**Classification עם pytorch**

בפוסט זה נערוך הכרות עם pytorch.

את ההכרות נעשה על ידי בניית מודל שיבצע classification על תמונות.

אז מה נראה:

1. התקנה
2. רקע קצר על pytorch
3. בניית מודל שיזהה ספרות (0-9) בתמונות של 28X28
4. אימון המודל
5. בחינת המודל
6. **התקנת pytorch:**
7. נכנסים לאתר <https://pytorch.org/>
8. בוחרים את הנתונים המתאימים כאשר כמובן נעבוד על python ועם הגרסה העדכנית ביותר של הספרייה.
9. מעתיקים את שורת הפקודה שנוצרה ומריצים.
10. בנוסף צריך להתקין torchvision: <https://pypi.org/project/torchvision/>

1. **אז מהי pytorch ?**

בעולם למידת המכונה ([machine learning](https://en.wikipedia.org/wiki/Machine_learning" \o ")) ישנן מגוון ספריות שמאפשרות לנו המשתמשים לבנות ולאמן מודלים למטרות מגוונות. מתוך ספריות אלו, הספריות הנפוצות ביותר הנן Tensorflow מבית Google, וPytorch מבית facebook. שתי הספריות הן open source.

כמו שהשם מרמז, Pytorch מיועדת לרוץ על python. זה מאפשר קליטה יחסית מהירה של הספרייה למי שמכיר פייתון, ומאפשר דיבוג נוח. tensorflow איננה כזאת, היא נבנתה תלאי על תלאי ועברה התאמות עבור python, ועל אף שבtensorflow2 יש כבר אפשרות ל [eager execution](https://www.tensorflow.org/guide/eager)עדיין יש בה סרבול מסויים.

**Tensor:**

מבנה הנתונים העיקרי בpytorch (וגם בtensorflow) נקרא Tensor. במובן הפרקטי לספרייה, Tensor הוא מבנה נתונים שמייצג מטריצה רב-ממדית שמכילה איברים מאותו data type. למה הכוונה?

נביט בדוגמא הבאה:

import torch

t = torch.tensor([[1, 2, 3],   
 [4, 5, 6]], dtype=torch.int8)

בדוגמא הנ"ל אנחנו מייצרים טנסור מדרגה 2 (שהוא בעצם מטריצה), והdata type הוא int8.

ניתן לייצר טנסור מכל דרגה כמובן, כל עוד נשמר יחס בין כל האיברים בכל דרגה ושכל האלמנטים מאותו הdata type. למשל הדוגמא הבאה:

t = torch.tensor([[1, 2, 301.],   
 [4, 5]], dtype=torch.int8)

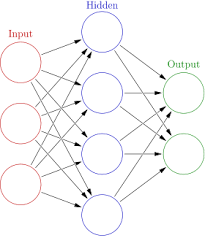
אינה תקינה כי יש לנו בדרגה הראשונה שתי רשימות שאינן באותו האורך. בנוסף תתבצע במקרה זה המרה של 3.01 שהוא float16 ל3 שהוא int8. לכן כדאי לשים לב שעובדים עם data type נכון

t = torch.tensor([1, 2, 3], dtype=torch.int8)  
t = t / 0.3

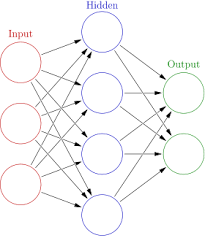
בדוגמא הנ"ל התחלנו עם טנסור מסוג int8, חילקנו אותו ב0.3 (float) אז הפעולה תתבצע והספרייה תשנה את הdata type לבד, לפי התוצאות שהתקבלו.

הסיבה לשימוש בtensor ולא בlist של פייתון למשל היא כי הlist של פייתון איטי הרבה יותר. בגלל שהdata type בtensor זהה לכל האלמנטים, ניתן לשמור את הtensor ב'גוש אחד' בזיכרון, ובצורה יעילה. באופן כללי tensor מאוד דומה לarray של numpy עם תוספות.

**אז למה טנסורים בעצם?**

רשתות נוירונים מוצגות בדרך כלל באופן הבא:

כאשר נרצה ליישם ייצוג כזה במחשב, מאוד נוח לעבוד עם טנסורים.

**איך זה בא לידי ביטוי**?

נביט על השכבה הבאה (linear layer):

הinput שלה באורך 3, והoutput שלה באורך 4.

נוכל לייצג את החישובים בשכבה באופן הבא:

x = וקטור הInput באורך 3

W = מטריצת משקלים מסדר (4,3) – המטריצה בצורה הזאת בגלל ההתנהגות של כפל מטריצות

b = bias, וקטור באורך 4 (שאינו מפויע בתמונה)

אם כן פעולת השכבה תהיה:

Wx + b (ועל זה נפעיל אקטיביזציה אבל על זה בהמשך)

ישנן עוד מגוון דוגמאות ושימושים ואת חלקם נראה בהמשך.

1. **בניית המודל:**

אז אחרי כל ההכנות, בואו ניגש לקוד:

נתחיל בלייבא את החבילות הבאות:

import torch  
import torchvision  
import torch.nn as nn  
import torch.nn.functional as F  
import torch.optim as optim  
import torchvision.transforms as transforms  
from torch.utils.data import DataLoader  
from tqdm import tqdm  
from time import time, sleep

import matplotlib.pyplot as plt

nn :

(קיצור של neural network) מכיל מספר רב של פעולות שימושיות כמו activation functions, conv layers, functional, Module, parallel..

nn.Functional:

מכיל מספר רב של פעולות שימושיות כמו activation functions, conv layers..

לקריאה נוספת - https://pytorch.org/docs/stable/nn.functional.html

ישנה חפיפה לא מעטה בין nn ל nn.Functional. באופן כללי, ניתן להשתמש בכל אחד מהם והתוצאה תהיה דומה. כאשר ניתן לממש משהו באמצעות שניהם, הקונבנציה היא להשתמש בפונקציות בnn כאשר מדובר ב layers עם learnable parameters ובfunctional כאשר אין learnable parameters (כמו Relu למשל)

:optim

חבילה (package) המכילה בתוכה מגוון רחב של אלגורתמי אופטימיזציה כמו SGD, Adam ורוב המוכרים.

בנוסף יש בה schedulers – אלגוריתמים שמאפשרים להתאים learning rate בכל כמה צעדים.

זה שימושי כי בדרך כלל אנחנו צריכים 'צעדים גדולים' בשביל להגיע למטרה, אבל ככל שמתקרבים נעדיף צעדים עדינים יותר כדי שלא נפספס את המטרה.

ישנו פוסט המפרט על אלגוריתמי אופטימיזציה פה:

<https://www.ai-blog.co.il/2020/05/04/%d7%a1%d7%99%d7%9b%d7%95%d7%9d-%d7%a9%d7%9c-%d7%a1%d7%95%d7%92%d7%99-%d7%90%d7%95%d7%a4%d7%98%d7%99%d7%9e%d7%99%d7%96%d7%a6%d7%99%d7%95%d7%aa-%d7%91%d7%9c%d7%9e%d7%99%d7%93%d7%94-%d7%a2%d7%9e%d7%95/>

**הכנת המידע:**

נרצה להכין\לייבא dataset עליו נאמן את המודל. שלב איסוף הנתונים הוא השלב הקשה בדרך כלל, והיקר יותר. אנחנו נעבוד בשלב ראשון עם datasetקיים ושמו Digit MNIST.

Digit MNISTהוא מאגר ובו 60,000 תמונות 28x28של ספרות 0-9. בנוסף לכל תמונה יש תבית (label) בה רשום איזו ספרה רשומה בתמונה.

ראשית נגדיר טרנספורמציה:

transform = transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize((0.5,), (0.5,))  
])

מהי טרספורמציה ולמה היא נחוצה?

המידע שאנו עובדים איתו יכול להגיע בהרבה צורות. למשל מאגר תמונות בפורמט jpg בגדלים שונים ..

על מנת שפורמט המידע 'יתאים' למודל אותו אנחנו בונים, אנו מבצעים טרספורמציות.

הביטוי 'יתאים' הוא די רחב כיוון שהוא מכליל בתוכו מגוון של פעולות שנוגעות בהיבטים שונים. בעבור תמונות למשל נוכל להפוך את תמונות לפורמט PIL, להקטין, להגדיל, לבצע נורמליזציה..

שימוש יעיל נוסף הוא עבור augmentation data – כאשר אין לנו מספיק תמונות, או שהתמונות אינן מגוונות מספיק נוכל לבצע טרנספורמציות שיבצעו חיתוך של תמונות, סיבוב וכו' וזה יגדיל את המאגר.

במקרה שלנו, לא נזדקק להרבה עבודה כיוון שmnist עשו את רוב העבודה, לכן רק נהפוך את התמונה\תמונות לtensor ואז ננרמל.

נטען את הדאטא:

train\_set = torchvision.datasets.MNIST('/files/',   
 train=True,   
 download=True,   
 transform=transform)

נשים לב שהעברנו ל Datasetאת הtransforms שיצרנו קודם.

בדרך כלל במימוש נכון הDataset עוד לא מפעיל את הטרנפורמציות ואפילו לא טוען את התמונות.

רק ברגע שנבקש ממנו איבר כלשהו באמצעות המתודה:

def \_\_getitem\_\_(self, idx):

הוא יטען את התמונה ויבצע עליה את הטרנספורמציות.

זוהי נקודה חשובה שמקבלת משנה חשיבות כאשר מדובר בbigdata כי לא נוכל לטעון את כל הDataset בפעם אחת.

נאתחל את הDataLoader:

train\_loader = torch.utils.data.DataLoader(  
 dataset=train\_set,  
 batch\_size=64,  
 shuffle=True)

בDataLoader טוענים את הdataset, מגדירים batch\_size, והאם לערבב או לא.

תפקיד הDataLoader הוא לערום את הנתונים בbatches כך שיהיו מוכנים לאימון.

נביט על צורת העבודה עם הDataLoader:

images, labels = next(iter(train\_loader))

לאחר שורה זו, imaegs יהיה טנסור מהצורה הבאה (28, 28, 1, 64)

נפרש זאת כ64 תמונות, לכל אחת מהן יש ערוץ אחד, וגודל כל אחת מהן הוא 28x28.

אם היינו עובדים עם תמונות באותו גודל רק צבעוניות (כלומר שלושה ערוצים) אז היינו מקבלים

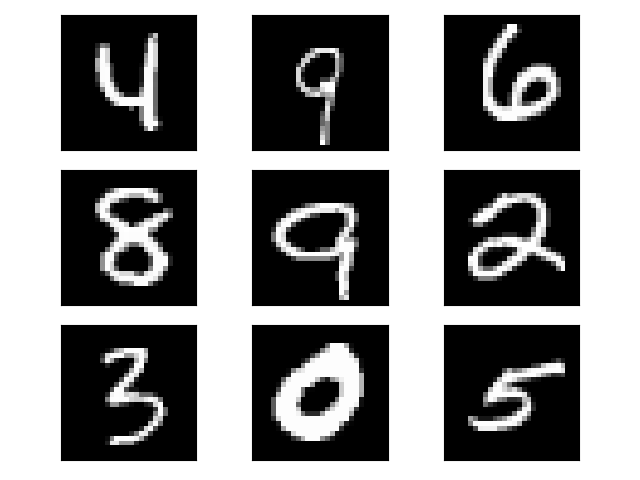
(28, 28, **3**, 64).

מוזמנים להריץ את השורות הבאות ולראות מהו labels ומהי צורתו:

print("the shape is ", labels.shape)  
print(labels)

קטע הקוד הבא יציג דגימה קטנה מן התמונות בbatch, רק כדי שנקבל תחושה:

fig = plt.figure()  
for i in range(9):  
 plt.subplot(3, 3, i + 1)  
 plt.tight\_layout()  
 plt.imshow(images[i][0], cmap='gray')  
 plt.xticks([])  
 plt.yticks([])  
  
fig.show()

הoutput אמור להראות כך (לאו דווקא עם אותן ספרות):

נוסיף את השורה:

device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

שתעזור לנו אח"כ להריץ את החישובים על ה- gpu, אם קיים.

נגדיר:

class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=20,  
 kernel\_size=)5, 5), stride=1)  
  
 self.conv2 = nn.Conv2d(in\_channels=20, out\_channels=50,  
 kernel\_size=5, stride=1)  
  
 self.fc1 = nn.Linear(in\_features=50\*4\*4,out\_features=500)  
 self.fc2 = nn.Linear(in\_features=500, out\_features=10)  
  
 def forward(self, t):  
 t = self.conv1(t)  
 t = F.relu(t)  
 t = F.max\_pool2d(t, kernel\_size=2, stride=2)  
  
 t = self.conv2(t)  
 t = F.relu(t)  
 t = F.max\_pool2d(t, kernel\_size=2, stride=2)

t = t.view(t.size(0), -1)  
 t = self.fc1(t)  
 t = F.relu(t)  
 t = self.fc2(t)  
 return t

בpytorch מימוש רשת נעשה על ידי ירושה של nn.Module.

בפונקציית הinit נאתחל את השכבות. בהמשך, כאשר נכניס batch של תמונות לרשת, הרשת תקרא למתודה forwardותשלח לה כארגומנט את הbatch בצורת tensor.

הרשת שהרכבנו בנויה משני חלקים עיקריים – קונבולוציה וlinear:

הרשת מתחילה בשתי שכבות קונבולוציה nn.Conv2d עם הפרמטרים הבאים:

1. in\_channels: מספר הערוצים הנכנסים. בשכבה הראשונה הארגומנט יהיה 1, כי במקרה שלנו אנחנו עובדים רק עם צבע אחד. אם היינו עובדים עם צבעוני אזי זה היה צריך להיות 3.
2. Out\_channels: מייצג את כמות הפילטרים שיעבדו על כל תמונה. לכן זהו מספר תתי התמונות השונות שתייצר כל תמונה.
3. Kernel\_size: גודל הפילטר. ניתן לציין גם כמספר בודד והוא ישלים לנו לפי מספר הממדים של התמונה.
4. Stride:כמות הצעדים שהפילטר יתקדם בכל קפיצה (אנכית ואופקית)

בין כל שכבת קונבולוציה, נפעיל על הoutput פונקציית אקטיביזציה: Relu.

לאחר שנסיים להריץ את הכל מוזמנים לנסות גם:

F.sigmoid()  
F.tanh()  
F.leaky\_relu()

בנוסף לפונקציית אקטיביזציה נבצע max\_pool2d שבמקרה שלנו ילך עם ריבוע 2x2 בקפיצות של שתיים, ויבחר את הערך הכי גדול בכל ריבוע שכזה. (כלומר יחצה את אורך ורוחב התמונה בשתיים)

משסיימנו את שכבות הקונבולוציה, נגיע לשתי שכבות linear/fully-connected/dense. משלב זה השכבות צריכות את הנתונים משוטחים, כלומר במימד אחד. ולכן נבצע:

t = t.view(t.size(0), -1)

המתודה view הופכת את t להיות מהצורה (1-batch\_size, ) כאשר 1- אומר לפונקציה להשלים לבד איזו צורה צריכה להיות. הסיבה למעבר הזה היא שאנחנו עובדים בbatches ולא בתמונה יחידה, ואיננו רוצים וקטור ארוך של כל התמונות יחד. לכן עברנו לצורה שמשטחת כל תמונה לוקטור אך שומרת על כל וקטור בנפרד.

איך הגענו ל 4\*4\*50?

התחלנו מ28x28

השכבה הראשונה הורידה ל24X24

Max\_pool2d הוריד ל 12X12

השכבה השניה הורידה ל 8X8

ואז שוב Max\_pool2d הוריד ל4X4

1. **אימון המודל:**

נאתחל את הרשת, האופטימייזר ופונקצית הloss, ונביא את הרשת לcpu/gpu:

net = Net().to(device)

optimizer = optim.SGD(params=net.parameters(), lr=0.001, momentum=0.9)

criterion = nn.CrossEntropyLoss()

**optimizer**: נשתמש בSGD. ניתן למצוא פירוט ואופציות נוספות כאן:

<https://pytorch.org/docs/stable/optim.html>.

params: המשקלים וbiases של הרשת אותם האופטימייזר ישפר במהלך הלמידה.

lr - learning rate : קצב הלימוד שיוכפל בגרדיאנט. שימו לב שהוא יחסית רגיש וכדאי לקבוע את הטווח שלו בדרך כלל בין 0.01 – 0.0001

momentum: עוזר לSGD להמשיך להאיץ או לשמור על תאוצה על ידי הכפלת הערך בוקטור העדכון מהסבב הקודם והוספה לוקטור הנוכחי. על פי Andrew Ng בדרך כדאי שינוע בין 0.9 – 0.99

**Criterion:**

במשימות קלסיפיקציה שבהן צריך לסווג תמונה למחלקה אחת יחידה, נשתמש בCrossEnthropyLoss.

CrossEnthropyLoss תקח את וקטור הoutput של הרשת, תהפוך את הפרמטרים בו להתפלגות (אותו רעיון של softmax רק עם log) ואז תחשב טעות ביחס להתפלגות וקטור המטרה (ground truth).

אם למשל הכנסנו תמונה עם הספרה 6, CrossEnthropyLoss תתן דגש לפער בין המקומות ה6 בשני הוקטורים. כלומר הפונקציה מנסה בעיקר לחזק לכיוון התשובה הנכונה ולא להחליש תשובות שגויות (על אף שגם המישור הזה יישתפר במהלך הדרך).

נקודה חשובה היא שהפונקציה משלבת בתוכה גם LogSoftmax כך שאין צורך להוסיף Softmax בסוף הרשת.

פירוט על אפשרויות נוספות: <https://pytorch.org/docs/stable/nn.html#loss-functions>

נגדיר פונקציית אימון:

def train(net, epoch\_num):  
 start = time()  
 for epoch in range(epoch\_num):  
 running\_loss = 0.  
 accuracy = 0.  
 msg = 'training: epoch {}'.format(epoch)  
 for images, labels in tqdm(train\_loader, desc=msg):  
 images = images.to(device, non\_blocking=True)  
 labels = labels.to(device, non\_blocking=True)

optimizer.zero\_grad()  
  
 preds = net(images)  
  
 loss = criterion(preds, labels)  
  
 loss.backward()  
  
 optimizer.step()  
  
 running\_loss += loss.item()  
  
 accuracy += preds.argmax(dim=1).eq(labels).sum().item()  
  
 print('results: Epoch {}, accuracy {}, loss:{}\n'.format(  
 epoch,  
 (accuracy / len(train\_loader.dataset)) \* 100,  
 (running\_loss / len(train\_loader))), flush=True)  
 sleep(.5)  
  
 end = time()  
 print('Done Training')  
 print('%0.2f minutes' % ((end - start) / 60))

כדאי להעתיק את הקוד ולהריץ כדי לחזות בתוצאה. זמן הריצה יהיה באזור ה10 דק' אם עובדים על cpu ממוצע. בסיום, הaccuracy צריך להיות 98+, והloss קטן מ0.1

השלבים עובדים כך:

לכל epoch:

לכל batch בtrain\_loader:

1. נאפס את כל חישובי הגרדיאנטים הקודמים
2. נבצע חיזוי על הbatch החדש
3. נחשב את גודל הטעות של הbatch בעזרת הcriterion שבחרנו קודם (crossEntrophyLoss)
4. נחשב loss.backward() – כל הגרדיאנטים עבור המשקלים והbiases.
5. נבצע optimizer.step() – כלומר עדכון המשקלים והbiases, כלומר לב הלמידה.

נחשב loss וaccuracy ונדפיס למסך כדי שנוכל לעקוב אחרי הלמידה.

בסיכום שלב הלמידה, בוודאי עלתה בכם תהייה לפי מה בחרנו את מבנה השכבות ברשת, גודל הbatch, מספר הepochs, learning rate, מומנטום, וכו'

פרמטרים אלו נקראים היפרפרמטרים (hyperparameters). יש לנו לא מעט מידע שנותן כיוון איך לבחור אותם, אך חלק גדול מהאימון הוא ניסוי היפרפרמטרים שונים ובדיקה מי מהם נותן תוצאות טובות יותר. שמעתי שיחה של מנהל AI בטסלה שאמר שהחברה מאמנת 1000 מודלים במקביל, זאת על מנת למצוא את הכיוון הנכון.

1. **הערכת המודל:**

test\_set = torchvision.datasets.MNIST('/files/',  
 train=True,  
 download=True,  
 transform=transform)  
test\_loader = torch.utils.data.DataLoader(  
 dataset=test\_set,  
 batch\_size=64,  
 shuffle=True)  
  
correct = 0  
net.eval()

with torch.no\_grad():  
 for images, labels in test\_loader:  
 images = images.to(device)  
 labels = labels.to(device)  
 preds = net(images)  
 correct += preds.argmax(dim=1).eq(labels).sum().item()  
  
print('Accuracy of on test images: {}'.format(  
 100 \* correct / len(test\_loader.dataset)))

כאשר נרצה להעריך (evaluate) את הרשת, נעבור למצב net.eval(). הסיבה לכך היא שבשלב זה לא נרצה ששכבות Dropout, BatchNorm  יופעלו, שכן זה יחרבש את התוצאה. במקרה זה לא ממשנו שכבות כאלו אך שווה להזכיר זאת כי משנים את המודל לא מעט ואחר כך קשה להבין מאיפה מגיע חוסר הדיוק.

בנוסף בשלב הטסט נרצה רק לבחון את הרשת, ולא לשפר אותה, לכן לא יהיה צורך שנעקוב אחרי הגרדיאנטים וכו'. הפקודה המתאימה היא torch.no\_grad().

**לסיכום:**

עשינו הכרות עם pytorch בעזרת בניית מודל שחוזה בדיוק גבוה ספרות בין 0-9 בתמונות ערוץ אחד, 28x28. בשלב הבא אני ממליץ לעשות התאמות קטנות לקוד, ולהריץ אותו על dataset משלכם.

כל הקוד לשימוש נוח:

import torch  
import torchvision  
import torch.nn as nn  
import torch.nn.functional as F  
import torch.optim as optim  
import torchvision.transforms as transforms  
from torch.utils.data import DataLoader  
from tqdm import tqdm  
from time import time, sleep  
  
  
transform = transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize((0.5,), (0.5,))  
])  
  
  
train\_set = torchvision.datasets.MNIST('/files/',  
 train=True,  
 download=True,  
 transform=transform)  
  
  
train\_loader = torch.utils.data.DataLoader(  
 dataset=train\_set,  
 batch\_size=64,  
 shuffle=True)  
  
  
images, labels = next(iter(train\_loader))  
print("the shape is ", labels.shape)  
print(labels)  
  
import matplotlib.pyplot as plt  
  
fig = plt.figure()  
for i in range(9):  
 plt.subplot(3, 3, i + 1)  
 plt.tight\_layout()  
 plt.imshow(images[i][0], cmap='gray')  
 plt.xticks([])  
 plt.yticks([])  
  
fig.show()  
  
device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')  
  
  
class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(in\_channels=1,  
 out\_channels=20,  
 kernel\_size=5,  
 stride=1)  
  
 self.conv2 = nn.Conv2d(in\_channels=20,  
 out\_channels=50,  
 kernel\_size=5,  
 stride=1)  
  
 self.fc1 = nn.Linear(in\_features=50 \* 4 \* 4,  
 out\_features=500)  
 self.fc2 = nn.Linear(in\_features=500,  
 out\_features=10)  
  
 def forward(self, t):  
 t = self.conv1(t)  
 t = F.relu(t)  
 t = F.max\_pool2d(t, kernel\_size=2, stride=2)  
  
 t = self.conv2(t)  
 t = F.relu(t)  
 t = F.max\_pool2d(t, kernel\_size=2, stride=2)  
  
 t = t.view(t.size(0), -1)  
 t = self.fc1(t)  
 t = F.relu(t)  
 t = self.fc2(t)  
 return t  
  
  
net = Net()  
  
criterion = nn.CrossEntropyLoss()  
  
optimizer = optim.SGD(params=net.parameters(), lr=0.001, momentum=0.9)  
  
  
def train(net, epoch\_num):  
 start = time()  
 for epoch in range(1, epoch\_num + 1):  
 running\_loss = 0.  
 correct = 0.  
 msg = 'training: epoch {}/{}'.format(epoch, epoch\_num + 1)  
 for images, labels in tqdm(train\_loader, desc=msg):  
 images = images.to(device)  
 labels = labels.to(device)  
 # נאפס את הגרדיאנט  
 optimizer.zero\_grad()  
  
 preds = net(images)  
  
 loss = criterion(preds, labels)  
  
 loss.backward()  
  
 optimizer.step()  
  
 running\_loss += loss.item()  
  
 correct += preds.argmax(dim=1).eq(labels).sum().item()  
  
 print('results: Epoch {}, accuracy {}, loss: {}\n'.format(  
 epoch,  
 (correct / len(train\_loader.dataset)) \* 100,  
 (running\_loss / len(train\_loader))), flush=True)  
  
 sleep(.5)  
  
 end = time()  
 print('Done Training')  
 print('%0.2f minutes' % ((end - start) / 60))  
  
  
net.train()  
train(net, 10)  
  
test\_set = torchvision.datasets.MNIST('/files/',  
 train=True,  
 download=True,  
 transform=transform)  
test\_loader = torch.utils.data.DataLoader(  
 dataset=test\_set,  
 batch\_size=64,  
 shuffle=True)  
  
correct = 0  
net.eval()  
with torch.no\_grad():  
 for images, labels in test\_loader:  
 images = images.to(device)  
 labels = labels.to(device)  
 preds = net(images)  
 correct += preds.argmax(dim=1).eq(labels).sum().item()  
  
print('Accuracy of on test images: {}'.format(  
 100 \* correct / len(test\_loader.dataset)))