}{\partial h_T} \frac{\partial h_T}{\partial w}$$

בבואהו לחשב את

$$\frac{\partial h_T}{\partial w} = \frac{\partial}{\partial w} \tanh\left(W_{input}X_t + b_{input} + wh_{T-1} + b_{hidden}\right)$$

נפגש באתגר שנוצר עקב המבנה הרקורסיבי של הרשות: מצד אחד,  $w$  מופיע במפורשות בנוסחת המעבר, מצד אחר גם  $h_{T-1}$  עצמו כושב בעזרת אותם פרמטרים, וחישוב זה משפייע אף הוא על הגרדיאנט. על כן, יש:

1. לחשב את הנזורת מיידית  $\frac{\partial h_T}{\partial w}$  כאיילו  $h_{T-1}$  קבוע. נסמן אותה ב-  $\left[\frac{\partial h_T}{\partial w}\right]$  למיניות בלבד.
2. לחשב לאחור, לזמן הקודם, את הנזורת, ככלומר לחשב את  $\frac{\partial h_T}{\partial h_{T-1}}$ .
3. לחשב את  $\frac{\partial h_{T-1}}{\partial w}$ .
4. לחבר את התוצאות בעזרת כלל השרשרת.

נקבל לבסוף,

$$\cdot \frac{\partial h_T}{\partial w} = \left[ \frac{\partial h_T}{\partial w} \right] + \frac{\partial h_T}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial w} .$$

נוכל לחשב את  $\frac{\partial h_{T-1}}{\partial w}$  בנקל, ונעשה זאת אחר כך. לעומת זאת, בבואהו לחשב את  $\frac{\partial h_T}{\partial h_{T-1}}$  בתעורר אותה בעיה כמפורט, שכן גם  $\frac{\partial h_T}{\partial w}$  תלוי ב-  $w$  באותו שתי דרכיהם:  $w$  מופיע בנוסחת המעבר מ-  $h_{T-2}$  אל  $h_{T-1}$ , וגם  $h_{T-2}$  עצמו חשוב בעזורת  $w$ . על כן יש המשיך ולחשב לאחר מכן את השגיאה אל  $h_{T-2}$ , בדומה לחישוב שביצענו לעיל:

$$\cdot \frac{\partial h_{T-1}}{\partial w} = \left[ \frac{\partial h_{T-1}}{\partial w} \right] + \frac{\partial h_{T-1}}{\partial h_{T-2}} \frac{\partial h_{T-2}}{\partial w}$$

נציב ביטוי זה בנוסחה עבור  $\frac{\partial h_T}{\partial w}$  ונקבל:

$$\begin{aligned} \frac{\partial h_T}{\partial w} &= \left[ \frac{\partial h_T}{\partial w} \right] + \frac{\partial h_T}{\partial h_{T-1}} \left( \left[ \frac{\partial h_{T-1}}{\partial w} \right] + \frac{\partial h_{T-1}}{\partial h_{T-2}} \frac{\partial h_{T-2}}{\partial w} \right) = \\ &= \left[ \frac{\partial h_T}{\partial w} \right] + \frac{\partial h_T}{\partial h_{T-1}} \left[ \frac{\partial h_{T-1}}{\partial w} \right] + \frac{\partial h_T}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial h_{T-2}} \frac{\partial h_{T-2}}{\partial w}. \end{aligned}$$

השיקולים הנ"ל תקפים עבור כל אחד מהמצבים החבויים הקודמים, וכך יהיה צורך המשיך ולחשב את השגיאה לאחר דרך הזמן עד שנגיע ל-  $h_1$ , כך שלבסוף הנוסחה שלפיה יש לחשב את  $\frac{\partial h_T}{\partial w}$  היא :

$$\begin{aligned} \frac{\partial h_T}{\partial w} &= \left[ \frac{\partial h_T}{\partial w} \right] + \\ &+ \frac{\partial h_T}{\partial h_{T-1}} \left[ \frac{\partial h_{T-1}}{\partial w} \right] + \\ &+ \frac{\partial h_T}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial h_{T-2}} \left[ \frac{\partial h_{T-2}}{\partial w} \right] + \\ &\vdots \\ &+ \frac{\partial h_T}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial h_{T-2}} \dots \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial w} \end{aligned}$$

בנוסחה זו יש שורה עבור כל אחת מאיתרציות הולאה המופיעה בחישוב ההסתטוטות פנימה בראשת אלמנן. השורה הראשונה היא התרומה של האיתרציה האחורונה, הקולעת את הטוון האחרון במשפט, בעוד השורה האחורונה היא התרומה של האיתרציה הראשונה. הביעיות חישובית בנוסחה זו, מעבר לאורךה, נגלית לעין כאשר שמים לב ש-

$$\cdot \frac{\partial h_t}{\partial h_{t-1}} = (1 - h_t^2) \left[ \frac{\partial}{\partial h_{t-1}} (W_{input} X_t + b_{input} + w h_{t-1} + b_{hidden}) \right] = (1 - h_t^2) w$$

שכן  $(x)' = 1 - \tanh^2(x)$ . בהצבת תוצאה זו בנוסחה שלעיל נקבל

$$\begin{aligned}
 \frac{\partial h_T}{\partial w} &= \left[ \frac{\partial h_T}{\partial w} \right] + \\
 &+ w(1 - h_T^2) \left[ \frac{\partial h_{T-1}}{\partial w} \right] + \\
 &+ w^2(1 - h_T^2)(1 - h_{T-1}^2) \left[ \frac{\partial h_{T-2}}{\partial w} \right] + \\
 &\vdots \\
 &+ w^{T-1}(1 - h_T^2)(1 - h_{T-1}^2) \cdots (1 - h_2^2) \frac{\partial h_1}{\partial w}
 \end{aligned}$$

כעת ניכרים הגורמים הביעיטיים בנוסחה : הם מהצורה  $w^T$ . אם ערך הפרמטר הנוichi גדול מ-1 (בערכו המוחלט), גורמים אלו שואפים במהרה לאינסוף, כך שעבור משפט קלט ארוך דיו הנזרת  $\frac{\partial h_T}{\partial w}$  תהיה גדולה מאוד בערוכה המוחלט, דבר אשר יمنع מאלגוריתם האופטימיזציה להתכנס – זהה תופעת הגרדיאנט המתפוץ (exploding gradient).

עבור  $|w| > 1$  המצב אינו טוב בהרבה : גורמים אלו ישאפו במהרה לאפס, וכך איטם גם ההשפעה של הטוקנים המופיעים בתחילת המשפט על  $\frac{\partial h_T}{\partial w}$ . עקב כך ערך נזרת זו יחוسب בעיקר על סמך התמונות של הטוקנים האחרנים שהזנו לרשות, והרשות לא תוכל ללמידה תלויות ארכוכות טווח בתוך המשפט.

מכיוון שהשווים אלו נובעים מההתפשטות לאחר דרך המצביעים החבויים, הם רלוונטיים גם עבור שאר פרמטרי הרשות המופיעים לפני התא הנשנה, שכן עבורם יש לחשב לאחר את הנזרות דרך התא. קשיים אלו מתעוררים גם כאשר המצב החבוי הוא רב-מדוי, מכך שאט הביטויים  $w$  מחליפות חזקות של המטריצה  $W_{hidden}$ , אשר להן התנחות אסימפטוטית דומה.

ישנם כלים אחדים להתמודד עם הקשיים המובנה שיוצרת התפשטות הגרדיאנט דרך רשות נשנית. חלקם מנסים לפטור את הבעיה באמצעות שינוי אלגוריתם האופטימיזציה, למשל בעזרת הקטנת קצב הלמידה כאשר הגרדיאנט גדול מדי, בדומה לאלגוריתמים בעלי קצב למידה אדפטיבי שלMANDO עליהם ביחידה 3.

הכלים המוצלחים יותר פותרים את הבעיה באמצעות שינוי ארכיטקטורת הרשות : במקומות תא אלמן פשוט, הרשות תשתמש בהתאם לשנים מתקדמים, המכילים רכיבים דומים לחיבור הדילוג של רשותות שיריות. הנפוץ בהם הוא תא תא LSTM (Long Short-Term Memory), אשר מכיל גם רכיב בקרה זרימה המאפשר לתא ללמידה פתוח ומתי לסגור את חיבור הדילוג, וכן רכיב נלמד המאפשר לתא לאפס את המצב החבוי, למשל לאחר שהוא זיהה נתק סמנטי בין שני חלקים של המשפט.

רשותות מבוססות LSTM מומשו ב-PyTorch, והפלט שלו זהה במבנהו לזו של רשות RNN פשוטה. על כן נוכל להשתמש בהן בנקל, כלהלן :

```
class LSTMClassifier(FasterDeepRNNClassifier):
    def __init__(self, embed_dim, hidden_dim, RNNlayers):
        super().__init__(embed_dim, hidden_dim, RNNlayers)
        self.rnn_stack = nn.LSTM(embed_dim, hidden_dim, RNNlayers)
```

ראו כיצד רשת הסיווג החדשה יורשת האחרונה שהגדנו בפרק הקודם, שבה `rnn_stack` הוא אובייקט מסווג `RNN`.nn. השינוי היחיד שיש לעשות הוא להגדירו מחדש כ-`LSTM`.

לסיום דיון זה, נציין כי בעוד רשותות LSTM (או רשותות בעלות תאים נוספים דומים) הפגינו את הביצועים הטובים ביותר במשימות עיבוד שפה טבעית עד לשנת 2017, נכון לכתיבת שורות אלו בשנת 2022, רשותות מבוססות transformer הן המוצלחות ביותר במשימות אלו. ארכיטקטורת transformer, שבהណון ביחסית הלימוד הבהא, אינה נשנית ולכל מתחמekaת משני האתגרים המרכזיים של שימוש ב-RNN: איטיות החישוב בטור ובעיות הגרדיינט המתפוצץ/הנעלים.

## שאלות לתרגול

1. חשבו את  $\left[ \frac{\partial h_t}{\partial w} \right]$  במפורש, והראו כי ביטוי זה חסום.

2. א. הראו כי קיים חסם  $K$  כך שלכל  $t$  מתקיים

$$\cdot \left| \left(1 - h_t^2\right) \left(1 - h_{t-1}^2\right) \cdots \left(1 - h_1^2\right) \left[ \frac{\partial h_{t-1}}{\partial w} \right] \right| < K$$

ב. על סמך הסעיף הקודם, מצאו חסם  $L$  התלוי ב- $w$  דרך גורמים מהצורה  $w'$  בלבד.

3. חשבו את  $\frac{\partial h_t}{\partial w}$  בהנחה ש-  $h_0$  הוא וקטור האפס, כנהוג באתחול תא נשנה. הסבירו מדוע לא מופיע

ביטוי מהצורה  $\left[ \frac{\partial h_t}{\partial w} \right]$  לעיל.

# ICHIDAH 7: ארכיטקטורות מוקודד-מפרש

## מוקודד עצמי

עד כה רأינו בקורס ארכיטקטורות רשת שונות, והשתמשנו בהן כדי לحلץ מאפיינים מועילים למטרת הרשת, אשר הייתה פתרון לביעות גרסיה או סיוגו. למטרה זו, חיברנו בסוף כל רשת ראש סיוג המורכב משכבה לינארית ופונקציית softmax או ראש גרסיה המורכב משכבה לינארית בלבד. ביחידה זו נתמודד עם שימושות מגוונות, כגון הפקחת מדדים, דחיסת מידע ותרגום טקסט, ולפיכך ראש הרשת שנבחר יהיה מותחכם יותר ומותאם לשימה הנדרשת.

המבנה הבסיסי של רשת المسؤول לפטור את הבעיות הניל הוא בתצורת מוקודד-מפרש (-decoder). תפקידו של המוקודד הוא כמחלץ המאפיינים ברשותו המוכrhoת לנו: עליו לייצר ייצוג עיל של הנתונים עבור משימת המשך. המפרש מחליף את ראש הסיוג או את ראש גרסיה, ותפקידו לתרגם את פלט המוקודד, הקרי לעתים גם הייצוג החבוי (latent representation) לפלט הרצוי מהרשת. למשל, עבור רשת המיעדת לתרגומים משפט מאנגלית לעברית, המוקודד ייצור ייצוג חבוי של המשפט באנגלית, וקטור של מספרים ממשיים מממד קבוע, בעוד המפרש יתרגם ייצוג חבוי זה למשפט בעברית.

כדוגמה ראשונה לארכיטקטורה זו, נעסק במרקודד עצמי (autoencoder) אשר ישמש אותנו להקנתת ממד ולדחיסת מידע.פלט הרשת יהיה ככל האפשר קלט הרשת, אך ממד המרחב החבוי (latent space) יהיה קטן מממד הקלט, כך שהנתונים יידשו לעבור דרך "צוואר בקבוק". לאחר תהליך אימון מוצלח, עם פונקציית מחיר מתאימה, נצפה שהמרקודד ילמד לייצג את הנתונים בצורה תמציתית, תוך כדי אובדן מידע מינימלי, שכן המפרש בתורו נדרש לתרגם את הייצוג החבוי חזרה לקלט המקורי.

לצורך הדוגמת רעיון זה נחוור לאוסף הנתודות השחורות והלבנות, הדוגמה הראשונה שעשכנו בה ביחידה 2. כזכור, דגנונו 100 נקודות במרחב דו-ממדי, ושמרנו את הקואורדינטות בטנזור  $X$ . כתעת נבנה מוקודד עצמי פשוט ביותר אשר יקבל קלט טזoor זה.

```
encoder      = nn.Linear(2, 1)
decoder      = nn.Linear(1, 2)
autoencoder = nn.Sequential(encoder, decoder)
```

ראו כי המוקודד מורכב משכבה לינארית ייחידה, המתקבל כקלט את שתי הקואורדינטות של הדוגימות ב- $X$  ומחזירה פלט חד-ממדי. המפרש ישוחרר את הנתונים לממד המקורי בשכבה לינארית אחרת.

ברצוננו לאמנו רשות זו לבצע משימהה בכל רשת אחרת, בעזרת אלגוריתם מבוסס גרדיאנט. לשם כך אנו נדרשים להגדיר את פונקציית המחיר. מכיוון שנרצה שפלט המפרש יהיה קרוב ככל האפשר לקלט הרשת, ריבוע המרחק האוקלידי ביניהם הוא בבחירה טبيعית. אם כן, נסמן את קלט הרשת (נקודת

דגימה יחידה ב-  $(x_0, x_1)$  ואת הפלט המתאים ב-  $(y_0, y_1)$ , ונקבל שהתרומה למחיר הכלל של נקודה זו היא:

$$H(x_0, x_1) = \|(y_0, y_1) - (x_0, x_1)\|^2 = (y_0 - x_0)^2 + (y_1 - x_1)^2$$

כרגיל, המחיר הכללי יהיה ממוצע של ערך זה עבור כל הדגימות, ונמש אוטו בקורס בעזרת הפונקציה `MSELoss`. מכאן, המוכרת לנו מושתנות הרגרסיה. חישוב פונקציית המחיר בתוך לולאת האימון יתבצע כך:

```
reconstructed_X = autoencoder(X)
loss            = MSELoss(reconstructed_X, X)
```

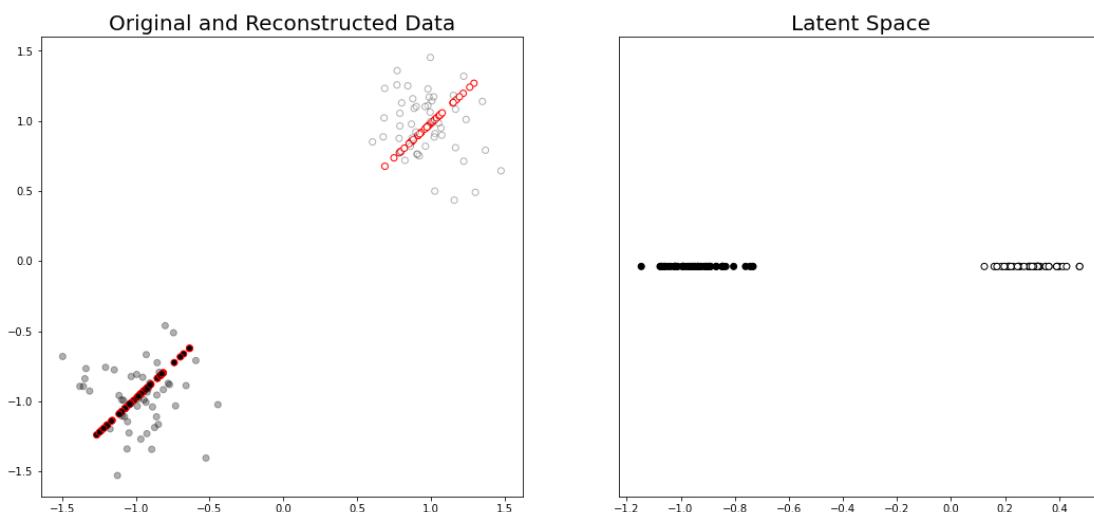
ראו כי הסיווג הנוכחי של הנקודות (שחורות או לבנות) אינו משתתף בתהליכי האימון, ותנו דעתכם לכך שתהליכי אימון המקודד העצמי מבוצע באופן בלתי מפוקח.

לאחר האימון נוכל לפצל את המקודד העצמי לחלקיו, להזין למקודד **בלבד** את הנתונים, ולקבל ייצוג חבוי של הנתונים במרחב חד-ממדי.

```
with torch.no_grad():
    encoded_X = encoder(X)
    encoded_X.size()
    torch.Size([100, 1])
```

פלט:

MOVED שמעבר לייצוג חבוי זה כרוכ בבדיקה מידע על הנתונים המקוריים, שכן פלט המקודד העצמי בהכרח יושב על קו ישר ייחיד, כפי שניכר מהאיור שלහן שבו המידע המשוחזר מסומן בנקודות בעלות שולטים אדומים.



עבור שימושות מיידת מסוימות אובדן המידע דוקא רצוי, למשל כאשר נרצה לאמן בהמשך מודול המבצע הכללה על סמך אוסף נתונים האימון. צוואר הבקבוק מלאץ את המודול להיפטר ממיען הספציפי לאוסף האימון (בין השאר), וה הכללה על סמך הייצוג החבוי לעתים פשוטה יותר. גישה זו שימושית במיוחד כאשר קיימים סט נתונים בלתי מפוקח גדול, ורק חלקו הקטן מתויג. על אוסף הנתונים כולו יוכל לאמן מודול עצמאי, ואחריו כן יוכל לאמן מודול סיווג פשוט על הנתונים המתוארים, כאשר הקלט למודול הסיווג יהיה הייצוג החבוי המתkeletal מהמודול.

## שאלות לתרגול

1. כתבו במפורש את החישוב המתבצע במודול העצמי הנ"ל, והוכחו שפלט הרשות כפונקציה של הקלט  $(x_0, x_1)$  הוא ישר.  
רמז: הציגו את  $y_1, y_0$  כפונקציות של  $x_1, x_0$  ומצאו הצגה של  $y_1$  כפונקציה של  $y_0$ . אפשר להניח שפרמטרים מסוימים אינם מתאפסים.
2. א. חקרו ראש סיווג לפט **המודול המאמן** ואמננו את ראש הסיווג בלבד להבדיל בין נקודות שחזורות לבנות על סמך הייצוג החבוי שלhn. השוו את המודול המתkeletal למודול הנירונו היחיד שאימנו ביחידה 2.  
ב. כתען חזרו על אימון שתי הרשותות מסעיף א, אך הפעם השתמשו בסט אימון המורכב רק מנקודה שחורה ייחידה ונקודה לבנה ייחידה. בדקו את ביצועיהם על שאר הנקודות.  
ג. באיזה מסעוג תעדיפו להשתמש עבור נתונים חדשים?

## שימושים נוספים של מקודדים עצמיים

עקרון הפעולה של המקודד העצמי שהכרנו בפרק הקודם תקף גם כאשר אוסף הנתונים המוzon אליו מורכב יותר מאשר נקודות במרחב דו-ממדי. פרק זה נאמן מקודדים עצמיים לדחיסת תמונות מאוסף הנתונים Fashion-MNIST ולשחזור תמונות מושחתות, כדוגמת לשניים מהשימושים האפשריים של ארכיטקטורה זו.

בשלב הראשון, אפשר למקודד ולמפרש ללמידה תלויות לא לינארית בעזרת תוספת של פונקציות אקטיבציה לאחר האגרגציה הלינארית. ראו זאת בקטע הקוד שלහלן, ושים לב לכך שבשלב ראשון אנו משטחים את טזוזר הקלט, שכן  $N$  תמונות מאוסף הנתונים מגע בטנזור בגודל  $1 \times 28 \times 28 \times N$ , בעוד השכבה הלינארית מצפה קלט במדים  $784 \times N$ .

```
class Encoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.linear = nn.Linear(784, latent_dim)
        self.relu = nn.ReLU()
    def forward(self, image):
        flattened = image.flatten(start_dim=1)
        compressed_image = self.relu(self.linear(flattened))
        return compressed_image
```

בוואנו לכתוב את המפרש, עליינו לתת את הדעת לצורת הפלט הרצואה שלנו. ברצונו לשחזר את התמונות המויזנות אל הרשות, אשר בזמן טיענתן עברו נרמול: ערך כל פיקסל הוא מספר ממשי בטוחה בין 0 ל-1. על כן נשתמש באקטיבציה הסיגומואיד המיצרת פלט בטוחה הדרוש. כמו כן, עליינו להפוך את פועלות השיטות, ולהחזיר פלט במדים זהים לאלו של קלט המקודד. ראו כיצד אילוצים אלו באים לידי ביטוי בקוד. שימו לב במיוחד לפרמטרים של מותודות ה-`reshape`.

```
class Decoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.linear = nn.Linear(latent_dim, 784)
        self.sigmoid = nn.Sigmoid()
    def forward(self, compressed_image):
        decoded = self.linear(compressed_image)
        decoded = self.sigmoid(decoded)
        reconstructed_image = decoded.reshape(-1, 1, 28, 28)
        return reconstructed_image
```

כעת נוכל לבחור ערך קטן עבור ממד המרחב החובי, לחבר בטור מקודד ומפרש, ולאמנם ייחדיו כמקודד עצמי. למשל, בשורה שלහלן אנו מגדירים מקודד עצמי בעל מרחב חובי עשר-ממדי (זכרו שמדד הקלט הוא  $28 \times 28$ ).

```
autoencoder = nn.Sequential(Encoder(10), Decoder(10))
```

לצורך אימון המקודד העצמי נעביר דרכו batch של תמונות ונשווה את התוצאה לתמונות המקוריות. פונקציית ההפסד תהיה שוב `MSELoss`. att, כאשר בהקשר הנוכחי היא קרויה "מחיר השחזור" (reconstruction loss), מסיבות ברורות. תוצאות אימון מוצלח מופיעות באיזור המוצרף, שבו מוצגים המקור ופלט המקודד העצמי של זוג תמונות מאוסף נתונים הבדיקה.



ראו כי השחזור אינם מושלים, שכן הנתונים עברו דרך צוואר בקבוק אשר גודלו כ-13% מגודל הנתונים המקוריים.

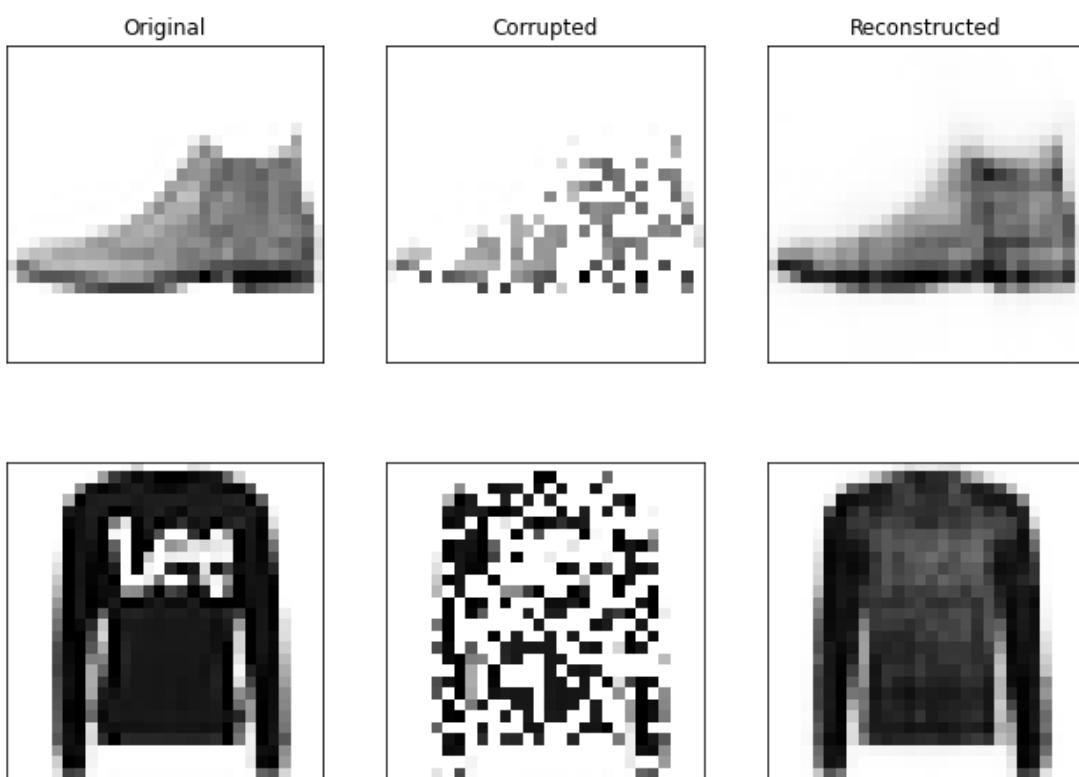
אם יהיה ברצוננו להשתמש במקודד לצורך משימות המשך, למשל עבור אימון מסוג על סמך הייצוג החבוי, נוכל לחלץ אותו בנקל, שכן הוא המודול הראשון באובייקט `autoencoder`

```
encoder = autoencoder[0]
```

כעת על בסיס הפלט של האובייקט `encoder`, נוכל לבנות רשת סיוג חדשה ולאמן בה את הפרמטרים של השכבות החדשות בלבד.

### מקודד עצמי לנקיי רעש

נוכל להשתמש באוטה ארכיטקטורת רשת פשוטה ששליל גם לצורך אימון מקודד עצמי המקבל תמונות אשר עברו השחטה ומשחרר אותן. רשת המשמשת למטריה זו נקראת באנגלית Denoising Auto-Encoder (DAE). לפניה שנדרן בפרטיו המקורי, רואו באיזור שלhalten שוחזרים של DAE מאומן. הקלט לרשת הוא התמונה האמצעית, אשר ח齊 מהפיקסלים שלה אופסו באקראי. הרשת מוחזירה את הפלט המקורי, מבלי שנחשפה מעולם לתמונה המקורית (משמאלו).



דרושים שניים ספורים לבניית המקודד העצמי ואימונו כדי לקבל תוצאה זו. ראשית, נרצה ליצור יתרות במרחב החבוי, כדי להתמודד עם אי-הוודאות המובנית בתנאים רועשים. לצורך כך נבחר מממד מרחב חבוי גדול כמעט מממד התנאים המקוריים.

```
latent_dim = int(784 * 1.2)
DAE = nn.Sequential(Encoder(latent_dim), Decoder(latent_dim))
```

שנית, علينا להשחית את התנאים למטרות האימון. לצורך כך, לאחר טעינת batch של תמונות למשתנה `original_imgs`, נעביר אותו דרך שכבת `dropout` אשר תאפס אקראית חצי מהפיקסלים בכל תמונה.

```
corruptor      = nn.Dropout()
corrupted_imgs = corruptor(original_imgs)
```

לבסוף, בתוך לולאת האימון נזכיר להזין לרשת את התמונות המושחתות, אך את הפלט נשווה לתמונות המקוריות. מחיר השזרור יתגמל מודלים אשר מלאים את החסר בתמונות המושחתות, כפי שאנו מעוניינים.

```
reconstructed = DAE(corrupted_imgs)
loss          = MSELoss(reconstructed, original_imgs)
```

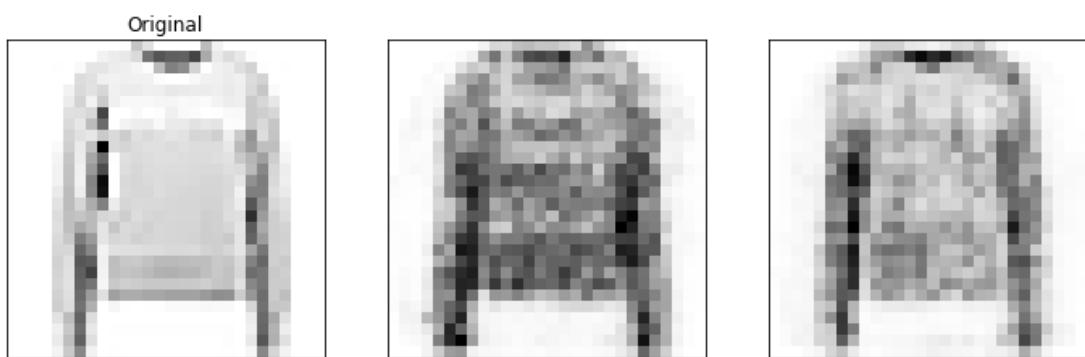
לאחר שינוי זה כל שנותר לעשות הוא להפעיל את אלגוריתם האימון.

### דגימת נתונים חדשים

שימוש נוסף של המוקודד העצמי הוא למטרת ייצור דוגמאות מלאכותיות חדשות הדומות לתמונות הקלט. לאחר האימון, אנו יכולים להזין ייצוג חבוי חדש לתוך **המפרש** המאומן, ולקבל תמונה משוחזרת המתאימה לייצוג חבוי זה, וזאת אף על פי שהוא אינו מייצג אף תמונה מקורית. ראו דוגמה לכך בקטע הקוד שלහן.

```
encoder = DAE[0]
decoder = DAE[1]
num_samples = 2
with torch.no_grad():
    encoded      = encoder(one_img)
    perturbation = torch.randn((num_samples,*encoded.size()))
    perturbed_latent = encoded*(1+perturbation)
    new_samples   = decoder(perturbed_latent)
```

ראו כיצד אנו מוסיפים לקידוד של תמונה מקורית רעש אקראי ומכוונים את התוצאה למפרש. שתי דוגמאות חדשות המתקבלות בדרך זו, לצד תמונה המקורי, מופיעות באיור שלහן.



בדוגמה זו אנו פוגשים לראשונה בראשו נוירונים **המייצרת** נתונים חדשים. רשותה המשמשות למטרות אלו הן **מודלים גנרטיביים**, וקיימות עבורה ארכיטקטורות רשות מוצלחות במיוחד אשר מפגינות יכולות יצירתיות יוצאות דופן לא רק במשימות הקלאסיות של מידת מכונה ומדעי הנתונים, אלא גם ביצירת אומנות ויזואלית. נושא זה הוא תחום מחקר עրوضה, אך בקורס הנוכחי לא עוסוק בו מעבר להיכרות הבסיסית שערכנו זה עתה.



בכל רשות נוירוניים, גם מקודדים עצמיים יכולים ללמידה ייצוגים חבויים מוצלחים באמצעות העמкат הרשת. בבעונו להוסיף שכבות למקודד לא תתעורר בעיה, ולמעשה יוכל אפילו להשתמש ברשת שאומנה למטרה אחרת, כגון ResNet, להוריד לה את ראש הסיווג ולהשתמש במקרה של המאפיינים שלא בתוור המקודד. עם זאת, עבור המפרש נדרש לבצע פעולה הפוכה: עיבוד נתונים הדרגתית בחזרה מהייצוג חבוי לתמונה. עם אתגר זה נתמודד בפרק הבא.

## שאלות לתרגול

1. אמנו כמה מקודדים עצמיים לנקיי רעש על אוסף הנתונים Fashion-MNIST הנבדלים זה מזה בגודל המרחב החבוי. השוו את התוצאות ומצאו את הערך האופטימלי. האם ממצאים DAE מוצלחים בעל ממד חבוי הקטן מממד הנתונים המקוריים? הסבירו תוצאה זו על סמך יתרונות המידע באוסף הנתונים עצמו.
2. חזרו על השאלה הקודמת עם אוסף הנתונים MNIST. מה תוכלו להגיד על ההבדל בין אוסף הערות:
  1. כדי לטעון את MNIST השתמשו בפונקציה `MNIST.torchvision.datasets`.
  2. שימו לב שאוסף הנתונים זהים מכל הבדיקות (גודל התמונה, מספר דגימות, מספר מחלקות וכו'), מלבד תוכן התמונות עצמן.

## קונבולוציה משוחפות ושימושה

שכבות הקונבולוציה שהשתמשנו בהן עד כה מקבלות כקלט תמונה בגודל  $W \times H$  בעלת מספר ערוצים כלשהו, ולרוב מוקטנות את ממך האורך והרוחב תוך כדי הגדלת מספר הערוצים, או לכל היותר שומרות על הגודל המקורי באמצעות ריפוד תמונה הקלט. היוצרו למשל במבנה הרשת ResNet, אשר שכבות הקונבולוציה שללה לעתים שומרות את ממך פלט זהה למדוד הקלט, ולאחרים חומרות את ממך האורך וממד הרוחב. שכבות אלו שימשו פתרון מצוין למטרת חילוץ מאפיינים אשר קשורים בהדרגה בין אזורים שונים בתמונה המקורית. עתה אנו רואים לתכנן מפרש המקבל מאפיינים אלו מהמודול ומשחזר את התמונה המקורית, ולשם כך עליינו לפעול בכיוון הפוך : הגדלת ממך האורך והרוחב, תוך כדי הקטנת מספר הערוצים. לשם כך נשתמש בפעולה חדשה : הקונבולוציה המשוחפת.

אנו מעוניינים בפעולה המתקבלת כקלט תמונה בגודל  $W \times H$  בעלת  $C$  ערוצים, והפלט שללה הוא תא תמונה גודלה יותר בממך האורך והרוחב, עם מספר ערוצים קטן יותר. לאור ההחלטה של שכבות קונבולוציה בעיבוד מידע ויזואלי, נרצה לשמר על תוכנותיה היסודיות גם בשכבות המבצעות את הפעולה ההופוכה, ובפרט :

- על השכבה להשתמש במספר קטן של פרמטרים בלבד, וכייה עלייה לבצע שימוש חוזר בהם, בדומה לגרעין של שכבת קונבולוציה.
- הפעולה תבצעה מוקנית : פיקסל בתמונה הפלט יחווש על סמך כמה פיקסלים הסמוכים זה לזה בתמונה הקלט בלבד.

בבחינה מעמיקה של תהליך האימון של רשתות קונבולוציה המוכרות לנו, נוכל להבין שambil לשימוש לבנו כבר משתמשים בפעולה העונה על כל הדרישות הנ"ל, והיא התפשטות הגדיאנט לאחרור דרך שכבת קונבולוציה. נדגים זאת באמצעות שכבת קונבולוציה המתקבלת כקלט תמונה בגודל  $W \times H$  בעלת ערוץ יחיד ומבצעת קונבולוציה שלה עם גרעין  $K$  בגודל  $q \times q$ . בדומה לדיוון שערכנו בעת הצגת שכבות הקונבולוציה, נסמן את תמונה הקלט ב-

$$, X = \begin{pmatrix} x_{1,1} & \cdots & x_{1,W} \\ \vdots & \ddots & \vdots \\ x_{H,1} & \cdots & x_{H,W} \end{pmatrix}$$

את הגרעין ב-

$$K = \begin{pmatrix} k_{1,1} & \cdots & k_{1,q} \\ \vdots & \ddots & \vdots \\ k_{p,1} & \cdots & k_{p,q} \end{pmatrix}$$

ואת תמונות הפלט ב-

$$. Y = \begin{pmatrix} y_{1,1} & \cdots & y_{1,W_y} \\ \vdots & \ddots & \vdots \\ y_{H_y,1} & \cdots & y_{H_y,W_y} \end{pmatrix}$$

זכרו שהמדד של תמונה הפלט תלוי בממדי תמונה המקור והגרעין, כך ש- $1 + p$ , וכן  $W_y = W - q + 1$ .

כעת נניח שבעת התפשטות לאחר חישבנו זה מכבר את גרדיאנט פונקציית המחיר לפי פלט השכבה. כלומר ערכי המטריצה

$$\frac{\partial C}{\partial Y} = \begin{pmatrix} \frac{\partial C}{\partial y_{1,1}} & \dots & \frac{\partial C}{\partial y_{1,W_y}} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial y_{H_y,1}} & \dots & \frac{\partial C}{\partial y_{H_y,W_y}} \end{pmatrix}$$

ידועים, וברצוננו לחשב את  $\frac{\partial C}{\partial X}$ .

התובנות העיקריות שיש להסיק ממחישוב העתיד לבוא כעת זו:

- פועלות התפשטות הגרדיאנט לאחר מקבלת קלט "תמונה" קטנה  $\frac{\partial C}{\partial Y}$ , וממדיה כממד  $Y$ .
- היא מחזירה כפלט "תמונה" גדולה  $\frac{\partial C}{\partial X}$ , מממד  $H \times W$ .
- כל זאת תוך כדי שימוש חוזר בגרעין הקונבולוציה ושמירה על אינטראקציות מקומיות.

ניגש אם כן למשימה. כזכור, איברי  $Y$  חושבו באמצעות חיתוך תתי-תמונה קטנות מהתמונה המקורי, חישוב המכפלה איבראיבר עם הגרעין וסכום התוצאה, ובנוסחה:

$$y_{r,s} = \sum_{i=1}^p \sum_{j=1}^q x_{r+i-1,s+j-1} k_{i,j}$$

נוסחה זו ניכר מיד כי:

$$\frac{\partial y_{r,s}}{\partial x_{r+i-1,s+j-1}} = k_{i,j} \quad i=1, \dots, p, \quad j=1, \dots, q$$

או במלילים: לכל פיקסל  $x_{n,m}$  בתמונה המקורי אשר השתתף בחישוב  $y_{r,s}$ , הנגזרת  $\frac{\partial y_{r,s}}{\partial x_{n,m}}$  היא איבר

הגרעין אשר כפל את  $x_{n,m}$  בעת חישוב פיקסל הפלט. מובן שלכל  $x_{n,m}$  אשר לא השתתף בחישוב  $y_{r,s}$

$$\frac{\partial y_{r,s}}{\partial x_{n,m}} = 0. \text{ נשלב תוצאה זו עם כלל השרשרת, שלפיו}$$

$$\frac{\partial C}{\partial x_{n,m}} = \sum_{r=1}^{H_y} \sum_{s=1}^{W_y} \frac{\partial C}{\partial y_{r,s}} \frac{\partial y_{r,s}}{\partial x_{n,m}}$$

ונקבל ש כדי לחשב את  $\frac{\partial C}{\partial x_{n,m}}$  יש לעבור על כל הפיקסלים  $y_{r,s}$  אשר בחישובם השתתף  $x_{n,m}$ , לכפול

את  $\frac{\partial C}{\partial y_{r,s}}$  באיבר הגרעין המתאים  $x_{n,m}$  בחישוב זה, ולסכום את כל התוצאות. בקטע הקוד של הלין

אנו ממשים רעיון זה. שימו לב כיצד אנו עוברים על כל פיקסל  $y_{r,s}$ , "נזכרים" בחישוב שביצעונו בהתקפות פנים ובחתams לכך מוסיפים איברים לגרדיאנט הנצבר במשתנה  $\Delta C_{dX}$ . ראו גם כי אנו עושים זאת בו בזמןית עבור כל הפיקסלים אשר השתתפו בחישוב  $y_{r,s}$ , וזאת בעזרה שידור הסקלר

$$\frac{\partial C}{\partial y_{r,s}}$$
 לממדיו של הגרעין הקונבולוציה.

```

def conv_backward(dCdY, K):
    p, q = K.size()
    Hy, Wy = dCdY.size()
    H = Hy+p-1
    W = Wy+q-1
    dCdX = torch.zeros((H, W))
    for r in range(Hy):
        for s in range(Wy):
            #forward: Y[r,s] = (X[r:r+p, s:s+q]*K).sum()
            #backward:
            dCdX[r:r+p, s:s+q] += dCdY[r, s]*K
    return dCdX

```

בעת נשכח את השימוש המקורי של פעולה זו ונעבור להשתמש בה כשכבה ברשת נירוניים. אם כן, נגידר את הגרעין  $K$  כפרמטר נלמד של השכבה ונפעילה על batch של תמונות בהתפשטות **לפניהם** ברשת. את התתפשטות לאחר מכן נשאיר למערכת הגזירה האוטומטית.

שכבות אלו מומשו ב-`PyTorch` במחלקה `ConvTranspose2d`. אשר מכילה את הפעולה הנילית `nn` לתמונות בעלות כמה ערוצי קלט ופלט, באופן אנלוגי לשכבות קוןבולוציה: לכל ערוץ פלט יהיה גרעין תלת-ממדי מסוים,  $C_{in} \times p \times q$ , כאשר  $C_{in}$  הוא מספר ערוצי הקלט. הפעולה המתואמת בקטע הקוד הקודם תבוצע בנפרד עבור כל ערוץ קלט, ולבסוף כל התוצאות ייסכמו ליצירת ערוץ פלט יחיד. בהנחה שתמונה הקלט היא מממד  $C_{in} \times H_y \times W_y$ , חישוב הקוןבולוציה המשוחלפת עבור אחד מערוצי הפלט יתבצע כך:

```

output = torch.zeros((H, W))
for c in range(C_in):
    output += conv_backward(input[c, ...], K[c, ...])

```

ריבוי ערוצי הפלט יתקבל באמצעות חזרה על פעולה זו באופן בלתי תלוי עבור כל אחד מהם.

בשכבות המובנות של `PyTorch` מומשה גם ההפונקציונליות של גודל הפסיטה והרייפוד. אך יש לשים לבן שהפרמטר `stride` מתייחס לפועלות הקוןבולוציה אשר ממנה נגורת הקוןבולוציה המשוחלפת, עם התוצאה שמדד האורך והרוחב של תמונה הפלט גדלים יחד עם ערך ה-`stride`, **בניגוד** להשפעת פרמטר זה על הקוןבולוציה הרגילה.

עתה, בעזרה שכבות אלו נוכל לחבר ישירות את פלט המקודד הקוןבולוציוני אל מפרש המורכב משכבות קוןבולוציה משוחלפת ולקבל רשת קוןבולוציונית מלאה ( – Fully Convolutional Network – FCN). רשותת מסוג זה שימושה במגוון מטרות ראייה ממוחשבות מתקדמות, בין השאר כאשר ברכנו לסוג את האובייקטים המופיעים בתמונה קלט, וכן על כן ליזות היכן הם נמצאים בתמונה.

## שאלה לתרגול

אםנו מקודד עצמי מנקה רעש על אוסף הנתונים Fashion-MNIST המשמש בשכבות קוןבולוציה, קוןבולוציה משוחלפת ואקטיבציות לא לינאריות בלבד.

## תרגום מכונה באמצעות רשות מקודד-מפרש נשית

בפרק זה נלמד כיצד לשלב ארכיטקטורת מקודד-מפרש רכיבים של רשותות נשות, ולאמן לבצע תרגום אוטומטי של קטע טקסט נתון. ראשיתណן באוסף הנתונים המתאים לשימוש זו : Tatoeba. זהו אוסף של זוגות משפטים בשפות שונות, שלהם משמעות זהה. המשפטים מתרגמים בהתקנות בידי משתמשי הפרויקט, כך שמצד אחד אוסף הנתונים ממשיך להתקדם כל הזמן, אך מצד אחר משתמשי האתר לרוב אין מתרגמים במקצועם וכן יש בו לא מעט שגיאות, קטנות וגדלות. לצרכינו ביחידה הנוכחית אוסף זה מספק, אך יש לזכור את מגבלותיו בבואהנו לאמן רשות למטרות החורגות מהדגמת יכולת.

אחד היתרונות של אוסף זה הוא מגוון השפות הגדול הזמין בו. השתמש שוב בספרייה Datasets לצורך טעינת הנתונים, כאשר לצורך הדוגמה שפת המקור שנבחר היא אנגלית, והיעד – צרפתית.

```
import datasets as ds
src="en"
tgt="fr"
dataset = ds.load_dataset("tatoeba", lang1=src, lang2=tgt)
```

הפרמטרים `lang1` ו-`lang2` שולטים בשפות של אוסף הנתונים המתתקבל, ובאמצעות החלפתם תוכלו להשתמש בקוד המופיע בהמשך פרק זה גם לצורך אימון רשות לתרגום מערבית לאנגלית, למשל.

נתבונן בדוגמה אחת מאוסף הנתונים המתתקבל.

```
dataset["train"]["translation"] [10]
```

**פלט:**

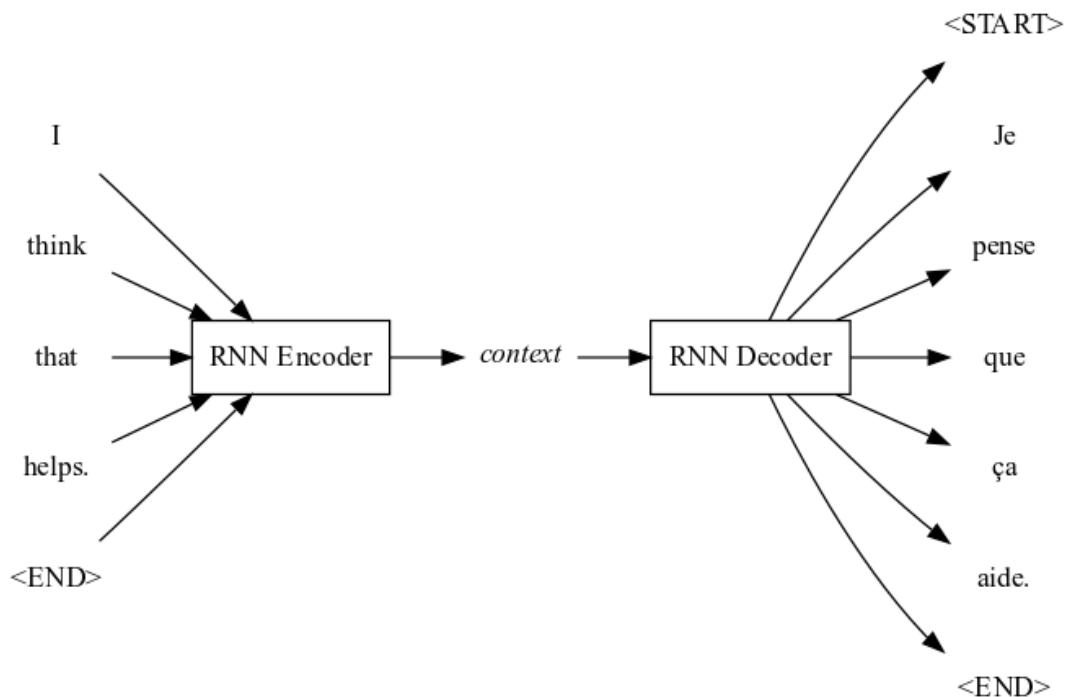
```
{'en': "Today is June 18th and it is Muiriel's birthday!",  
 'fr': "Aujourd'hui nous sommes le 18 juin et c'est  
 l'anniversaire de Muiriel !"}  
}
```

בריגיל במשמעותו עיבוד שפה טבעית, עליו לבצע עיבוד מקדים לאוסף הנתונים לפני הזרתו לרשות ניירניים. מכיוון שכעת אנו עוסקים בשתי שפות, יש ליצור אוצר מילים שונה עבור כל שפה: אחד מהם ייבנה מכל המשפטים בשפת המקור, אנגלית, והשני – מכל המשפטים בשפת היעד, צרפתית. כמו כן, כפי שנראה מייד, ארכיטקטורת הרשות הנבחרת דורשת לאותת למקודד על סוף משפט קלט, והמפרש מצפה לקבל אותן נספּן גם על תחילתו של המשפט. על כן נוסיף לכל אוצר מילים טוקנים מתאימים. דוגמה למשפטים לאחר טוקניזציה מופיעה להלן.

```
Source:  
['I', 'think', 'that', 'helps.', '<END>']  
tensor([ 2, 140, 43, 4795, 1])  
Target:  
['<START>', 'Je', 'pense', 'que', 'ça', 'aide.', '<END>']  
tensor([ 2, 4, 184, 39, 75, 644, 1])
```

הקלט לרשות יהיה הטזוזר המספרי הראשון, והטזוזר השני הוא הפלט הצפוי מהרשota, שבעזרתו נחשב את פונקציית המחיר.

באיור שלහלו מוצגת ארכיטקטורת הרשת ברמת ההפשטה הגבוהה ביותר : מקודד-מפרש אשר שני חלקיו מבוססים על רשותות נשנות.



המקודד בארכיטקטורה זו קיבל משפט מקור יחיד וימיר אותו לייצוג חבוי, הקורי בהקשר הנוכחי וקטור ה"הקשר" (context). בידינו כבר כל הרכיבים לכתיבת המקודד, זהו רכיב חילוץ המאפיינים ב-RNN אשר שימשה אותנו לסייע הרגש המובע במשפט. השוו את קטע הקוד שלහלו זהה של הרשת FasterDeepRNNClassifier אשר כתבנו ביחידה 6, וודאו כי אתם מוצאים בינהם דמיון.

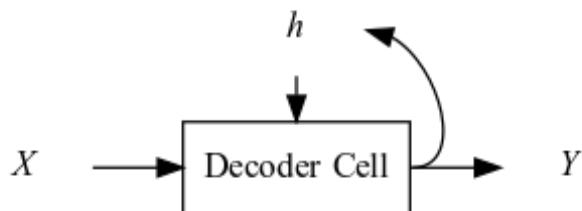
```
class Encoder(nn.Module):
    def __init__(self, embed_dim, hidden_dim, RNNlayers):
        super().__init__()
        self.src_embedding = nn.Embedding(len(src_vocab),
                                         embed_dim)
        self.rnn_stack     = nn.LSTM(embed_dim,
                                    hidden_dim,
                                    RNNlayers)

    def forward(self, src_tokens):
        all_embeddings      = self.src_embedding(src_tokens)
        all_embeddings      = all_embeddings.unsqueeze(1)
        hidden_state_history, _ = self.rnn_stack(all_embeddings)
        context            = hidden_state_history[-1, :, :]
        return context
```



זכרו שלט המודול LSTM. זה הוא סדרת המוצבים החבויים של התא הנשנה בראש עירימת ה-RNN, ולכן עברו וקטור ההקשר אנו בוחרים את האחרון שבה.

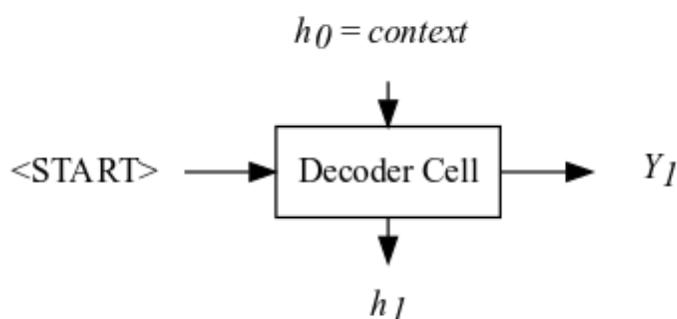
תכונן המפרש מציב לפניו אתגר: עליו **לייצר משפט** בשפת היעד כפלט. נמשח זאת בעזרתו תא נשנה אשר קיבל כקלט את וקטור ההקשר מהמוקוד ויבנה את משפט הפלט, טוקן אחר טוקן, מתחילה המשפט ועד סופו. על כן علينا להוציא לארכיטקטורת התא הנשנה הפשט אפשרות **לייצר פלט**, כמוואר להלן.



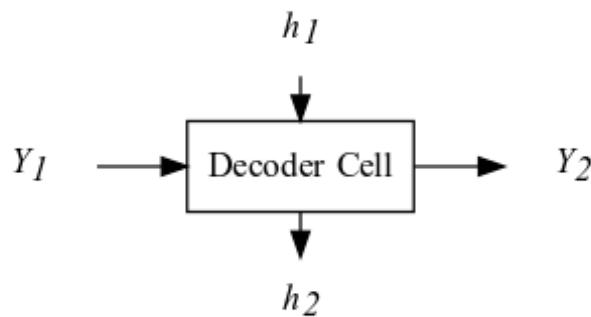
קלט התא  $X$  הוא טוקן (מושוכן) בשפת היעד צרפתית, וכך גם פלט התא  $Y$ .

לפני כתיבת תא זה כמודול של PyTorch, נתאר את הדרך שבה ישמש בו המפרש. ראשית, נדון במצב אימון שבו המפרש קיבל כקלט את וקטור ההקשר מהמוקוד, וכן את סדרת הטוקנים של המשפט המטרה, שננסמנם  $(Z_0, \dots, Z_T)$ . זהו בן הזוג של משפט המקור שהזון למוקוד. נשים לב לכך שלפי העיבוד המקדמים אשר את תוצאותיו ראיינו לעיל,  $Z_0$  הוא תמייד הטוקן המייצג את תחילת המשפט, ו-  $Z_T$  מייצג את סוף המשפט. פלט המפרש יהיה גם הוא סדרת טוקנים, שננסמנה  $(Y_0, \dots, Y_T)$ .

בתחלת החישוב הרקורסיבי בתא הנשנה נאתחל את המצב החבוי **בערכו של וקטור ההקשר** ונגידיר ידנית את  $Y_0$  להיות טוקן תחילת המשפט. כעת נזין טוקן זה בתור הקלט המסומן ב- $X$  לעיל (לאחר שיכוון כמובן), וכך נסמן למפרש שלו לשתף להתחיל לתרגם את משפט המקור על סמך וקטור ההקשר. ראו את הצעד הראשון במפרש באIOR המצורף.



כעת, לאחר קבלת הפלט  $Y_1$ , נזין אותו חוזה לתא, הפעם בתפקיד הקלט, ונקבל את  $Y_2$ .



נחזיר על פעולה זו עד לקבלת הטוקן  $Y_T$ , אשר כדי לייצרו נידרש לעדכן את המצב החבוי  $1-T$  פעמים, וליצור את כל הטוקנים הקודמים לו. ראו מימוש חישוב זה בקוד שלහן.

```

class TrainingDecoder(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.tgt_embedding = nn.Embedding(len(tgt_vocab),
                                         embed_dim)
        self.RNNcell = DecoderRNNCell(embed_dim,
                                      hidden_dim)

    def forward(self, context, tgt_tokens):
        self.RNNcell.hidden_state = context
        translated_tokens = [START_Token]
        for idx in range(len(tgt_tokens)-1):
            previous_token = translated_tokens[idx]
            embedded_token = self.tgt_embedding(previous_token)
            predicted_token = self.RNNcell(embedded_token)
            translated_tokens.append(predicted_token.detach())
        return translated_tokens

```

שיםו לב שאנו משתמשים באובייקט `DecoderRNNCell` בתוך בנאי המפרש אלום נגידיר מחלוקת זו רק בהמשך הפרק. לעת עתה ניתן להבין מהו הפלט הדרוש ממנו לפי השימוש בו במתודה `forward` והאירועים שלעיל.

יתכן שבאחד משלבי הבניינים הטוקן החוויתי יהיה טוקן סיום המשפט, שהרי גם הוא חלק מאוצר המילים של שפת היעד. במקרה זה נפרש זאת באמצעות הרשות על כך שהיא סימנה את עובדת התרגום, ועל כן נוסיף במתודה `forward` אפשרות לסיים את החוויתי מוקדם מהצפוי, בעזרת שורות הקוד הלאה.

```

if predicted_token == END_Token:
    break

```

לבסוף, נזכיר שעל הרשות לעבור אימון, ולמטרה זו יש לבחור פונקציית מחיר מתאימה. מובן שנרצה לתגמל במחיר נמוך מודל אשר חוזה נכונה טוקן  $Y$  הזזה לטוקן המתאים ממשפט המטרה  $Z$ , אך מעבר לכך, נרצה לתגמל במחיר נמוך עוד יותר מודל העושה זאת בביטחון רב. באופן דומה, נרצה לקנות



מודל אשר כלל לא היה קרוב לחזות את  $Z$  במחיר גבוה יותר מאשר מודל אשר כמעט עשה זאת, אך בסוף "התבלבל".

שיקולים אלו מוביילים אותנו לחשב על בעיית ייצור הטוקן  $Y$  כבעיית סיוג קלאסית: התא הנשנה מקבל כקלט את המצב החבוי והטוקן הקודם, ועליו לייצר  **לכל טוקן באמצעות המילים של שפט היעד את הסתברות  $P(Y_t = token)$** . זהו למעשה פלט הפונקציה softmax המוכרת לנו. לבסוף, נבחר בתור הטוקן הבא במשפט המתורגם את זה בעל הסתברות הסיוג הגבוהה ביותר, ה-`argmax`. היתרונות הגדול בנקודת מבט זו הוא שפונקציית המבחן המביטה את כל רצונותינו ידועה כבר, הלא היא האנתרופופיה הצלבת.

כעת אנו מוכנים לפרט את התהליך החישובי המבוצע בתא הנשנה, המורכב משני שלבים:

1. חישוב המצב החבוי החדש. שלב זה יבוצע בעבר, לפי הנוסחה

$$h_{t+1} = \tanh(W_{input}X + b_{input} + W_{hidden}h_t + b_{hidden})$$

כאשר  $X$  הוא הטוקן המתתקבל כקלט, ו-  $h_t$  הוא המצב החבוי הקודם.

2. חישוב הסתברויות הסיוג  $P(Y = token)$  באמצעות הזנת המצב החבוי החדש לשכבה לינארית ומיד אחריה – פונקציית softmax. אם כן, פלט התא יהיה וקטור בעל ערך לכל טוקן באמצעות המילים, המוחושב כך:  $\text{softmax}(W_{out}h_{t+1} + b_{out})$

הפרמטרים של התא הנשנה הם משקלים השכבות הלינאריות המופיעות לעיל וערך ה-`bias` שלו.

מימוש התא מופיע בקטע הקוד שלחן.

```
class DecoderRNNCell(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_state = torch.zeros(hidden_dim)
        self.RNNcell = nn.RNNCell(embed_dim, hidden_dim)
        self.output_linear = nn.Linear(in_features=hidden_dim,
                                       out_features=len(tgt_vocab))
        self.logsoftmax = nn.LogSoftmax(dim=0)

    def forward(self, one_embedded_token):
        new_state = self.RNNcell(one_embedded_token,
                               self.hidden_state)
        tgt_token_scores = self.output_linear(new_state)
        tgt_token_logprobs = self.logsoftmax(tgt_token_scores)

        self.hidden_state = new_state
        return tgt_token_logprobs
```

ראו כי השתמשנו בתא הנשנה המובנה ב-`PyTorch`, אשר מבצע את החישוב הרשום בסעיף 1 לעיל.

עבדתנו עוד לא הסטיימה, שכן עליינו לחשב את המחיר עבור כל דגימה לאחר הזנתה בראשת. יהיה לנו לעשות זאת **תוך כדי** ההתחפשות פנימי בראשת, שכן כך לא נדרש לשמור את ההיסטוריה של וקטורי הסטבוריות הסיווג של כל הtokנים ( $Y_0, \dots, Y_T$ ) עד לסוף תרגום המשפט. בעודנו זוכרים שהtokנים הנכונים נתונים הצלבת היא מינוס לוגריתם ההסתברות שהמפרש מנסה לתקן הנכוון, וכן שהtokנים הנכונים נתונים במשתנה `TrainingDecoder`, המשתנה `tgt_tokens`, נוסיף גם את חישוב זה למトודה `forward` של `TrainingDecoder`. המופיעה בשלמותה להלן. תוספת חישוב המחיר מסומנת ב-# בקוד.

```
def forward(self, context, tgt_tokens):
    self.RNNcell.hidden_state = context
    translated_tokens = [START_Token]
    sentence_loss = 0
    for idx in range(len(tgt_tokens)-1):
        previous_token = translated_tokens[idx]
        embedded_token = self.tgt_embedding(previous_token)
        logprobs = self.RNNcell(embedded_token)
        predicted_token = logprobs.argmax()
        translated_tokens.append(predicted_token.detach())

        correct_token = tgt_tokens[idx+1] #  

        token_loss = -logprobs[correct_token] #  

        sentence_loss += token_loss #  
  

        if predicted_token == END_Token:
            break
    return translated_tokens, sentence_loss
```

ראו כי כת המפרש מחזיר למשתמש גם את המחיר הכלול של המשפט, שהוא סכום המחיר של כל אחד מהtokנים במשפט המתורגם. מחיר זה יוניש מודל אשר חוצה tokנים שגויים וכן מודל אשר מסיים עבדתו מוקדם מהצפי, שכן אז הוא יחזיא את tokן סיום המשפט במקום הלא הנכוון, וישם על כך מחיר באנטרופיה הצלבת של tokן זה.

כל שנוטר לעשות כתה הוא לחבר את פלט המקודד אל המפרש, ולאמנם ייחדיו. לאחר האימונו, נרצה להשתמש ברשות תרגום משפט המקורי שעובדו אין לנו תרגום מוקן בסט האימונו. על כן יש להוסיף למפרש מצב חיזוי, שבו הוא מקבל קלט רק את וקטור ההקשר ומיציר tokנים עד לייצור tokן סוף המשפט (או מספרtokנים קבוע מראש). מובן שבמצב חיזוי המפרש לא יחזיר את מחיר המשפט, שכן לא ניתן לחשבו כלל ללא משפט המטרה.

## שאלות לתרגול

1. השלימו את שלב עיבוד הנתונים המקדים עבור שתי השפות, כך שלבסוף יתקבלו אוצר המילים של כל שפה וטנзорים המכילים את המספר הסידורי של tokנים באוצר המילים, לכל משפט בכל שפה.
2. מוחים ממדדי הפרמטרים  $W_{out}, b_{out}$  המשמשים לחישוב tokן הפלט בתא הנשנה החדש שהגדכנו לעיל?
3. כתבו בקוד את מצב החיזוי של המפרש, כמתואר בפסקה الأخيرة בפרק זה.
4. א. חקרו את המקודד והמפרש למודול PyTorch יחיד.

- ב. בחרו זוג שפות המוכר לכם וטענו את האוסף המתאים להן לזכרו המחשב.
- ג. אמנו רשות תרגום המכונה על batch קטן של זוגות משפטיים.
- ד. הדפיסו כמה משפטיים מתרגומים לצד התרגומים המקוריים, כפי שהוא מופיע באוסף הנתונים.
5. ניתן לאמן את המפרש באמצעות הזנת הטוקן הנכון לתא הנשנה בכל איטרציה במקום הטוקן האחרון שהפרש חזה, וזאת כדי לייצר את הטוקן הבא במשפט המתרגמים. מובן שאפשרות זו, הנראית באנגלית teacher forcing,teacher forcing רק במצב אליו, שכן רק אז בידינו הטוקנים המקוריים.
- א. הוסיפו את השימוש ב-teacher forcing, ואמנו שוב את הרשות על batch קטן.
- ב. ציירו על גראף אחד את המחיר המוצע כפונקציה של epoch האימון עבור שני המודלים שאליהם: בשאלת הקודמת.

## שכבת תשומת לב ו שימושה במפרש הנשנה

אחד האטגרים המרכזיים בשימוש ברשותות נוירוניות נשנות הוא למידת קשרים ארכוי טווח. דנו בעיה זו כבר ביחידה 6. בובאנו לבצע תרגום מוכנה, בעיה זו מקבלת משנה תוקף. חשבו לדוגמה על זוג המשפטים :

"I have a dog, and I love it"  
"יש לי כלב, ואני אוהבת אותו"

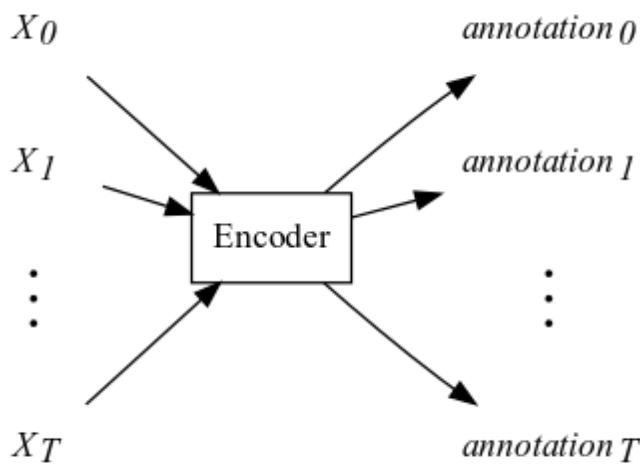
כדי לתרגם נוכנה מאנגלית לעברית את המילה الأخيرة בזוג זה, יהיה על הרשות לזהות שהמילה "אותו" מתיחסת ל"dog", ולדעת שמילה זו בעברית היאimin זכר. בראשת שבינוי בפרק הקודם, קשר זה בהכרח חייב להישמר דרך כל החישובים המבוצעים במקודד לאחר הזנת המילה "dog", וכן דרך כל החישובים המבוצעים במפרש עד להזנת המילה "אותו", שכן המקודד מעבד את המשפט המקורי מתחילה לסופו, מייצר את וקטור ההקשר, ולאחר מכן בונה את המשפט המתורגם, שוב מההתחלה אל הסוף.

כדי להקל על הרשות בלמידת קשרים מסווג זה, נרצה לספק למפרש גישה מיידית לכל המילים במשפט המקורי, כך שבדוגמת התרגומים הקודמת היא תוכל (באופן אידיאלי) להבין בעת עובדתה שהמילה الأخيرة במשפט צריכה להיות כינוי גוף; וכן כדי להחליט אם מדובר ב"הוא", "היא" או "זה", היא תוכל לחפש את שם העצם המתאים במשפט המקורי, מוביל להסתמך על וקטור ההקשר שהוזן לתא הנשנה באיטרציה הראשונה שלו.

פונקציונליות חיפוש זו, הקרויה שכבת תשומת לב (attention layer), היא הבסיס לארכיטקטורת רשות מוצלחת במיוחד, ה-*transformer*, ונמצאת בשימוש רבות לתרגומים מוכנה, ואף מוחז בתחום של עיבוד שפה טבעי. בפרק זה נראה צורה פשוטה שלה, ונראה כיצד לשולב בין המפרש למקודד ברשות תרגום המוכנה במטרה להקל על הרשות בלמידת הקשרים של מילה במשפט היעד למילים במשפט המקורי.

### המקודד

ראשית, علينا לשנות את פלט המקודד שכתבנו בפרק הקודם : ברצונו לייצר כתע ייצוג חבוי של כל טוקן במשפט המקורי, כך שבמקום וקטור הקשר היחיד המסכם את המשפט כולו, המקודד יעביר כפלט וקטור אנותציה (annotation) **עבור כל אחד מהטוקנים** במשפט המקורי. לאחר מכן מוצלח, וקטורים אלו יהיו סיכום של הטוקן בהקשרו בתוך המשפט הנתון. נאייר זאת להלן.



למזהנו, כדי לממש פונקציונליות זו בקוד יש לשנות רק שורה אחת במקודד מהפרק הקודם. זכרו שפלט המודול LSTM. nn הוא היסטוריה המצביעים החבויים של השכבה העליונה בערימות התאים. עד כה העברנו הלאה רק את המצב החבוי האחרון, השיך לטוקן הקלט האחרון, אך כעת משתמש בכל ההיסטוריה: המצב החבוי  $h_t$  יהיה האנוטציה של הטוקן  $X_t$ . ראו את המקודד החדש בקטע הקוד שוללון, אשר השורות הנבדלות מהמקודד המקורי מסומנות ב-#.

```

class Encoder(nn.Module):
    def __init__(self, embed_dim, hidden_dim, RNNlayers):
        super().__init__()
        self.src_embedding = nn.Embedding(len(src_vocab),
                                         embed_dim)
        self.rnn_stack     = nn.LSTM(embed_dim,
                                     hidden_dim,
                                     RNNlayers)

    def forward(self, src_tokens):
        all_embeddings      = self.src_embedding(src_tokens)
        all_embeddings      = all_embeddings.unsqueeze(1)
        annotations, _ = self.rnn_stack(all_embeddings)      #
        return annotations.squeeze()                         #
  
```

## המפרש

שוב נשנה את הקוד שכתבנו בפרק הקודם, וicut נאפשר למפרש גישה ישירה אל האנוטציות: לפני כל צעד עדכון של התא הנשנה נחשב **וקטור הקשר** רטבי בעזרת שכבת תושמת הלב, שנלמד עליה בפירות במחזור הפרק. לעת עתה נסתפק בתובנה שכבה זו קיבל קלט את המצב החבוי הנוכחי של המפרש, ותחשב עבורו ממוצע משוקל של האנוטציות, כאשר טוקני משפט המקור הרלוונטיים ביותר לצעד המפרש הנוכחי יזכו לבכורה בממוצע זה. בנוסחה,

$$c_t = \sum_{k=0}^T \alpha(h_t, annotation_k) \cdot annotation_k$$

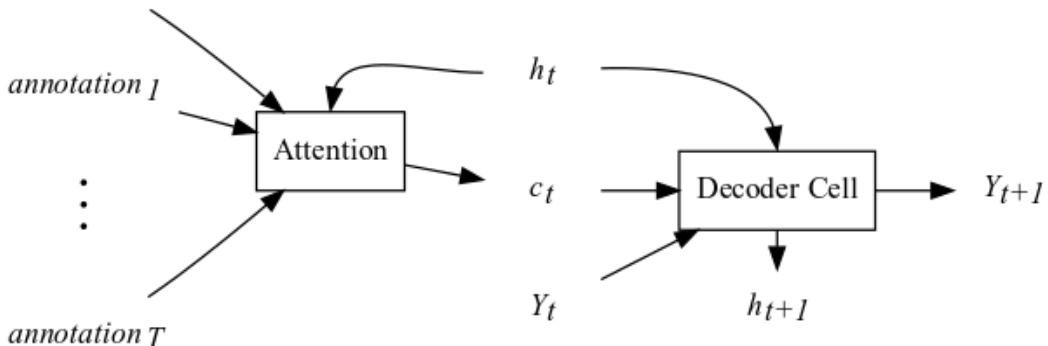
$\alpha$  היא הפונקציה שלפיה מתקבלת החלטה אם האנוטציה היא רלוונטית לייצרת הטוקן הבא בתור במשפט הפלט. אם זה המצב, על משקלה של אනוטציה זו בוקטור הקשר הנוכחי להיות גבוה. שימושו

לב שլפי נסחה זו ממד  $c_t$  זהה לזה של האנווטציות, שמדובר כמדד המרחב החבוי של המקודד. לאחר קבלת וקטור ההקשר הפרטני, התא הנשנה ישלו בצד עדכון המצב החבוי כדלהלן,

$$h_{t+1} = \tanh(W_{input} X + b_{input} + W_{hidden} h_t + b_{hidden} + W_c c_t + b_c)$$

מלבד תוספת זו, החישוב בתא הנשנה בעל מנגנון תשומת הלב נשאר זהה. נסיים דיוון זה באירוע סכמטי של התהילך החישובי של איטרציה יחידה במפרש, ומימוש התא החדש בקורס.

*annotation 0*



```

class AttnDecoderRNNCell(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_state = torch.zeros(hidden_dim)
        self.RNNcell = ContextRNNCell(embed_dim, hidden_dim) # hidden_dim
        self.output_linear = nn.Linear(in_features=hidden_dim, out_features=len(tgt_vocab))
        self.logsoftmax = nn.LogSoftmax(dim=0)

    def forward(self, one_embedded_token, attn_context):
        new_state = self.RNNcell(one_embedded_token, self.hidden_state, attn_context) # 
        tgt_token_scores = self.output_linear(new_state)
        tgt_token_logprobs = self.logsoftmax(tgt_token_scores)
        self.hidden_state = new_state
        return tgt_token_logprobs

```

השו תא זה לתא הנשנה DecoderRNNCell מהפרק הקודם וראו כי החידוש בא לידי ביטוי בשורות המסומנות ב '#' בלבד. בראשונה שבחן אנו מאתחלים תא אלמן מסווג ContextRNNCell, אשר מבצע את עדכון המצב החבוי על סמך שלושה וקטורי קלט, החדש שבהם הוא ההקשר הפרטני,  $c_t$ . ניתן לראות זאת גם בשורה השנייה המסומנת, שם התא מקבל קלט נוסף: פלט שכבת תשומת הלב.

כדי לשלב תא זה בתוך המקודד וליציר את טוקני המשפט המתורגם, זה אחר זה, יהיה علينا לחשב ולהעביר לו את וקטור ההקשר בכל איטרציה, כפי שניכר מהפרמטרים של המתודה forward שלו. געשה זאת בעזרת שכבת תשומת הלב, וcutout נדוע בה בפירות.

## תשומת לב

מנגנון תשומת הלב שעובד השראה מchiposh במאגר נתונים, ולפיכך נפתח את הדיוון בדוגמה זו. הניחו כי ברשותנו מיליון המורכב מזוגות של מפתחות וערכים, למשל,

```
values = ["woman", "man", "bicycle", "queen", "house"]
keys    = values
my_dict = dict(zip(keys, values))
pprint(my_dict)

{'bicycle': 'bicycle',
 'house': 'house',
 'man': 'man',
 'queen': 'queen',
 'woman': 'woman'}
```

פלט:

ראו כי במיילון זה השתמשנו בערכים עצם בטור המפתחות לצורך פשטות הדוגמה.

כשנרצה לחפש מילה במילון זה נבצע השוואה של שאלתת החיפוש למפתחות (למעשה ההשוואה מתבצעת לאחר מעבר לפונקציית gibob, איך אין זה רלוונטי עבורנו), ואם קיימים ערך במילון בעל מפתח זה בדיק, הוא יוחזר. אם אין במילון מפתח זהה לשאלתה, לא יוחזר דבר. ראו דוגמה לכך להלן.

```
print(my_dict.get("woman"), my_dict.get("girl"))

woman None
```

פלט:

למטרתנו, נרצה שהחיפוש המילוני יהיה "רץ": למשל בזודענו שהמילה "girl" בעלת קשר סמנטי הדוק למילה "woman", נצפה לראותה כתוצאה החיפוש, אך שפתחה זהה לשאלתה "girl" לשאלתה "girl" לא קיים במילון. בבואנו ממש רצון זה עומדים לפניו שני אטגררים:

1. המרת המפתחות והשאלות לייצוג המביא לידי ביטוי את הקשר הסמנטי בין מילים.
2. מדידת הדמיון בין שתי מילים על סמך ייצוג זה, שכן נרצה להחזיר כפלט החיפוש את המילה מהמילון שדומה ביותר לשאלתה.

המשך הראונה נראה מأتגרת על פניה, אך לאחר מחשבה נוכל להסיק שכבר ראיינו את פתרונה – אלו הם שיכוני המילים המאומנים מראש של GloVe, שפגשו לראשונה כאשר עסקנו בעיבוד מקדים של נתוני טקסט ביזידה 6. נייבא אותם שוב, וcutout נציג את מפתחות המילון שלנו להיות וקטורי השיכון של המילים המתאימות.

```

from torchtext.vocab import GloVe
glove_embedder = GloVe(name='6B', dim=50)
keys = glove_embedder.get_vecs_by_tokens(values)
print(keys.size(), keys.dtype)
torch.Size([5, 50]) torch.float32

```

**פלט:**

שימוש לב שכעת מפתחות המילים במיילון הם וקטורים ממשיים למרחב בעל 50 ממדים.

כאשר בידינו שיכון המילים, פתרון הבעה השנייה נעשה פשוט – כל מdad של דמיון או קרבה בין שני וקטורים יבצע עבודה סבירה. כנהוג בשימושי עיבוד שפה טבעיות, נבחר לעת עתה את דמיון הקוסינוס (Cosine similarity) כדי לדמיון הסמנטי. על בסיס השיכונים של שתי מילים, שניהם וקטורים של מספרים ממשיים באותו מרחב, פונקציית דמיון זו מחשבת את **קוסינוס הזוויות** בין שני הווקטורים, לפי הנוסחה זו :

$$S_c(q, k) = \frac{q \cdot k}{\|q\| \|k\|}$$

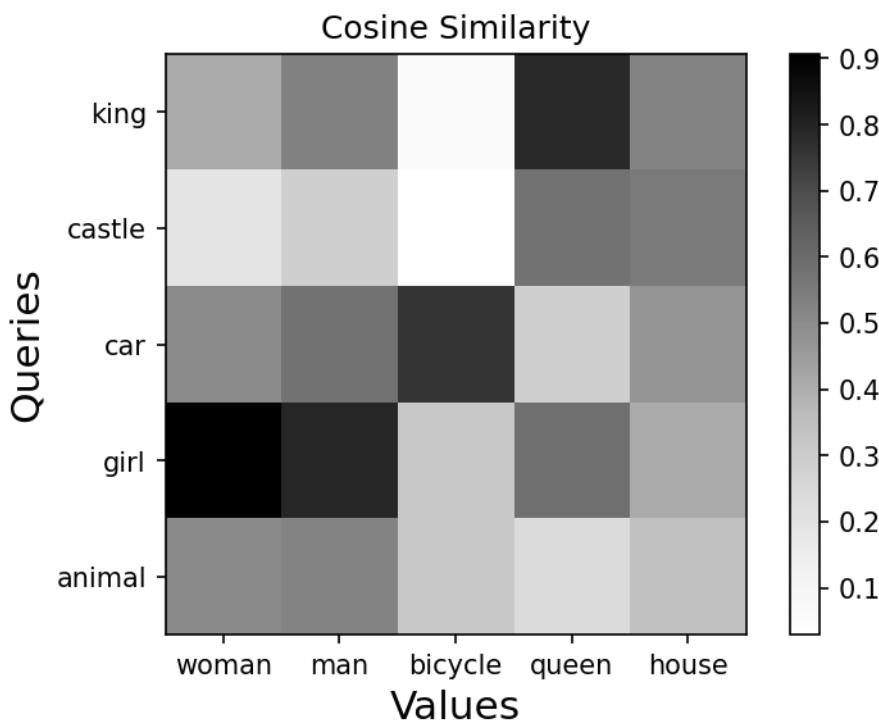
הפונקציה מחשבת את המכפלה הפנימית של וקטורי הקלט ומnormלת אותם כך שיתקבל פלט בטוחה [−1,1]. ככל שהערך  $S_c(q, k)$  גדול יותר, כך הזוויות בין הווקטורים קרובה יותר לאפס, ולפיכך השאלה  $q$  דומה יותר למפתח  $k$ . בעזרת פונקציה זו נמשח את החיפוש הרץ במיילון בקורס.

```

def soft_search(query, keys, values):
    dict_size      = len(values)
    cos_similarity = torch.zeros(dict_size)
    q_norm         = query.norm()
    for idx in range(dict_size):
        dot          = (query * keys[idx]).sum()
        k_norm       = keys[idx].norm()
        cos_similarity[idx] = dot / (q_norm * k_norm)
    best_match_idx = cos_similarity.argmax()
    return values[best_match_idx], cos_similarity

```

ראו כי פונקציה זו מצפה לקבל את השאלה המשוכנת באמצעות GloVe בפרמטר `query` וסדרה של מפתחות בפרמטר `keys`; אלו יהיו שיכוני המילים הנמצאות בפרמטר `values`. תוצאות החישוב עברו שאלות אחזות לדוגמה מאוריות להלן, כאשר הערך הכהה ביותר מצביע על תוצאת החיפוש עבור השאלה בשורה זו.



מ מבט באIOR שלעיל, נסיק של עתים ישנו כמה תשובות הולומות עבור שאלתה נתונה, למשל ל- "king".  
קשר סמנטי חזק ל- "queen", כמובן, אך גם ל- "man". במשמעות הנוכחי, תוצאה החיפוש אינה מביאה זאת בחשבון, שכן התוצאה הדומה ביותר בלבד היא שמהזורת כפלט. נתקו זאת באמצעות חישוב ממוצע משוקל עבור הפלט, ובזהדנות זו גם יותר על חישוב הנורמות, במטרה ליעיל את זמן הריצה של הפונקציה. התוצאה המתקבלת היא פונקציית תשומת לב המכפלה הפנימית (dot product) של הפקודה. (attention).

```
def dot_attention(q, K, V):
    dict_size = K.size(0)
    similarity = torch.zeros(dict_size)
    for idx in range(dict_size):
        similarity[idx] = (q*K[idx,:]).sum()
    attn_weights = F.softmax(similarity, dim=0)
    weighted_V = torch.zeros_like(V)
    for idx in range(dict_size):
        weighted_V[idx,:] = attn_weights[idx]*V[idx,:]
    output = weighted_V.sum(dim=0)
    return output, attn_weights
```

נ notch פעולה פונקציה זו בפרוטרוט :

- ראשית, כמו קודם, אנו מחשבים ממד דמיון בין השאלה  $q$  לבין כל אחד מהഫוחות, שם עמודותיה של המטריצה  $K$ .
- אחרי כן אנו מנורמלים את ממד הדמיון בעזרת הפונקציה  $\text{softmax}$ , שכן ברצוננו להשתמש בו כמשקל לחישוב ממוצע משוקל.
- התוצאה המתקבלת ב-  $\text{attn\_weights}$  היא וקטור שסכום איבריו הוא 1, וכך שהשאלה דומה יותר למפתח מסוים, כך משקלו של וקטור זה עולה בתוצאות החיפוש.

- לבסוף, ניכר כי הפלט של פונקציית חיפוש זו הוא וקטור יחיד, המmoצע המשוקל של עמודות המטריצה  $V$ . אלו הם הערכים במילון המתאיםים למפתחות ב- $A$ .

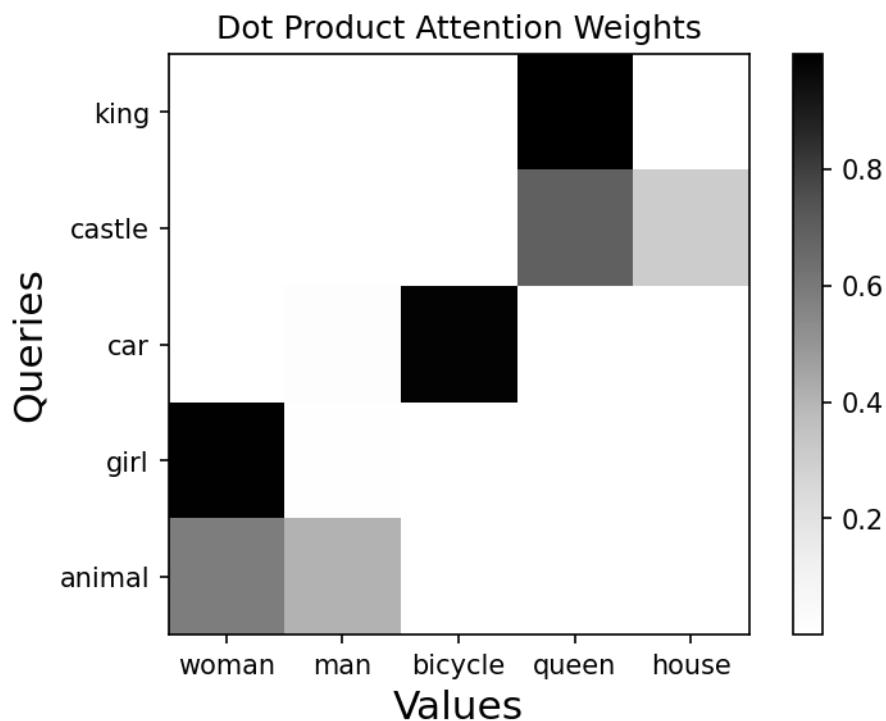
שימוש לב שבעת גם על **ערבי** המילון להיות וקטוריים ממשיים, כדי לאפשר את חישוב המmoצע. לפיכך, עבור דוגמת המילון שלנו, נעביר לפונקציית תשומת הלב את המילים לאחר השיכון באמצעות GloVe, כך שהמטריצות  $K$  ו- $V$  יהיו זהות. בקטע הקוד של להן נדגים את השימוש בפונקציה.

```

src_words = ["woman", "man", "bicycle", "queen", "house"]
values     = glove_embedder.get_vecs_by_tokens(src_words)
keys       = values
query      = glove_embedder.get_vecs_by_tokens("king")
search_result, attn_weights = dot_attention(query, keys, values)
print(search_result.size(), search_result.dtype)
print(attn_weights)

平淡:
torch.Size([50]) torch.float32
tensor([[7.0899e-05, 9.9381e-04, 1.3337e-09, 9.9831e-01,
6.2366e-04]])
```

ראו כי תוצאת החיפוש אף היא וקטור באוטו מרחב שיכון בעל 50 ממדים. במבט למשקלים המmoצע ניכר כי למילה "queen" ההשפעה הרבה ביותר על הערך המתקבל, באופן המתאים לדמיון הסמנטי בין מילה זו לשאלתה. להלן נאייר כמה תוצאות נוספות של שימוש בפונקציית תשומת לב המכפלה הפנימית.



מהאייר ניכר למשל שהטזאת החיפוש עבור השאלה "castle" היא בעיקרה ממוצע של שכוני המילים "queen" ו- "house", כצפוי.

## чисוב תשומת הלב במנפרש

כעת ברשותנו כל הדורש כדי לתאר ולמשש את החישוב המבוצע במנפרש אשר ישמש לתרגום המשפט. נניח מאחרינו את דוגמת היפוש המילים במילון ונזכיר שעליינו לייצר עתה בכל איטרציה וקטור הקשר פרטי עבור התא הנשנה. וקטור זה יהיה תוצאת היפוש של שכבת תשומת לב המכפלה הפנימית עם החלט זהה: הערכים והפתרונות של השכבה יהיו וקטורי האנותציה שיצר המקודד, והשאילתת תהיה המצב החבוי הנוכחי. אם כן, חישוב תשומת הלב המבוצע לפני יצירת הטוקן ה- $i$  במשפט המתורגם הוא:

$$c_i = \sum_{k=0}^T \alpha(h_i, annotation_k) \cdot annotation_k$$

כאשר

$$\alpha(h_i, annotation_k) = \text{softmax} \begin{pmatrix} h_i \cdot annotation_0 \\ h_i \cdot annotation_1 \\ \vdots \\ h_i \cdot annotation_T \end{pmatrix}$$

שימוש לב沈די לבצע חישוב זה ללא תקלת, ממד המצב החבוי של המפרש,  $h_i$ , צריך להיות זהה לזה של המקודד, זהו ממד וקטורי האנותציות. בשימוש בתשומת לב אנואפשרים לרשות לייצג את המצב החבוי  $h_i$  בצורה דומה לאנותציות הרלוונטיות לתרגום הטוקן הבא בתור. כמובן, כדי לנצל אפשרויות זו יהיה על הרשות ללמידה לעשות זאת בתהיליך האימון. קוד המפרש מופיע בסוף הפרק. התבוננו בו וראו כי הוא דומה ברובו למפרש ללא תשומת הלב, כפי שתتبנו בפרק הקודם.

לבסוף נחבר את המפרש למקודד ונקבל את הרשות המלאה, כדלהלן.

```
class Translator(nn.Module):
    def __init__(self, embed_dim, hidden_dim, encoder_layers):
        super().__init__()
        self.encoder = Encoder(embed_dim,
                              hidden_dim,
                              encoder_layers)
        self.decoder = AttnDecoder(embed_dim, hidden_dim)
    def forward(self, src_tokens, tgt_tokens):
        annotations = self.encoder(src_tokens)
        return self.decoder(annotations, tgt_tokens)
```

נסיים פרק זה בהסתיגות: בIMPLEMENTATION הרשות שמננו דגש על בהירות המימוש ופשטות החישוב, וזאת על פניו ייעילותו ואפקטיביות המודל הנלמד. וביתר פירוט:

- השתמשנו באוסף נתונים אשר בנוי מנתנדים, שידוע שיש בו טעויות.
- בעת העיבוד המקדדים ביצענו טוקנייזציה בצורה פשוטה ביותר האפשרית.
- השתמשנו במנפרש בעל שכבה ייחודית של RNN המבוססת על תא אלמן, התא הנשנה הפשטוט ביותר.
- מימשו את החישוב הרקורסיבי בתא הנשנה של המפרש וכן את החישוב המבוצע בשכבת תשומת הלב באמצעות לולאות פירטו, אשר השימוש בהן אינו יעיל.
- השתמשנו בצורה פשוטה ביותר של שכבת תשומת הלב, שלא גרשאות רבות ומתחוכמות יותר.

התוצאה המתקבלת מהחלהות אלו היא שהרשות שלנו אמונה מסוגלת ללמידה, אך היא עשויה זאת באטיות רבה ואין להיא מספקת תוצאות טובות במיוחד. על כן, לאחר קריאת פרק זה וקדומו עליהם להבין את עקרון הפעולה של רכיביה, ועם זאת לזכור שדרושים שיפורים טכניים ותיאורתיים נוספים כדי למלא את מילוי יכולתם.

## שאלות לתרגול

1. כתבו את המחלקה ContextRNNCell המממשת תא אלמן בעל שלושה וקטורי קלט שונים: המצב הנוכחי, טוקן הקלט הנוכחי וקטור ההקשר הפרטאי.  
רמז: השתמשו בשכבות LINAR כדי לכל אחד מווקטורי הקלט, ולבסוף הפעילו עליהם את פונקציית האקטיבציה הדרשיה. השיקעו מחשבה במודול השכבות הLINAR.
  2. כתבו את פונקציית תשומת לב המכפלה הפנימית בצורה וקטורית, ללא שימוש בלאוות.
  3. תכננו וכתבו את מצב החיזוי של המפרש עם תשומת הלב.  
רמז: שabay הראה מצב החיזוי של המקודד מהפרק הקודם.
  4. הניחו שנתנו משפט קלט באורך  $T_1$  טוקנים אשר עברו אובייקט  $\text{Translator } T_2$  מייצר  $T_2$  טוקנים כפלט. כמה פעמים המפרש קורא לפונקציית תשומת הלב? כמה פעמים מבוצע חישוב  $k \cdot q$  עבור?
- שאילתת ומפתח יחידים?

## נספח: קוד המפרש בעל שכבת תשומת לב

```
class AttnDecoder(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.tgt_embedding = nn.Embedding(len(tgt_vocab),
                                         embed_dim)
        self.RNNcell = AttnDecoderRNNCell(embed_dim,
                                         hidden_dim)
        self.attn_layer = dot_attention
    def forward(self, annotations, tgt_tokens):
        self.RNNcell.hidden_state = torch.zeros(self.hidden_dim)
        keys = annotations
        values = annotations
        translated_tokens = [START_Token]
        sentence_loss = 0
        for idx in range(len(tgt_tokens)-1):
            previous_token = translated_tokens[idx]
            embedded_token = self.tgt_embedding(previous_token)
            query = self.RNNcell.hidden_state
            attn_context, _ = self.attn_layer(query, keys, values)
            logprobs = self.RNNcell(embedded_token,
                                   attn_context)
            predicted_token = logprobs.argmax()
            translated_tokens.append(predicted_token.detach())

            correct_token = tgt_tokens[idx+1]
            token_loss = -logprobs[correct_token]
            sentence_loss += token_loss

            if predicted_token == END_Token:
                break
        return translated_tokens, sentence_loss
```

מהדורות פנימיות  
לא להפצה ולא למכירה  
מק"ט 22961-0000