ראייה ממוחשבת

תאריך: _____18.12.2020_____

שם סטודנט: _____אביב כספי_____

מספר סטודנט: _____311136691_____

מייל: _____avivcaspi@campus.technion.ac.il_____

מספר גיליון: _____2_____

שאלה 1

<u>סעיף 1:</u>

בסעיף זה חישבנו את הפרצוף הממוצע, על ידי חישוב ערך ממוצע לכל פיקסל בתמונות בסט האימון.
הפרצוף הממוצע שקיבלנו הוא:

כעת חיסרנו את פרצוף זה מכל סט האימון

<u>סעיף 2:</u>

בסעיף זה חישבנו את *r* הוקטורים העצמיים השייכים ל*r* הערכים העצמיים הגדולים ביותר.
על מנת לחשב ערכים אלה בצורה יעילה ללא צורך בחיפוש של מטריצת הקוואריאנס הענקית (77760*X*77760), השתמשנו בפירוק *SVD* כמו שראינו בתרגול.
כאשר קיבלנו פירוק מהצורה: $A = U\Sigma V^T$
כאשר הוקטורים העצמים נמצאים במטריצה *V*, ומסודרים לפי גודל הע"ע המתאים להם.
המשוואה נכונה לפי הפירוט הבא:
מטריצת הדאטה שלנו *A* היא מגודל *n x p* , כאשר *n* זה מספר הדוגמאות, ו-*p* מספר הפיצ'רים (פיקסלים).
ומטריצת הקוואריאנס *C* היא מגודל *p x p* כאשר נוכל לחשב אותה לפי הנוסחא הבאה:

$$C = \frac{A^T A}{n-1}$$

זו מטריצה סימטרית, לכן ניתן לבצע לה לכסון מהצורה הבאה:

$$C = VLV^T$$

כאשר *V* זו מטריצת הוקטורים העצמיים ו*L* היא מטריצה אלכסונית עם ערכים עצמיים על האלכסון.
כעת נוכל לבצע פירוק *SVD* ל *A* ולקבל:

$$A = U\Sigma V^T$$

כאשר *U* מטריצה יוניטרית ו- Σ מטריצה אלכסונית עם ערכים סינגולרים, לכן נוכל להסיק שמתקיים:

$$C = \frac{A^T A}{n-1} = \frac{V\Sigma U^T U\Sigma V^T}{n-1} = V\frac{\Sigma^2}{n-1}V^T$$

לכן נקבל כי V היא בדיוק מטריצת הוקטורים העצמיים של C.

כעת נציג את 5 הפנים העצמיים שקיבלנו על ידי החישוב הנ"ל:

<u>סעיף 3:</u>

החישובים של הוקטורים המייצגים של כל סט האימון, וביצוע השיחזור של התמונות מפורט בקוד,
להלן דוגמא של שיחזור של אחת התמונות מסט האימון:

השגיאה *RMSE* שקיבלנו היא 13.13%, זאת שגיאה טיפה גדולה. כאשר השגיאה חושבה בצורה הבאה:

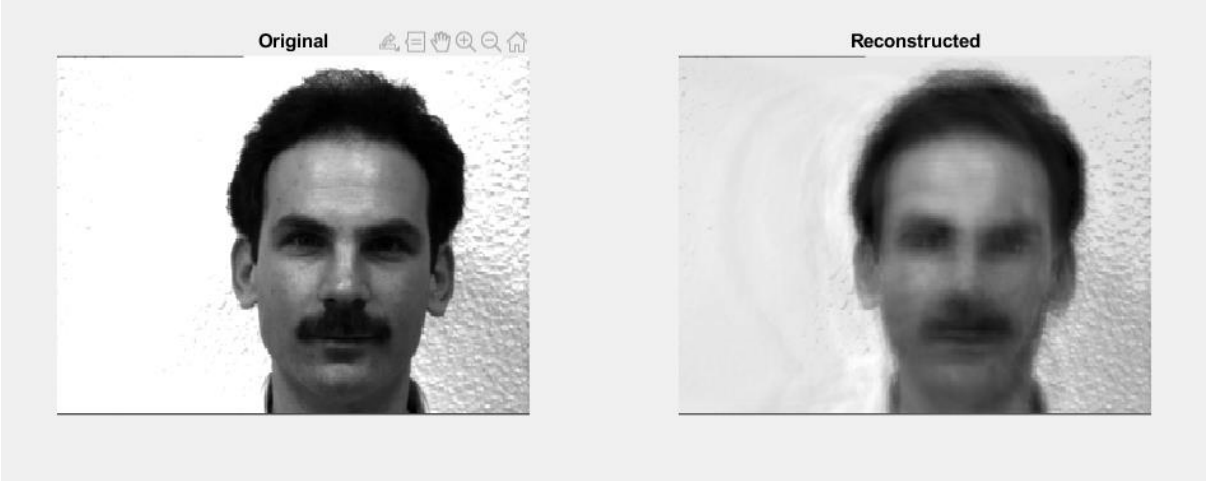
$$e = \frac{100}{255} * \frac{1}{150}\sum_{i=1}^{150}\sqrt{\frac{1}{77760}\sum_{k=1}^{77760}(Ai - Ai_{reconstruct})^2}$$

יכלנו לצפות לשגיאה גדולה, מכמה סיבות.
תחילה בגלל שלקחנו רק 25 וקטורים עצמיים מתוך 150, לכן השחזור אינו מושלם.
בנוסף אם נסתכל על התמונות שקיבלנו של הפנים העצמיים, נשים לב כי התמונות בסט האימון לא ממורכזות, לכן כל הפנים בסט לא נמצאות בדיוק באותן הנק', והתמונות הנ"ל לא יצאו בדמות אדם יחיד. זה כמובן פגע בנוסף באיכות השחזור. בנוסף יש הבדלי תאורה וצל שונים שגם פוגעים באיכות הוקטורים העצמיים.

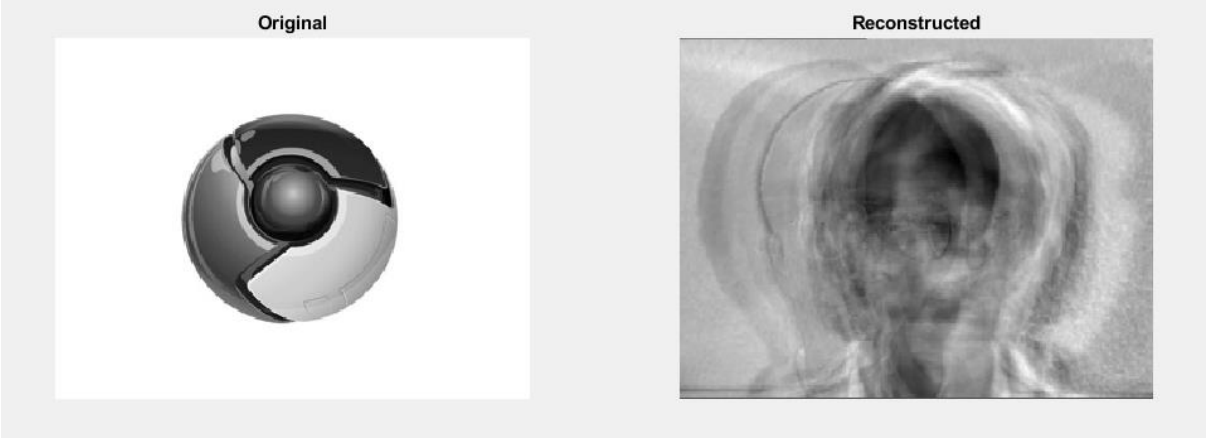
<u>סעיף 4:</u>

תחילה נשים לב כי בסט האימון יש תמונות שלא מופיעות בסט המבחן, ובנוסף בחלקן אין אנשים, כלומר כנראה שלא יהיה אפשרי לשחזר בצורה טובה תמונה של כדור על ידי תמונות בסיס של פרצופים.
ממוצע שגיאת השחזור לכלל סט המבחן הוא: 19.98% , נשים לב שקיבלנו שגיאה יותר גבוהה ממקודם, מהסיבה שחלק מהתמונות כלל לא של פרצופים.
נציג כעת דוגמא לשחזור מוצלח ושחזור גרוע:

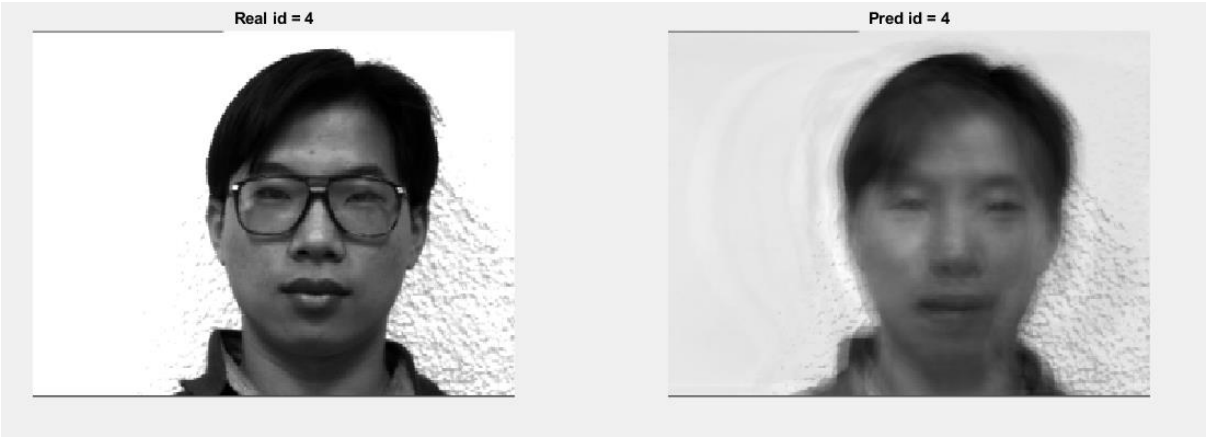
שחזור מוצלח



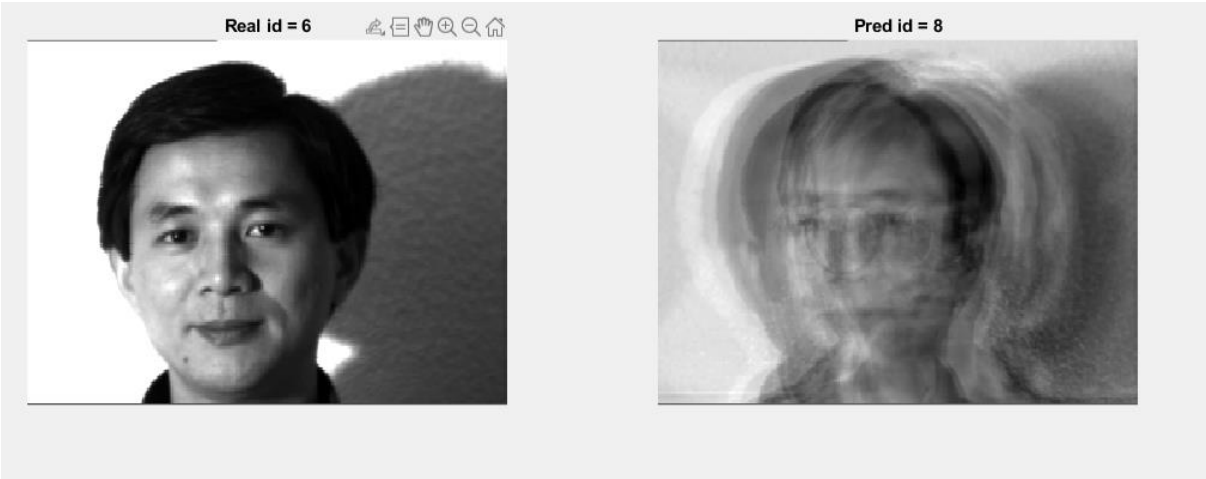
שחזור גרוע



כעת השתמשנו במודל *KNN*, על מנת לבצע זיהוי של האנשים בתמונות.
אחוז ההצלחה שהגענו עליו הוא *81.82%* כאשר מתוך 11 תמונות בסט המבחן (של אנשים מסט האימון) הצלחנו לזהות 9 תמונות.
דוגמא לזיהוי נכון:



התמונות שזוהו בצורה שגויה:



לדעתי השגיאות בזיהוי נבעו מייצוג לא מספיק טוב של התמונה במימד הנמוך יותר.
נוכל לשים לב כי התמונות המשוחזרות מאד לא ברורות, וכי עבור התמונה השנייה, מיקום הפרצוף אינו ממורכז בתמונה ולכן הייצוג שלה נראה יותר מטושטש ולא ברור. וזה כנראה הסיבה לכך שהיה טעות בחיזוי הנ"ל.
בנוסף, התמונות הנ"ל בעלי שטח גדול של צל, וכנראה זה מוסיף לשגיאה בזיהוי.

אלגוריתם שיכול לשפר את דיוק המודל:
נשים לב כי באלגוריתם הקודם, השתמשנו בוקטורים העצמיים, על מנת לחשב ייצוג לכל תמונה מתוך סט האימון שלנו, ולאחר מכן, לקחנו תמונה מסט המבחן, מצאנו ייצוג שלה לפי הוקטורים העצמיים, ומצאנו את הייצוג מתוך סט האימון הקרוב ביותר לייצוג מסט המבחן ולפי כך בחרנו את החיזוי שלנו.
בנוסף נשים לב, כי בחישוב הוקטורים העצמיים של מרחב הפרצופים, לא הבאנו בחשבון את תיוג הפרצופים אלא רק את העובדה שמדובר בפרצוף.

כעת אני מציע לשנות את דרך החיזוי בצורה הבאה:
(אני מציע דרך שלא משתמשת באותה וקטורים עצמיים שמצאנו באלגוריתם הקודם, אלא מוצאת וקטורים חדשים)
כעת, אני מציע להשתמש באלגוריתם דומה ל*PCA* , אשר נקרא *LDA* (*Linear Discriminative Analysis*).
אלגוריתם *PCA* מוצא לנו מרחב קטן יותר אשר מייצג את הדאטה שלנו בצורה טובה. (מוצא את כיווני השונות הגדולים ביותר)
כלומר אלגוריתם זה מוצלח בעיקר על מנת להחליט האם תמונה היא פרצוף או לא, כלומר תמונה שאינה פרצוף, תיוצג במרחב החדש במקום מרוחק משאר הייצוגים.
לעומת זאת, אלגוריתם *LDA*, מוצא לנו תת מרחב, כך שמטרתו היא למצוא הפרדה כמה שיותר גדולה בין קבוצות שונות של תוויות (*supervised*).

כלומר, אם נסתכל על המקרה שלנו, בו סט האימון מורכב מתמונות של 16 אנשים שונים, אלגוריתם *LDA* יחפש מרחב בו הדוגמאות של כל אדם יהיו כמה שיותר רחוקות מדוגמאות של אנשים אחרים.

תהליך האלגוריתם יהיה בצורה הבאה:
חישוב על ידי אלגוריתם *LDA* את הוקטורים המגדירים את מרחב הפרצופים בצורה שמפרידה את סט האימון הכי טוב.
חישוב הייצוג לכל אחד מהתמונות מסט האימון
חישוב הייצוג לתמונה מסט המבחן
מציאת התמונה מסט האימון שהייצוג שלה קרוב ביותר לייצוג של התמונה מסט המבחן
פלט – התווית של התמונה מסט האימון שמצאנו.

אני מניח שאלגוריתם זה יעבוד יותר טוב, בגלל ההתייחסות לתוויות בזמן חישוב הוקטורים העצמיים, לעומת *PCA*, אלגוריתם זה נועד על מנת להפריד בין האנשים הקיימים, כלומר אם התמונה לא שייכת לאף אדם מסט האימון, תוצאת האלגוריתם לא רלוונטית (לעומת *PCA* בו יכלנו לזהות תמונות שלא שייכות לאנשים לפי המרחק מכלל התמונות האחרות)

חשוב לציין, כי אלגוריתם *LDA* מבצע כמה הנחות שלא בטוח תקפות, כמו הנחה כי התפלגות התמונות של כל אדם היא נורמלית, ובנוסף כי הקוואריאנס של כל מחלקה היא זהה.

שאלה 2

<u>סעיף 3</u>

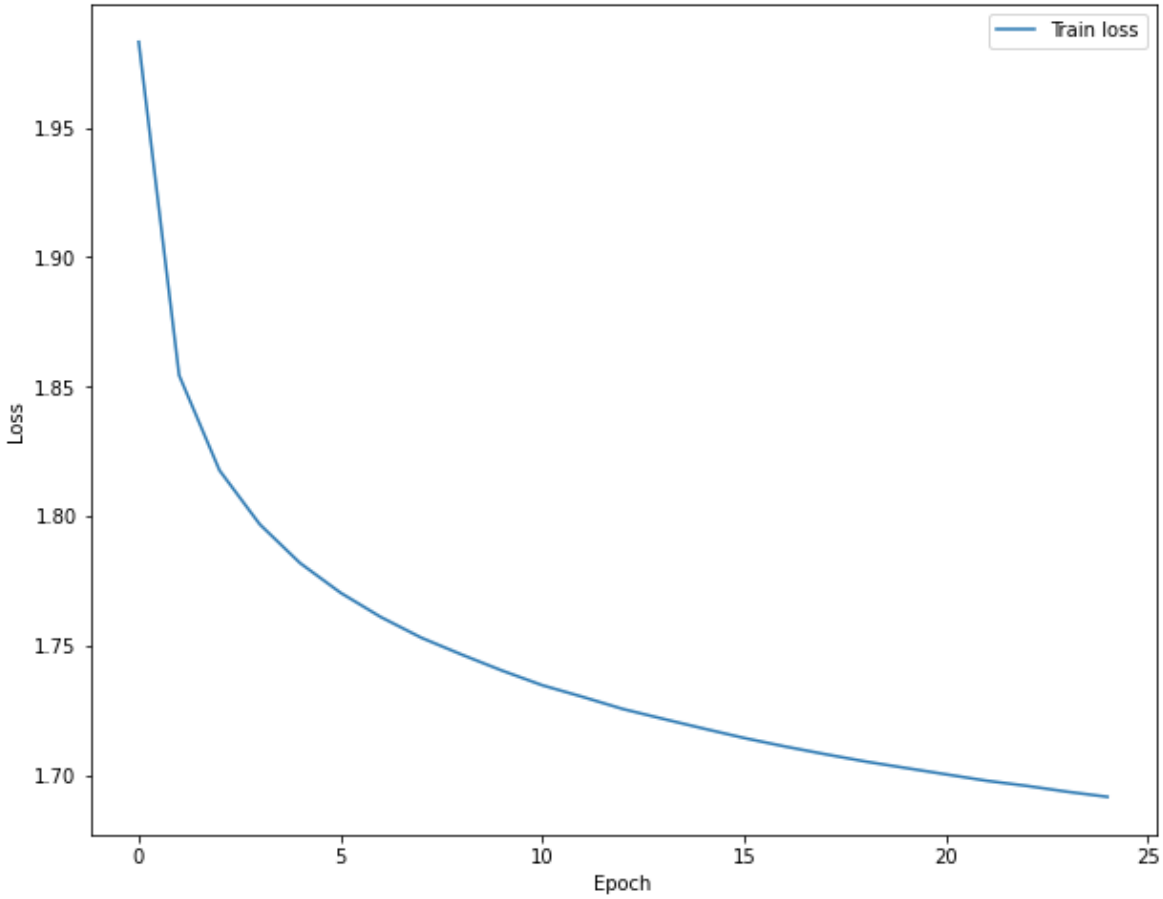
בסעיף זה אימנתי מודל לינארי ללא רגולריזציה על סט האימון של *CIFAR10*.
בזמן האימון השתמשתי בהיפר פרמטרים הבאים:

```
batch_size = 32 lr = 3e-5 num_epochs = 25
```

על מנת למצוא את הפרמטרים הנ"ל ביצעתי *cross validation* ובחרתי את הפרמטרים שהביאו לי דיוק גבוה ביותר על סט הולידציה.
המודל שהשתמשתי בו *Linear* מתוך *Pytorch*.
ב*notebook* ניתן לראות כי לא הוספתי *softmax* במודל עצמו, וזה נובע מכך שב *pytorch* פונקציית ההפסד *CrossEntropyLoss* מכילה בתוכה את ה*softmax* , ולכן אין צורך להוסיף את זה גם במודל עצמו.

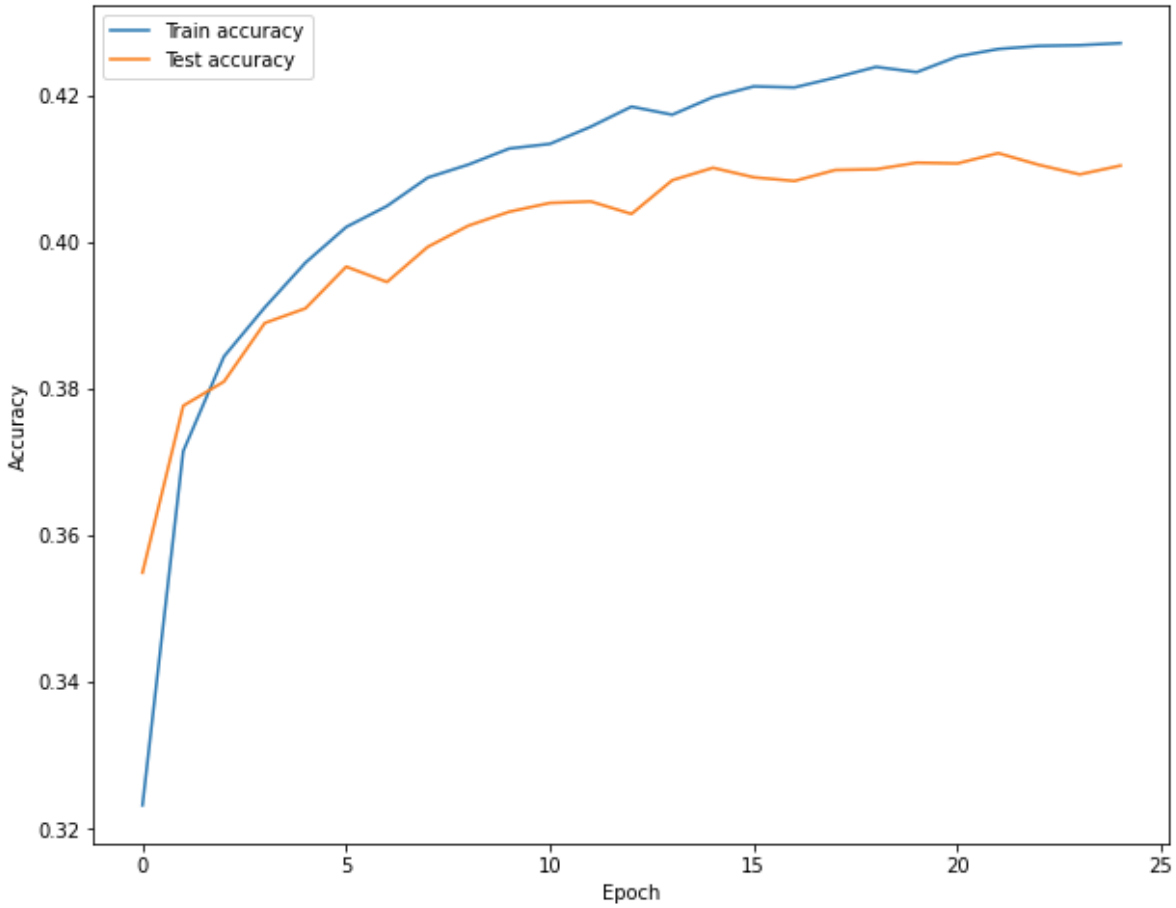
בזמן האימון השתמשתי באופיטימיזר מסוג *Adam*.

גרף פונקציית ההפסד לאורך האימון:



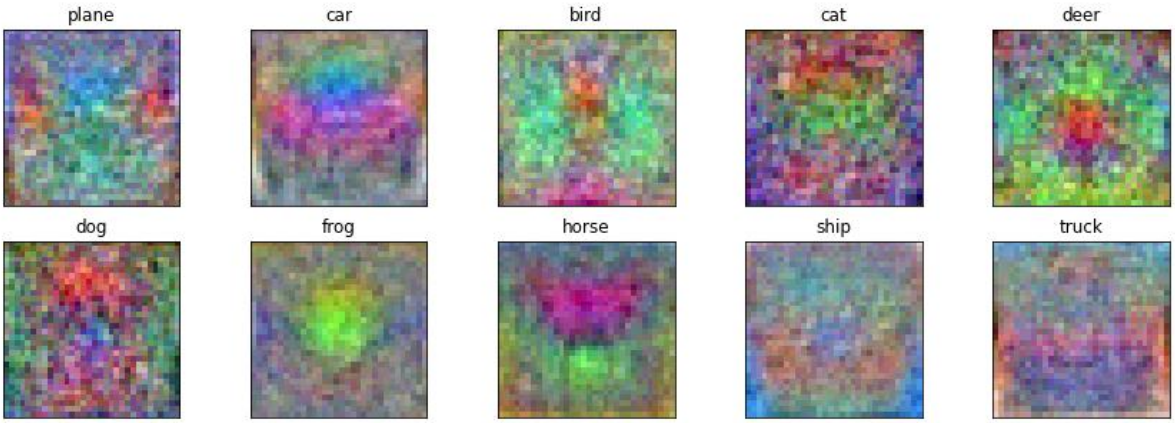
ניתן לראות בגרף כי לכל אורך האימון ההפסד שלנו יורד, כלומר המודל מצליח להתאמן.

גרף הדיוק לאורך האימון:



ניתן לראות בגרף זה כי המודל שלנו ביצע *over fitting* , כאשר לאחר בערך *12* איפוקים, הדיוק שלנו על סט המבחן נשאר בערך קבוע והדיוק על סט האימון ממשיך לעלות.
כלומר, על מנת לקבל מודל מוצלח ללא *overfitting* , נעצור את האימון יותר מוקדם.

המשקולות של המודל שקיבלנו:



ניתן בחלק מהמקומות לזהות את העצם ששייך לתווית ובחלק טיפה יותר קשה.
עבור מכונית אפשר לזהות במרכז צורה של מכונית, עבור הציפור אפשר לזהות כנפיים וגוף.
גם עבור שאר המשקולות אפשר לזהות חלקי דמות.
ניתן גם לראות כי במקרה של סוס, נראה שיש פה סוס עם שני ראשים, אחד לכל כיוון כדי להתאים לסיבוב של התמונה.

סעיף 4

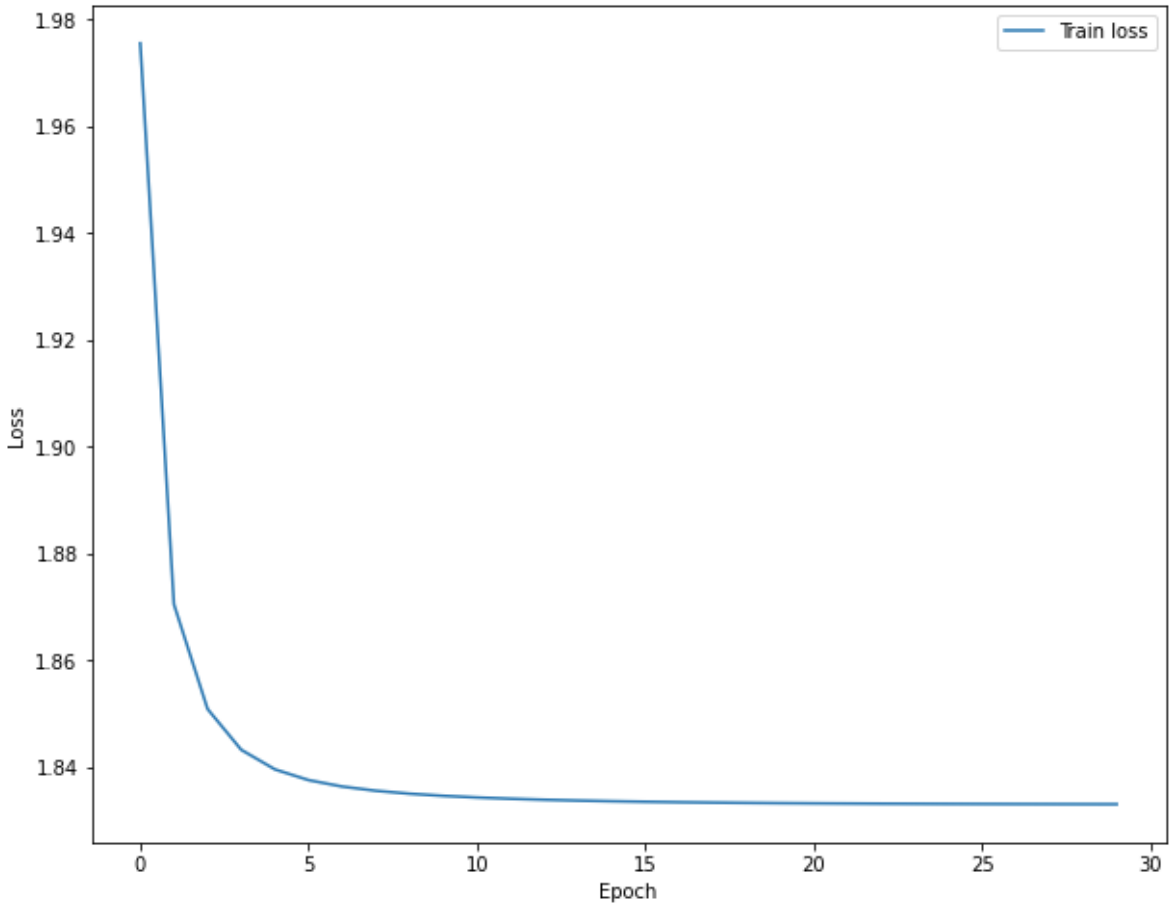
כעת אימנתי שוב מודל לינארי אך כעת הוספתי לו רגולריזציה *L2*, נשים לב כי ב*pytorch* ניתן להגדיר את הרגולריזציה הזאת בתוך ה*optimizer* כאשר זה מוגדר תחת *weight_decay*.

היפר פרמטרים לאימון זה:

```
batch_size = 32
lr = 4e-5
num_epochs = 30
reg = 0.1
```

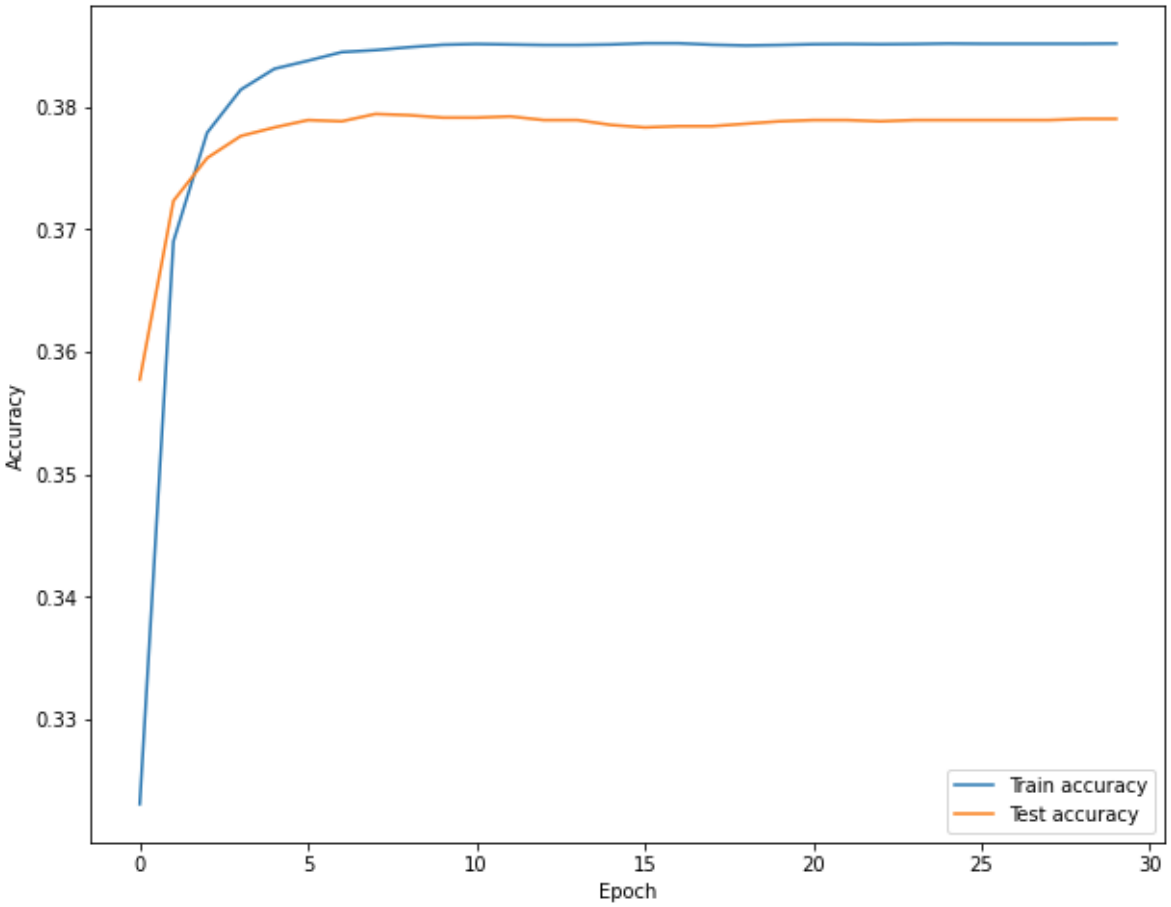
הפרמטרים נמצאו באותה שיטה כמו בסעיף הקודם למעט הרגולריזציה שנתונה.

גרף פונקציית ההפסד בזמן האימון:



כמו קודם ניתן לראות אימון מוצלח של הרשת (כאשר גם פה ניתן לראות כי כבר לאחר 8 איפוקים בערך, אין שיפור בפונקציית ההפסד).

גרף הדיוק בזמן האימון:



בדומה למקרה הקודם, ניתן לזהות כי קיבלנו *over fitting* אך לעומת המקרה הקודם, הרמה של ה *over fitting* הרבה יותר נמוכה.
בנוסף ניתן לזהות כי האימון היה הרבה יותר יציב מאשר אימון ללא רגולריזציה וכי הדיוק על סט המבחן לא קופץ כל איפוק אלא נשאר דומה לאיפוק לפני.

משקולות המודל שלנו לאחר אימון:

כעת ניתן לזהות הרבה יותר בבירור אך הדמויות המתאימות לתווית בכל משקולת.

בסעיף זה הוספנו רגולריזציה המגבילה את גודל המשקולות של המודל שלנו, כלומר הורדנו למודל דרגות חופש שהיו קיימות במודל ללא רגולריזציה מהסעיף הקודם.

כעת בזמן האימון, המשקולות לא יושפעו אך ורק מהדאטה, אלא גם מגודל המשקולות.
כלומר אם בסעיף הקודם היה לנו הרבה מודלים אפשרים המתאימים לפתרון האופטימיזציה, כעת מספר המודלים קטן.

ניתן לראות כי בזמן האימון אין קפיצות של הדיוק בזמן האימון, כלומר כל צעד שנבצע באופטימיזציה לא ישפיע בצורה קיצונית על המודל כמו בסעיף הקודם.

יתרון נוסף של הרגולריזציה זה הקטנת הסיכוי ל*over fitting* , נשים לב כי במודל החדש כמעט ולא נגרם *over fitting* למרות שאימנו 20 איפוקים יותר ממה שצריך.
זה נובע מכך שהמשקולות תלויות גם ברגולריזציה ולא רק בסט האימון, לכן ניתן להתכנס מהר לפתרון שכמעט ולא ישתנה ככל שנמשיך לאמן.

<u>סעיף 5</u>

בסעיף זה הגדרתי מודל של רשת קונבולוציה, כמו שהומלץ לנו בתרגיל, רשת מסוג *LeNet-5*.
הרשת הזאת היא מהצורה הבאה:

$$Conv(6) \rightarrow ReLU \rightarrow maxPool(2) \rightarrow Conv(16) \rightarrow ReLU \rightarrow maxPool(2) \rightarrow dense(120) \rightarrow ReLU \rightarrow dense(84) \rightarrow ReLU \rightarrow dense(10)$$

מבנה הרשת מתוך הקוד:

```
LeNet(
  (feature_extarctor): Sequential(
    (0): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=576, out_features=120, bias=True)
    (1): ReLU()
    (2): Linear(in_features=120, out_features=84, bias=True)
    (3): ReLU()
    (4): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

כעת נעבור על כל שכבה ונציין בדיוק את הפרמטרים שלה:

<u>*Feature extractor*</u>

שכבת קונבולוציה ראשונה – בשכבה זו השתמשנו ב6 פילטרים (כלומר, הפלט יהיה בעומק 6), כאשר גודל כל פילטר הוא 3*X*3
שכבת *ReLU – activation*
שכבת *maxPool* – בשכבה זו ביצענו *maxPool* עבור ריבועים בגודל *2X2* כאשר ביצענו *stride=2*, כלומר קפצנו כל פעם 2 פיקסלים על מנת לקבל פלט יותר קטן במימדים.
שכבת קונבולוציה שנייה – בשכבה זו השתמשנו ב16 פילטרים (כלומר, הפלט יהיה בעומק 16), כאשר גודל כל פילטר הוא 3*X*3
שכבת *ReLU – activation*
שכבת *maxPool* – בשכבה זו ביצענו *maxPool* עבור ריבועים בגודל *2X2* כאשר ביצענו *stride=2*, כלומר קפצנו כל פעם 2 פיקסלים על מנת לקבל פלט יותר קטן במימדים.

<u>*Classifier*</u>

שכבה לינארית ראשונה – קלט בגודל 16*6*6 (מימדי פלט החלק הקודם), פלט בגודל 120.
שכבת *ReLU – activation*
שכבה לינארית שניה – קלט בגודל 120, פלט בגודל 84.

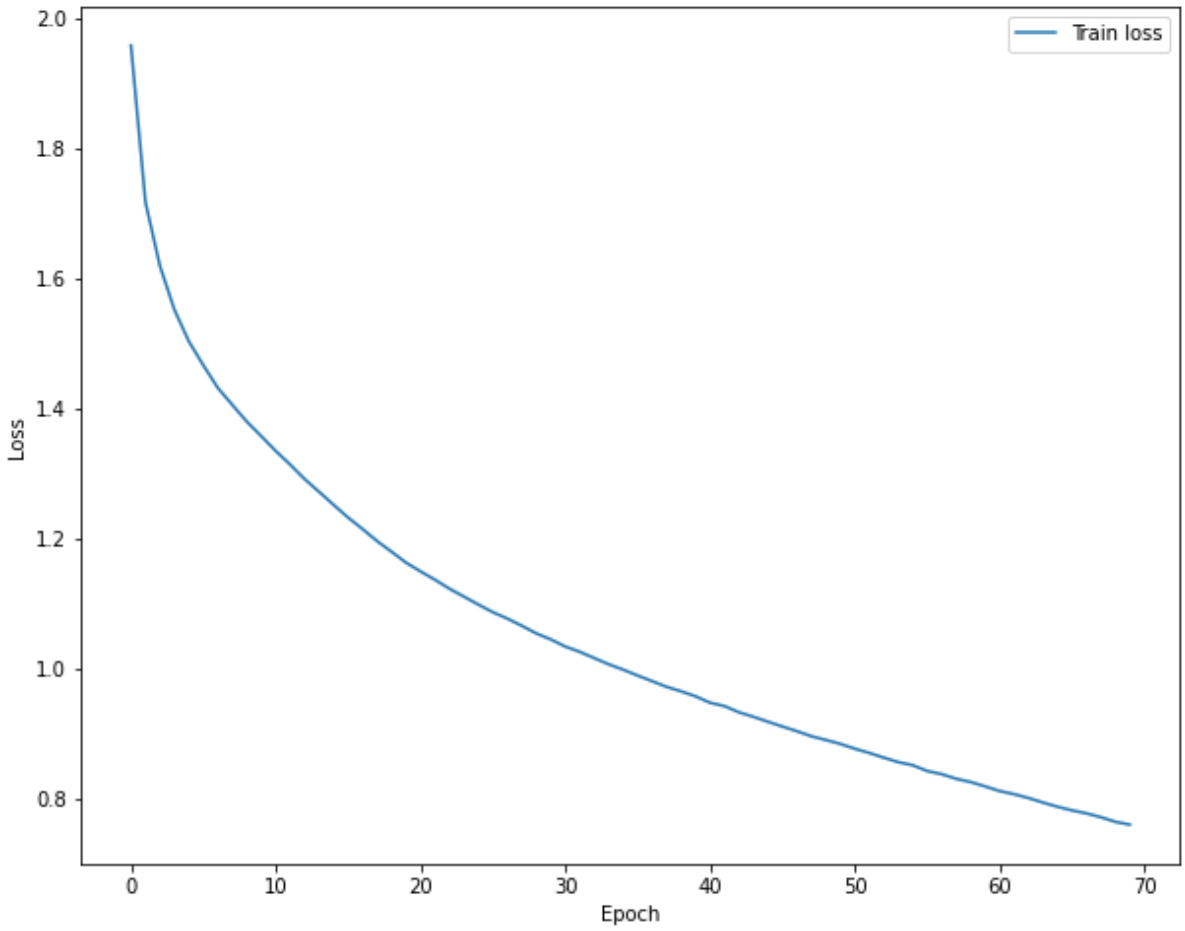
שכבת *ReLU – activation*
שכבה לינארית שלישית – קלט בגודל 84, פלט בגודל 10.

חשוב לציין כי בזמן השימוש ברשת, לאחר מעבר בחלק של הוצאת הפיצ'רים, יש לבצע שיטוח של הפלט לוקטור אחד ארוך, אשר יהיה הקלט למסווג.
בנוסף, נציין כי כמו בסעיפים הקודמים, השתמשנו בפונקציית הפסד של *CrossEntropyLoss* אשר המימוש שלה ב *pytorch* מכיל בתוכו *softmax* לכן אין צורך להוסיף זאת למודל.

היפר פרמטרים שהשתמשתי באימון:

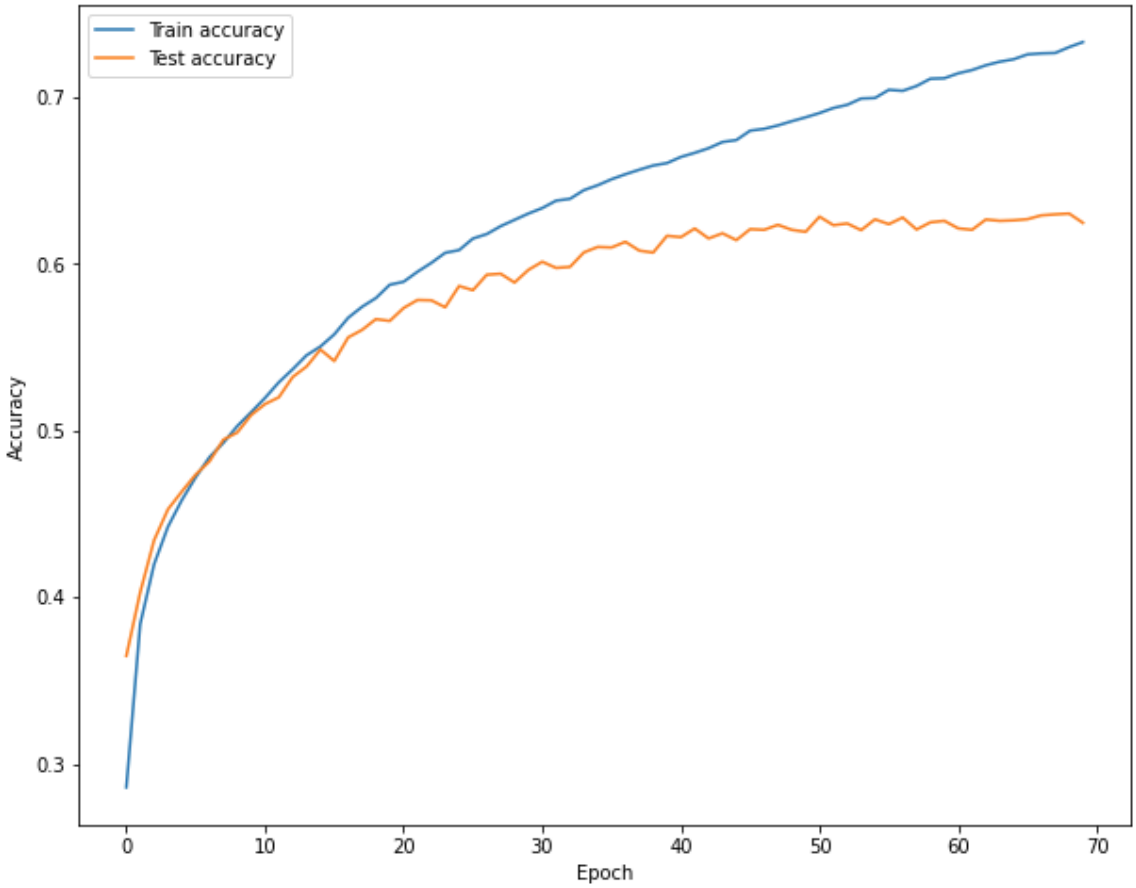
```
batch_size = 32
lr = 0.0001
num_epochs = 70
```

גרף פונקציית ההפסד בזמן האימון:



כמו קודם ניתן לראות אימון מוצלח של הרשת

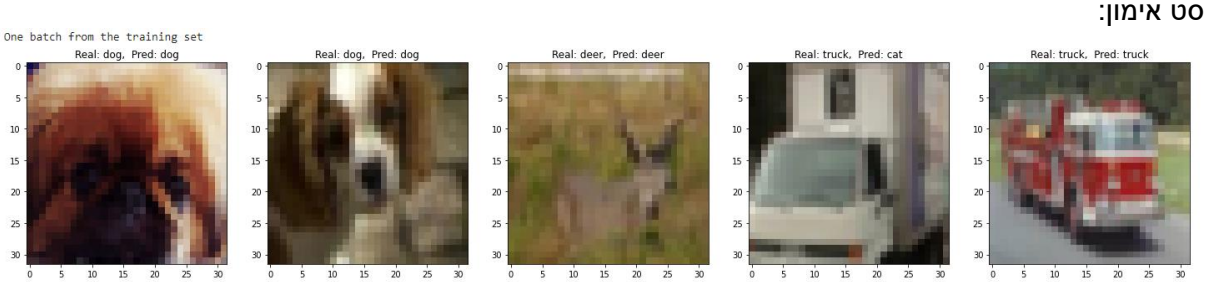
גרף הדיוק בזמן האימון:



ניתן לזהות שוב *overfitting* לאחר *25* איפוקים בערך, לכן ביצעתי אימון חוזר עם פחות איפוקים.

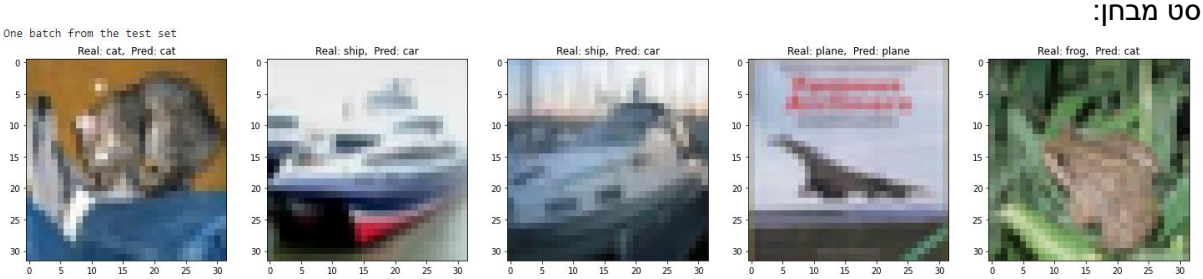
הצגת דוגמא לתוצאות המודל על סט אימון וסט מבחן:

סט אימון:



ניתן לראות כי המודל טעה עבור תמונה יחידה בדוגמא (מעל התמונה ניתן לזהות את החיזוי)

סט מבחן:



ניתן לראות כי המודל טעה עבור 3 תמונות

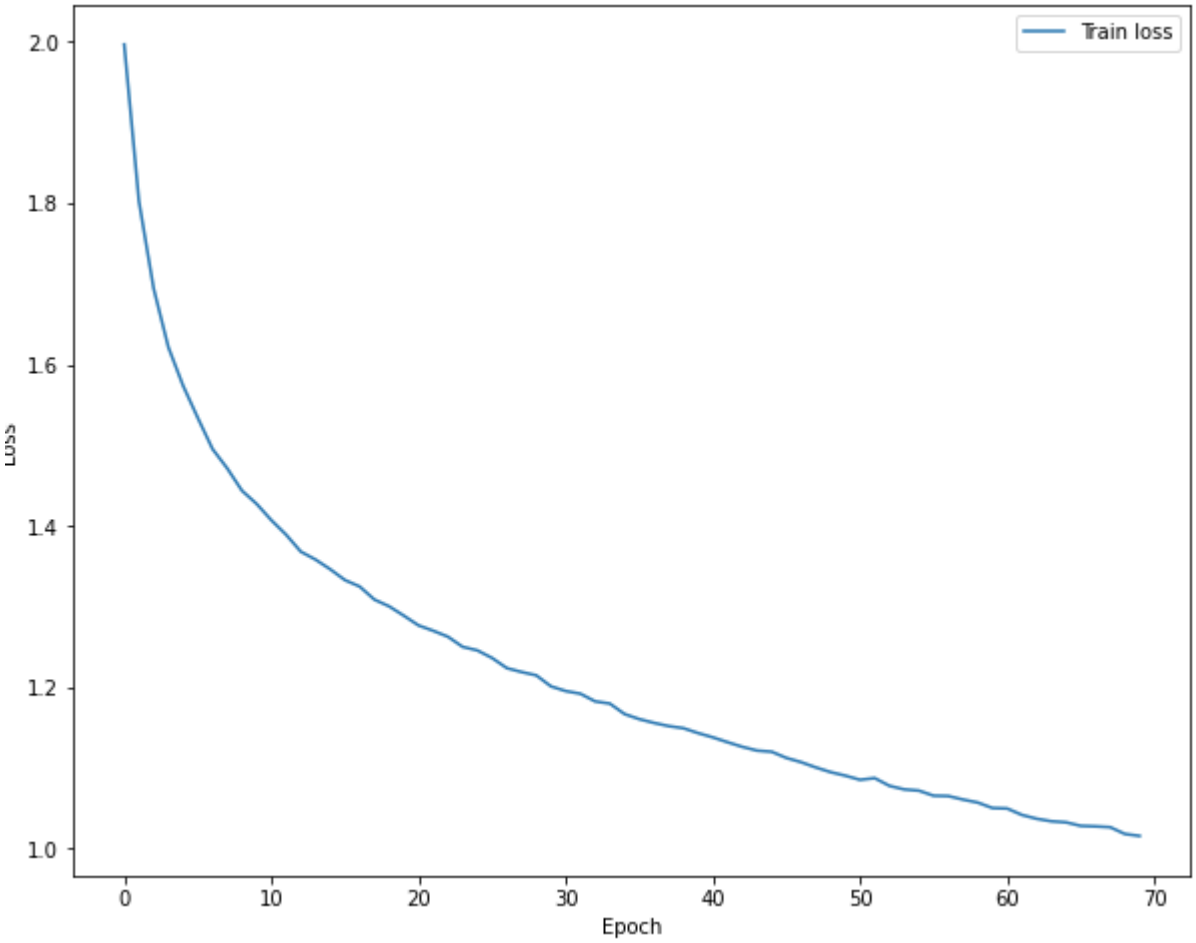
אחוז הדיוק שלנו לסט המבחן הוא 63%

סעיף 6

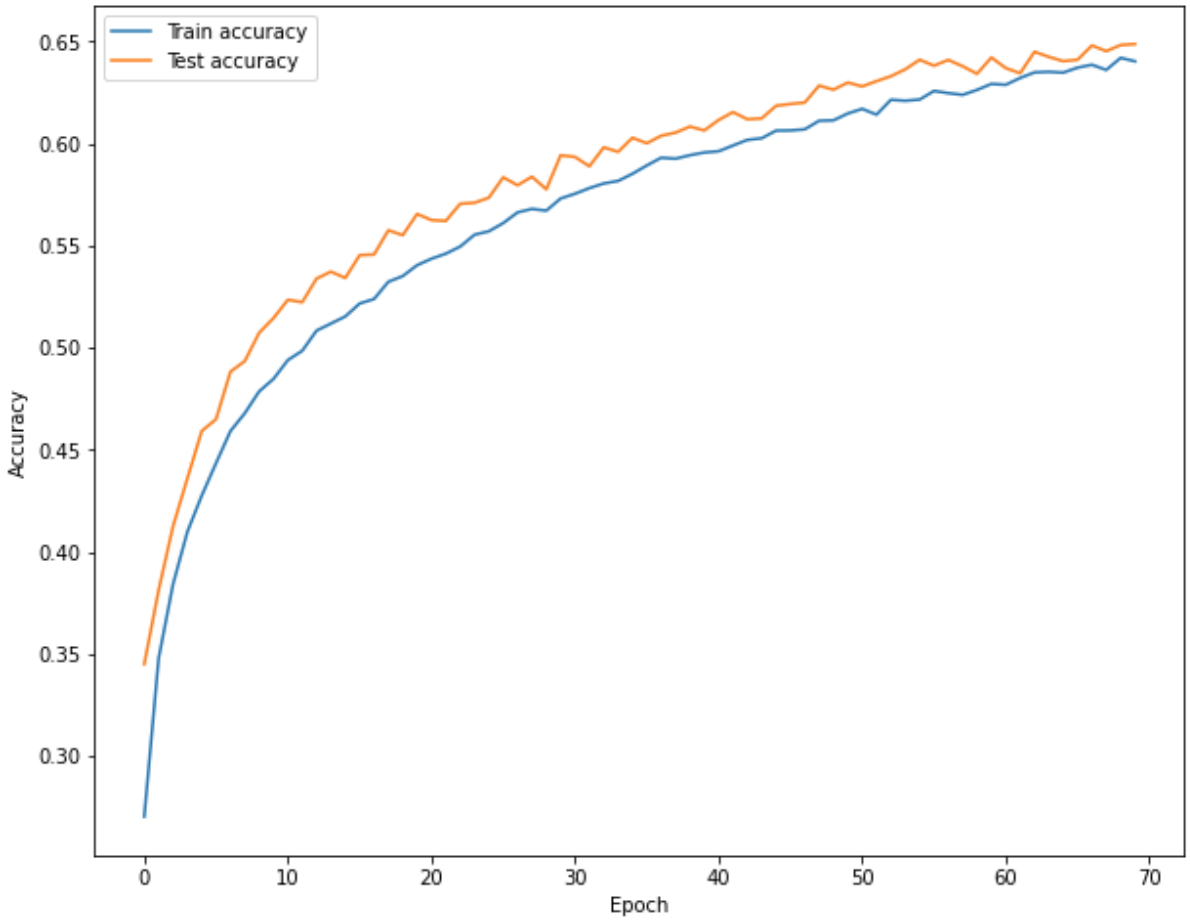
בסעיף זה הוספתי *augmentations* לסט האימון, והשתמשתי בתמונות שנוצרו על מנת לאמן מודל חדש.
מתוך כל השינויים שבדקתי, היה שילוב של כמה שהביא לשיפור בתוצאות.
האוגמנטציות שהשתמשתי בהם בסופו של דבר הן:

- *RandomHorizintalFlip* – בהסתברות של 0.2 מבצע היפוך של ציר האופקי של התמונה (תמונת מראה)
- *RandomResizedCrop* – בהסתברות של 0.2 מבצע זום לתמונה ברמות שונות וחותך מתוך הזום, תמונה בגודל המקורי.
- *RandomPerspective* – בהסתברות של 0.2 מבצע שינוי בפרספקטיבה של התמונה, כלומר בזוויות שלה ובגדלים.

לאחר ביצוע האוגמנטציות ביצענו אימון של הרשת עם פרמטרים דומים לסעיף הקודם.
גרף ההפסד בזמן האימון:

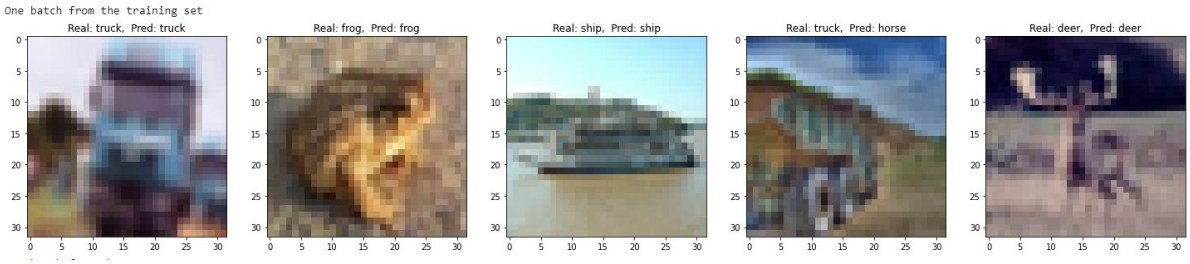


גרף הדיוק בזמן האימון:



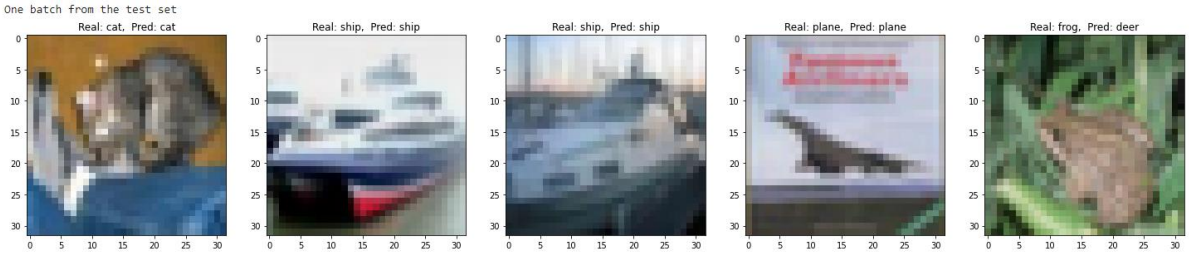
נשים לב, כי בעזרת האוגמנטציות הצלחנו להקטין את הoverfitting ולשפר את הדיוק הכולל של המודל.
בנוסף, ניתן לזהות תופעה מעניינת, בה המודל שלנו מקבל דיוק גבוה יותר על סט המבחן מאשר על סט האימון.
לדעתי זה נובע מכך שסט האימון משתנה בכל איפוק בגלל האוגמנטציות השונות שאנחנו מפעילים בצורה אקראית, לכן הוא בעצמו מתפקד כמו סוג של סט מבחן אשר המודל לא ראה קודם.

דוגמא להפעלת המודל שלנו על סט האימון והמבחן:
סט האימון:



ניתן לראות כי המודל טעה בתמונה אחת בלבד מסט האימון.

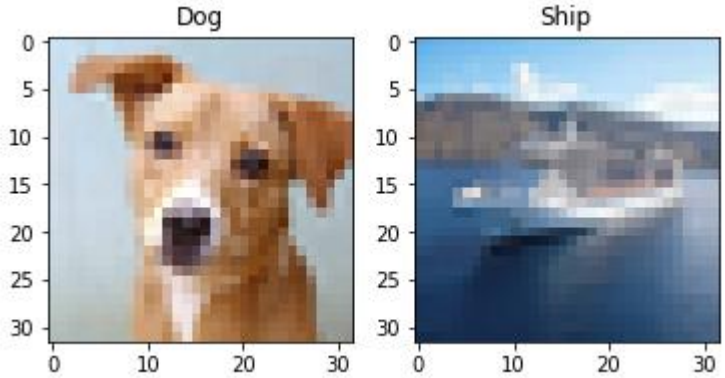
סט המבחן:



גם עבור סט המבחן, המודל טעה רק עבור תמונה אחת.
דיוק המודל עבור סט המבחן הינו 64.9% (יותר גבוה מהמודל הקודם)

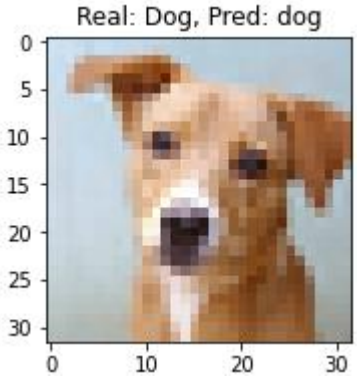
סעיף 7

כעת בחרתי 2 תמונות מהאינטרנט, אחת של כלב והשנייה של סירה:



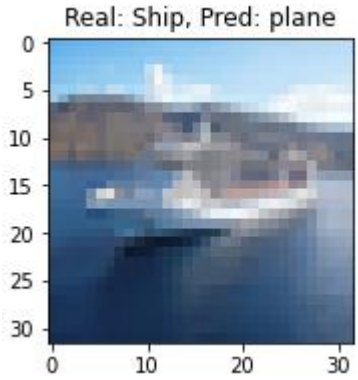
והפעלתי את המודל על כל אחת מהתמונות הנ"ל.
יש לציין, כי בחלק זה כאשר ביצעתי את ה*inference* לתמונות, הוספתי למודל את ה *softmax* על מנת לקבל את ההסתברויות, מפני שכעת אין שימוש בפונקציית ההפסד.

התוצאות והסתברויות הפלט של המודל:




```
[('dog', 0.7756), ('cat', 0.1425), ('plane', 0.0295), ('horse', 0.0283), ('bird', 0.0136), ('deer', 0.0057), ('frog', 0.0023), ('ship', 0.0015), ('truck', 0.0009), ('car', 0.0001)]
```

נשים לב כי עבור תמונה זו המודל שלנו צדק והוציא מידת ביטחון של 77.56%




```
[('plane', 0.6248), ('ship', 0.3658), ('bird', 0.0025), ('car', 0.0023), ('truck', 0.0021), ('deer', 0.0009), ('cat', 0.0006), ('horse', 0.0006), ('dog', 0.0003), ('frog', 0.0)]
```

כעת נשים לב, כי המודל שלנו טעה וחשב שהתמונה היא מטוס ולא סירה, אך אם נסתכל על ההסתברויות נראה, כי סירה קיבלה את ההסתברות השנייה הכי גדולה עם 36.58%.