ממ"ן 12 - ראייה ממוחשבת

אסף רזון

7 בינואר 2025

בתרגיל זה 3 חלקים.

שאלה 1

כצעד ראשון, התבקשנו לחלק את הנתונים לנתוני אימון train-set כאשר 60% מהתמונות ישמשו לאימון, 20% לוולידציה ו- 20% לבדיקה.

חלוקה של הדאטה למחיצות

החלוקה נעשתה תוך שימוש בפונקציית train_test_split מהספרייה, הקוד נמצא בקובץ split_train.py .
לאחר יצירת רשימת כל הקבצים, נעשית הגרלת חלוקה לשלוש המחיצות Train, Validation, Test לפי ההסתברויות שהתבקשנו. לאחר מכן הקבצים מופנים ל-3 ספריות בשמות המתאימים.

שיטת העבודה

ניתן לחלק את ה-pipeline לכמה שלבים, שכל אחד מהם קורא פלט מדיסק, מעבד אותו, ושומר לדיסק את השלב הבא.
כך ניתן לפתח כל שלב בנפרד ולחזור אחורה בלי להתחיל בכל פעם מההתחלה.
השלבים הם:

0. חלוקה של הדאטה (תמונות) למחיצות
1. הפעלה של Feature Extraction על כל התמונות ושמירת הפיצ'רים בלבד
2. פעולת קוונטיזציה (Vector Quantization) - אימון K-Means על מרחב הפיצ'רים שהתקבל בשלב הקודם ושמירת המודל. מספר המרכזים שבחרתי בשלב זה הוא 128.
3. שלב תרגום ע"י Bag Of Features - לכל תמונה המאופיינת כעת ע"י סט פיצ'רים, חישוב האשכול (Cluster) שאליו שייך כל פיצ'ר. כעת הייצוג של התמונה הופך להיות היסטוגרמה - לכל אשכול k מהו מספר המאפיינים ששייך לו.
   הייצוג הזה של תמונה כהיסטוגרמה נשמר לדיסק.
4. אימון מסווג אשר מקבל כקלט בשלב האימון זוגות של היסטוגרמה+מחלקה , ולומד לסווג היסטוגרמה למחלקה
5. הפעלת המסווג על מחיצות test, validation וחישוב המטריקות המתאימות להערכת הביצועים

נשים לב כי ה-pipeline הזה זהה לגמרי בין שאלות 1,2 - ההבדל העיקרי הוא איזה אלגוריתם מבצע את פעולת Feature Extraction .
בשאלה מס' 1 השתמשתי באלגוריתם Orb , אשר מחזיר פיצ'רים בגודל 32 - במספר שאינו ידוע מראש אך חסום.
בשאלה 2 השתמשתי בשכבות הראשונות של רשת נוירונים מסוג VGG16 ללא השכבות האחרונות שלה - המוצא של שכבות אלה הוא 64 פיצ'רים בגודל 512, שכל אחד מהם מתאים לסביבה של ריבוע אחד בתמונה המקורית ומתאר אותו ואת סביבתו (ברשת קלאסית בגודל 64x64 כל סביבה כזאת היא פיקסל).

נקודות חשובות בפתרון של שני הסעיפים האלה

- מספר הפיצ'רים שמחזירה רשת הנוירונים הוא קבוע. אבל מספר הפיצ'רים שמוחזרים ע"י אלגוריתם כגון Orb יכול להיות קטן יותר או גדול יותר מהמספר שהחלטנו לקבוע (כאן בחרנו לקבוע קבוצה בגודל 500).
  במקרה זה יש לחתוך או להשלים את הפיצ'רים למספר קבוע, כיוון שהיישום הבסיסי של היסטוגרמה כזו מניח מספר קבוע (וסכום ערכי ההיסטוגרמה יהיה שווה למספר הזה).
  לכן חייבים לבצע התאמה למספר קבוע של פיצ'רים.
  - אם המספר קטן מדי, משלימים את הקבוצה באמצעות אפסים (כלומר פיצ'רים שערך כולם הם אפס).
  - אם המספר גדול מדי, ממיינים את הפיצ'רים לפי ערך ה- Response ושומרים רק את בעלי הערכים הגבוהים עד לגודל הקבוצה הרצוי.
  - ערך זה מייצג "איכות" או "בטחון" של האלגוריתם בנקודת המפתח שהתגלתה. הנקודות שבהן ערך זה גבוה הן הבולטות יותר. לכן הגיוני להשתמש בנקודות שיש להן ערך תגובה גבוה (עד למספר הנדרש) ולהתעלם מן האחרות.
  - אם אכן יהיו הרבה "השלמות" של פיצ'רים ריקים (ערך 0) בסט האימון, סביר להניח שלפחות אחד מהאשכולות שיחושב באלגוריתם k-means יהיה קרוב מאד אל הנקודה של ראשית הצירים, כשאליה ימופו כל הקואורדינטות הריקות הנ"ל.
- המידע הטבלאי נשמר באמצעות Pandas DataFrame בפורמט feather המאופיין ביעילות קריאה גבוהה.
- המידע הבינארי, כגון סריאליזציה של מודלים, נשמר בפורמט pickle .
- לצורך בניית המסווג בשלב הסופי, ניסיתי מספר אפשרויות.
  - שתי וריאציות של מסווג AdaBoostClassifier
  - שתי וריאציות של מסווג XGBClassifier

הרצת השאלה

לצורך הרצת פתרון השאלה יש להריץ את הפקודה:

```
cd src
python mmn12_q1.py
```

במצב ברירת המחדל ללא פרמטרים, ההרצה תתחיל משלב 1 כפי שמפורט ב"שיטת העבודה". לחילופין ניתן להריץ החל משלב מתקדם יותר, למשל:

```
python mmn12_q1.py --stage=3
```

שאלות שנשאלנו

מה הפרמטרים האופטימליים של המסווג?
יש להציג את הביצועים ע"י עקומות ROC , חישוב ה-AUC וגם confusion matrix ו- precision-recall

תשובות לשאלות

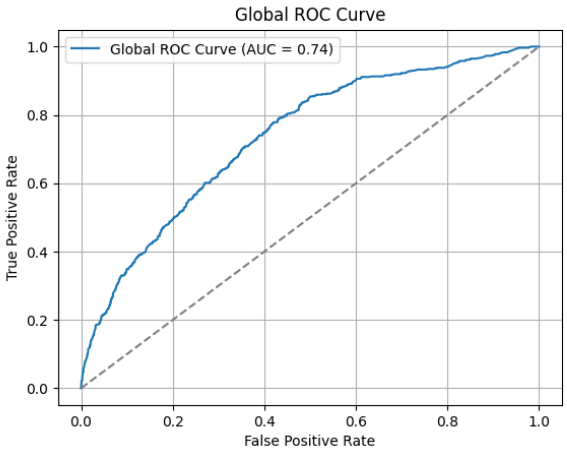
הטבלה: על-פי מדד ה-AUC , נראה שהמסווג המוצלח ביותר עבור Orb הוא ADABOOST2 . הפרמטרים שבהם השתמשתי עבורו הם:

```
'ADABOOST2':
AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=2),
    n_estimators=20,  # Number of boosting rounds
    learning_rate=0.2,  # Learning rate
    random_state=42  # Seed for reproducibility
)
```

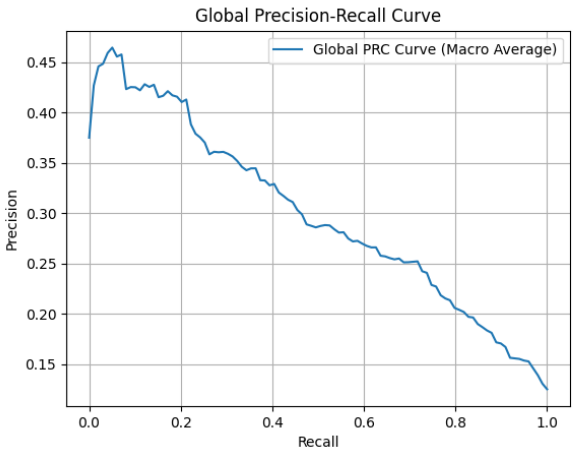
Table 1: ביצועי המסווגים השונים

extractor	classifier	accuracy	AUC
Orb	XGB1	0.3149	0.7394
Orb	XGB2	0.3467	0.7507
Orb	ADABOOST1	0.3099	0.7111
Orb	ADABOOST2	0.33	0.7359

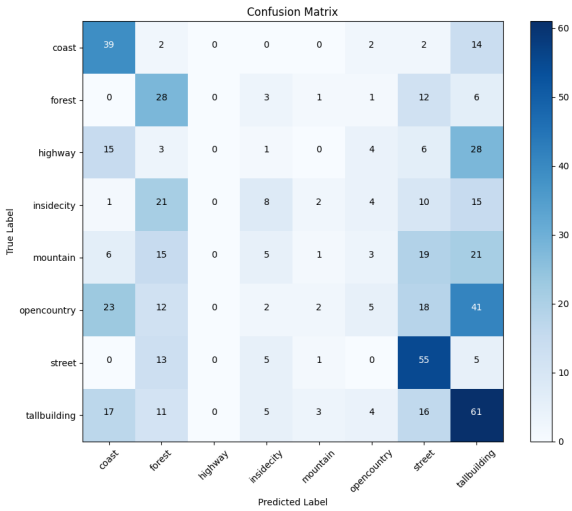
והגרפים המתאימים עבור מחיצת ה-Test הם:



איור 1: עקום ROC



איור 2: עקום Precision-Recall



איור 3: מטריצת הבלבול

שאלה 2

בשאלה 2 השתמשתי באותו ה- pipeline , עם הבדלים מעטים בלבד: שימוש בטנזורים (התוצר שיוצא מתוך הרשת), מימדים שונים במעט, וכו'.

גם כאן בשלב האחרון בדקתי את אותם מסווגים ואלה התוצאות שלהם:

טבלה 2: ביצועי המסווגים השונים

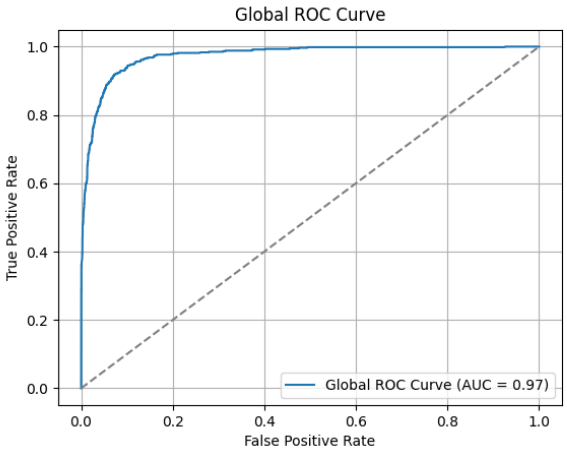
extractor	classifier	accuracy	AUC
VGG	XGB1	0.7554	0.968
VGG	XGB2	0.7856	0.9735
VGG	ADABOOST1	0.7873	0.9608
VGG	ADABOOST2	0.8074	0.9721

כל ארבעת המסווגים הגיעו לתוצאות דומות, אם כי כאן המוצלח ביותר היה XGB2 , אשר הפרמטרים שלו הם:

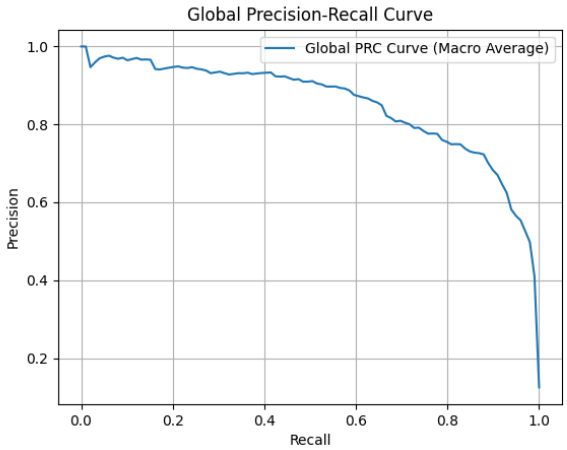
```
'XGB2':
XGBClassifier(
    objective='multi:softmax',  # Multiclass classification
    num_class=8,  # Number of classes
    max_depth=3,  # Tree depth
    learning_rate=0.03,  # Learning rate (eta)
    n_estimators=10,  # Number of trees
    random_state=42  # Seed for reproducibility
),
```

ואילו הגרפים שמתאימים לתוצאות שלו הם מופיעים כאן -

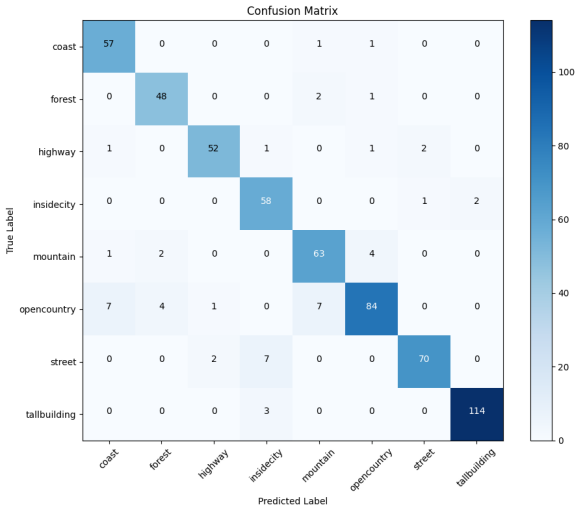
הביצועים על סט הבדיקה:



איור 4: עקום ROC



איור 5: עקום Precision-Recall



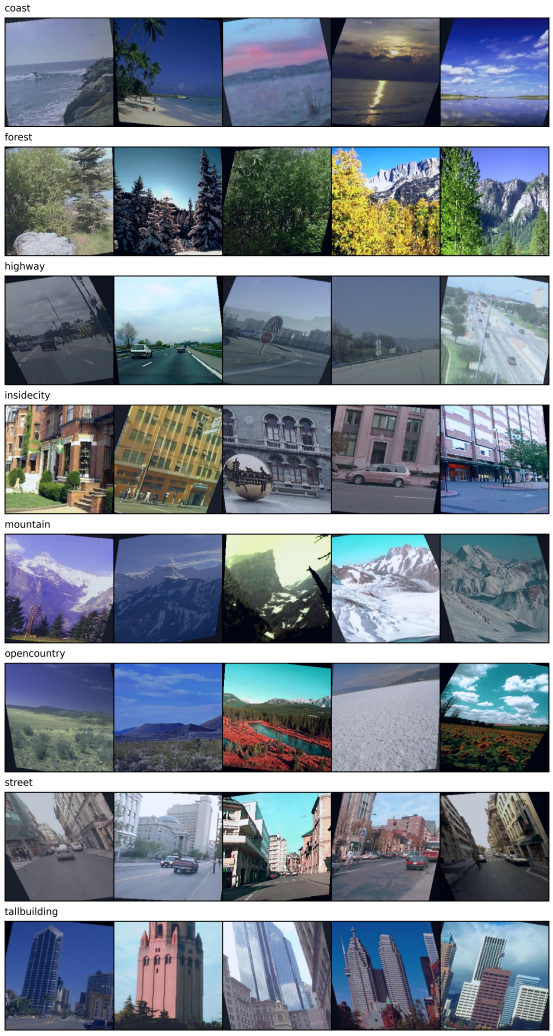
איור 6: מטריצת הבלבול

שאלה 3

בשאלה 3 התבקשנו לממש רשת נוירונים מסוג קונבולוציה ולאמן אותה כדי לסווג את הדאטה ל-8 המחלקות שלו. להרצה:

```
cd src
python mmn12_q3.py
```

כצעד ראשון התבקשנו להגדיר dataloader ולהציג מספר דוגמאות מכל מחלקה. תמונת הדוגמאות נוצרת בשלב מס' 1 של השאלה השלישית.



איור 7: דוגמאות מתוך סט האימון עבור כל אחת מ-8 המחלקות שאותן נדרשים לסווג

שיטת העבודה

על-מנת להגיע לתוצאות הנדרשות (90 אחוזי דיוק - הבנתי זאת בתור

החלטתי לבחור בטכניקה של Fine-Tuning על רשת קיימת. במצב כזה, הרשת שאומנה על כמויות גדולות של דאטה (תמונות) למדה לחלץ פיצ'רים רלוונטיים למשימה של זיהוי וסיווג תמונות. נשאר רק לבצע כוונון עדין יותר עבור המטרה הספציפית הזאת.

כדי לבצע זאת "מקפיאים" את השכבות הראשונות ברשת, כלומר מונעים מהן להתעדכן גם בשלב האימון (לא מתבצע עליהן Backpropagation וחישובי גראדיינטים). לעומת זאת השכבות האחרונות, בד"כ שכבת Fully Connected

, כן לומדות את הדאטה החדש ומתכוננות לפיו.

יתרונות נוספים של גישה זאת:

- הרשת גדולה ולכן יכולה להתמודד עם משימות מתוחכמות
- נשים לב שהדאטה שלנו קטן יחסית, ולכן לא מספיק לאימון מלא של רשת גדולה. באופן כללי מספר פרמטרים גדול דורש כמות גדולה יותר של מידע לאימון (אחרת הרשת תשנן את המידע)
- כיוון שאנחנו מאמנים רק שכבות אחרונות, מספר הפרמטרים המאומנים קטן יותר ולכן יכול להתאים לגודל דאטה קטן
- אם מספר הפרמטרים גדול מאד מגודל הדאטה, יש להקפיד על טכניקות להתמודדות עם אפשרות של Overfit - כגון שימוש ברגולריזציה, Dropout , ואוגמנטציה.
- השימוש באוגמנטציה מאפשר גם אימון ליכולת הפשטה גבוהה יותר וגם כסוג של הגדלה מלאכותית של הדאטה.

נקודות וטכניקות שבהן השתמשתי על-מנת לשפר את ביצועי הרשת:

- בחירת רשת וטכניקה של כוונון עדין
- בחירת מספר אפוקים גדול מצד אחד כדי לתת לרשת אפשרות ללמוד לאורך יותר איטרציות, אבל גם עצירה מוקדמת כדי למנוע ממנה להיתקע.
- בחירה נכונה של אלגוריתם Optimizer . שימוש ב-SGD סטנדרטי לא הביא לתוצאות טובות. לעומת זאת החלפתו באלגוריתם ממשפחת (Adam, AdamW) שיפרה בהרבה את התכנסות הרשת והתוצאות הסופיות אליהן הגיעה.
- בחירה נכונה של פרמטר Learning Rate . בניגוד לאינטואיציה הראשונית, קצב לימוד גבוה מדי לא הביא להתכנסות מהירה יותר אלא להיפך, התכנסות איטית לתוצאות לא מוצלחות. דווקא הקטנה שלו הביאה להתכנסות אל תוצאות טובות יותר.
- שינוי של גודל האצווה (batch) , הקטנתו מאטה את החישובים אבל משפרת את ההתכנסות - בגלל
- שינוי דינאמי של קצב הלימוד ע"י שימוש ב- Learning Rate Scheduler . ה-scheduler מאפשר להתחיל בקצב לימוד מסויים, ולהקטין אותו שוב ושוב אם הרשת לא מראה שיפור בביצועים. הדבר מאפשר לנצל קצב לימוד גבוה בהתחלה כדי לבצע התכנסות ראשונית, ולאחר מכן שיפור מקומי (שיכול להיות משמעותי מאד) בקצבים נמוכים יותר.
- עצירה מוקדמת (Early Stopping) - הפסקת אימון הרשת אם התוצאות מדגימות שהיא אינה משתפרת יותר, או שהשיפור נעשרה רק על סט האימון, כלומר הרשת מתחילה לבצע Overfitting ואין טעם להמשיך באימון.
- שימוש באוגמנטציות. עם זאת, אוגמנטציות אגרסיביות מדי פגעו מאד בביצועי הרשת - כנראה מפני שבכל זאת כמות הדאטה (בעיקר מחיצת הולידציה) קטנה מדי והרשת מבצעת overfit על המידע. למשל פעולת GaussianBlur שאמורה לסייע במקרים כלליים, רק הפריעה על הדאטה הקטן הספציפי שלנו.

הפרמטרים של Learning Scheduler שנבחרו הם כנ"ל:

```python
# Define the scheduler
scheduler = ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.5,
    patience=3,
    threshold=1e-5,
    cooldown=2,
    min_lr=1e-6,
)

# Use in the code:

# Step the scheduler with validation loss
scheduler.step(val_precision)

# Print the current learning rate
for param_group in optimizer.param_groups:
    print(f" Learning Rate: {param_group['lr']}")
```

דוגמה לפלט של הרשת בשלב סיום האימון:

```
 Learning Rate: 0.00025

Epoch 8/50
 Training   : 100%|██████████| 214/214 [00:02<00:00, 91.56it/s]
 Validation: 100%|██████████| 76/76 [00:00<00:00, 125.22it/s]
 Training Loss:      0.2174, Validation Loss:      0.2102
 Training Precision: 0.9239, Validation Precision: 0.9287
 Learning Rate: 0.00025
Early stopping triggered. Patience: 2
Metrics saved to ../report/data/RESNET18_5LAYERS.csv
Loss plot saved to ../report/images/RESNET18_5LAYERS.png
-------------------
Final on best model:
 Validation: 100%|██████████| 76/76 [00:00<00:00, 123.19it/s]
 Test       : 100%|██████████| 75/75 [00:00<00:00, 119.60it/s]
 Train      : 100%|██████████| 214/214 [00:01<00:00, 122.16it/s]
Validation Loss:           0.2067, Train Loss     : 0.0454
Validation precision:      0.9270, Train Precision: 0.9906
Test       precision:      0.9146
End
```

בעמוד הבא ניתן לראות עבור שתי רשתות שנבדקו, מהו מבנה הרשת - כולל אילו שכבות נשארו לא מוקפאות ומה מספר הפרמטרים שלהן.

התוצאות הסופיות שנבחרו הן מהרשת RESNET18_5LAYERS , כלומר רשת המבוססת על המשקלות של Resnet18 , אבל עם חמש שכבות אחרונות שניתנות לאימון. שכבות אלה שקולות לבלוק הבסיסי בשכבה הסדרתית האחרונה ולאחריה החלק הלינארי.

בגרסת RESNET18_2LAYERS , רק השכבה הלינארית האחרונה היא זו שמתאמנת.

Model: RESNET18_2LAYERS
Trainable parameters: ['fc.1.weight', 'fc.1.bias']
==========

```
==========================================================================================
Layer (type:depth-idx)            Output Shape              Param #     Trainable
==========================================================================================
ResNet                            [1, 8]             --                  Partial
├─Conv2d: 1-1                     [1, 64, 128, 128]         (9,408)     False
├─BatchNorm2d: 1-2                [1, 64, 128, 128]         (128)       False
├─ReLU: 1-3                       [1, 64, 128, 128]         --          --
├─MaxPool2d: 1-4                  [1, 64, 64, 64]           --          --
├─Sequential: 1-5                 [1, 64, 64, 64]           --          False
|    └─BasicBlock: 2-1            [1, 64, 64, 64]           (73,984)    False
|    └─BasicBlock: 2-2            [1, 64, 64, 64]           (73,984)    False
├─Sequential: 1-6                 [1, 128, 32, 32]          --          False
|    └─BasicBlock: 2-3            [1, 128, 32, 32]          (230,144)   False
|    └─BasicBlock: 2-4            [1, 128, 32, 32]          (295,424)   False
├─Sequential: 1-7                 [1, 256, 16, 16]          --          False
|    └─BasicBlock: 2-5            [1, 256, 16, 16]          (919,040)   False
|    └─BasicBlock: 2-6            [1, 256, 16, 16]          (1,180,672) False
├─Sequential: 1-8                 [1, 512, 8, 8]            --          False
|    └─BasicBlock: 2-7            [1, 512, 8, 8]            (3,673,088) False
|    └─BasicBlock: 2-8            [1, 512, 8, 8]            (4,720,640) False
├─AdaptiveAvgPool2d: 1-9          [1, 512, 1, 1]            --          --
├─Sequential: 1-10                [1, 8]                    --          True
|    └─Dropout: 2-9               [1, 512]                  --           --
|    └─Linear: 2-10               [1, 8]                    4,104       True
==========================================================================================
Total params: 11,180,616
Trainable params: 4,104
Non-trainable params: 11,176,512
Total mult-adds (Units.GIGABYTES): 2.37
==============================================================================
Forward/backward pass size (MB): 51.90
Params size (MB): 44.72
Estimated Total Size (MB): 97.41
==============================================================================
```

Model - RESNET18_5LAYERS
Trainable parameters:
['layer4.1.conv2.weight', 'layer4.1.bn2.weight', 'layer4.1.bn2.bias', 'fc.1.weight', 'fc.1.bias']

```
==========================================================================================
Layer (type:depth-idx)           Output Shape              Param #           Trainable
==========================================================================================
ResNet                           [1, 8]                    --                Partial
├─Conv2d: 1-1                    [1, 64, 128, 128]         (9,408)           False
├─BatchNorm2d: 1-2               [1, 64, 128, 128]         (128)             False
├─ReLU: 1-3                      [1, 64, 128, 128]         --                --
├─MaxPool2d: 1-4                 [1, 64, 64, 64]           --                --
├─Sequential: 1-5                [1, 64, 64, 64]           --                False
|    └─BasicBlock: 2-1           [1, 64, 64, 64]           (73,984)          False
|    └─BasicBlock: 2-2           [1, 64, 64, 64]           (73,984)          False
├─Sequential: 1-6                [1, 128, 32, 32]          --                False
|    └─BasicBlock: 2-3           [1, 128, 32, 32]          (230,144)         False
|    └─BasicBlock: 2-4           [1, 128, 32, 32]          (295,424)         False
├─Sequential: 1-7                [1, 256, 16, 16]          --                False
|    └─BasicBlock: 2-5           [1, 256, 16, 16]          (919,040)         False
|    └─BasicBlock: 2-6           [1, 256, 16, 16]          (1,180,672)       False
├─Sequential: 1-8                [1, 512, 8, 8]            --                Partial
|    └─BasicBlock: 2-7           [1, 512, 8, 8]            (3,673,088)       False
|    └─BasicBlock: 2-8           [1, 512, 8, 8]            4,720,640         Partial
├─AdaptiveAvgPool2d: 1-9         [1, 512, 1, 1]            --                --
├─Sequential: 1-10               [1, 8]                    --                True
|    └─Dropout: 2-9              [1, 512]                  --                --
|    └─Linear: 2-10              [1, 8]                    4,104             True
==========================================================================
Total params: 11,180,616
Trainable params: 2,364,424
Non-trainable params: 8,816,192
Total mult-adds (Units.GIGABYTES): 2.37
=======================================================================
Input size (MB): 0.79
Forward/backward pass size (MB): 51.90
Params size (MB): 44.72
Estimated Total Size (MB): 97.41
==============================================================================
```

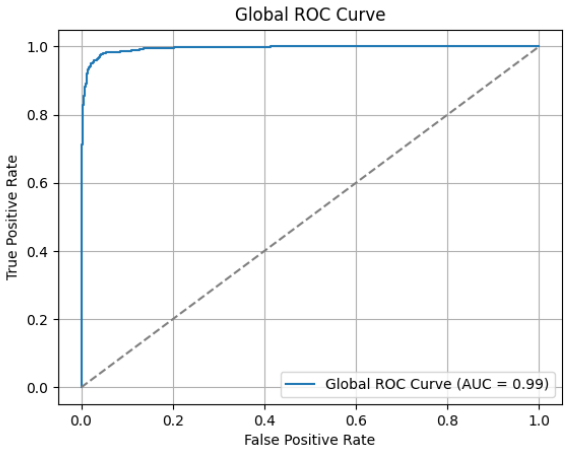
הציגו גרף של ערכי ה- loss על סט הוולידציה והאימון עבור מקרה של over-fitting .

במקרה הזה יש over-fitting משילוב של הסיבות הבאות:

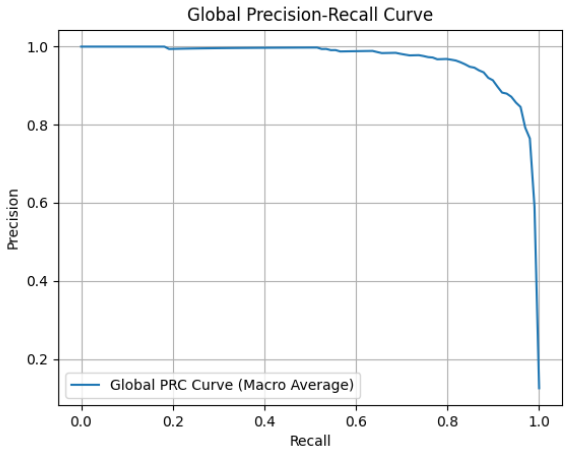
- רשת פשוטה מדי (VGG16)
- אוגמנטציות עדינות מאד
- תהליך אימון ממושך מדי ללא Early Stopping

איור 8: דוגמה להתקדמות Train/Validation לאורך תהליך האימון

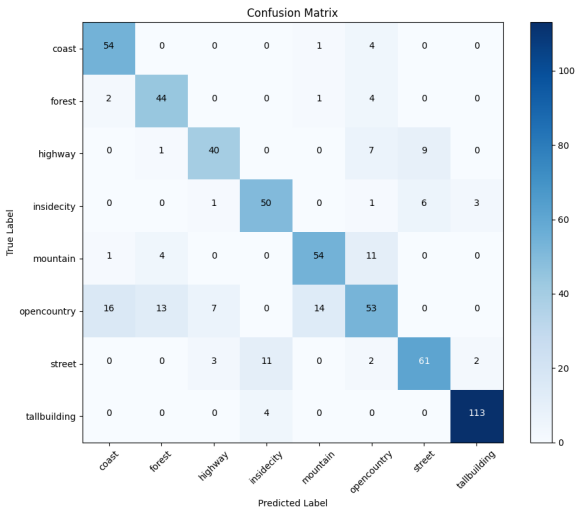
חשבו את הביצועים על סט הבדיקה. יש להציג את הדיוק, גרף של ROC וגרף של Precision-Recall .



איור 9: עקום ROC



איור 10: עקום Precision-Recall



איור 11: מטריצת הבלבול

איור 12: דוגמה להתקדמות Train/Validation לאורך תהליך האימון: