# Pytorch

## בניית מודל

נתאר את כל השלבים הנצרכים לבניית מודל רשת נוירונים ב-Pytorch.

From torch import nn

1. בשלב ראשון צריך לבנות מחלקה ב-python שיורשת מ-nn.module. למחלקה זו צריכים להיות שני פונקציות:
2. Init – מגדיר את הארכיטקטורה של הרשת.
3. Forword – מקבלת דאטה x, מריץ אותו על רשת הנוירונים ומחזיר ניבוי.

Class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork, self).\_\_init\_\_()

self.flatten = nn.Flatten()

self.linear\_relu\_stack = nn.Sequential(

nn.Linear(28\*28, 512),

nn.ReLU(),

nn.Linear(512, 10),

nn.ReLU()

)

def forward(self, x):

x = self.flatten(x)

logits = self.linear\_relu\_stack(x)

return logits

1. ניצור אובייקט של המחלקה לעיל, פונקציית loss ואופטימייזר.

model = NeuralNetwork()

loss\_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=learning\_rate)

1. נגדיר שני פונקציות. אחת ל-train ואחת ל-test.

def train\_loop(dataloader, model, loss\_fn, optimizer):

for batch, (X, y) in enumerate(dataloader):

pred = model(X)

loss = loss\_fn(pred, y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

def test\_loop(dataloader, model, loss\_fn):

num\_batches = len(dataloader)

test\_loss, correct = 0, 0

with torch.no\_grad():

for X, y in dataloader:

pred = model(X)

test\_loss += loss\_fn(pred, y).item()

correct += (pred.argmax(1) == y).type(torch.float).sum().item()

test\_loss /= num\_batches

correct /= size

print(f"Test Error: \n Accuracy: {(100\*correct):>0.1f}%, Avg loss: {test\_loss:>8f} \n")

1. נריץ את שני הפונקציות לעיל של train ו-test לפי מספר epochs שנבחר.

epochs = 10

for t in range(epochs):

train\_loop(train\_dataloader, model, loss\_fn, optimizer)

test\_loop(test\_dataloader, model, loss\_fn)

print("Done!")

# NumPy

## הגדרה

NumPy היא ספרייה בפייתון המוסיפה תמיכה במערכים ומטריצות רב-ממדיות, יחד עם אוסף גדול של פונקציות מתמטיות ברמה גבוהה להפעלת מערכים אלה. NumPy היא תוכנת קוד פתוח ויש לה תורמים רבים.

לאחר התקנת NumPy, בכל תוכנית צריך לייבא אותו על ידי import. NumPy מיובא בדרך כלל תחת הכינוי np.

import numpy as np

## מערך ב-NumPy

בפייתון יש לנו רשימות המשרתות את מטרת המערכים, אך הן איטיות לעיבוד. NumPy שואפת לספק אובייקט מערך המהיר פי 50 מהרשימה המובנית של פייתון. מערכי NumPy מאוחסנים במקום אחד רציף בזיכרון בניגוד לרשימות, כך שתהליכים יכולים לגשת אליהם ולנהל אותם בצורה יעילה מאוד. כמו כן, הוא מותאם לעבודה עם ארכיטקטורות המעבד העדכניות ביותר. מערכים משמשים לעתים קרובות במדעי הנתונים, שם המהירות והמשאבים חשובים מאוד.

אובייקט המערך ב-NumPy נקרא ndarray, מפני שניתן להגדיר לכל מערך n ממדים (מערכים מקוננים). מערך ב-NumPy מספק שיטות תומכות רבות שהופכות את העבודה עם ndarray לקלה מאוד. הגישה לאיברים במערך היא כמו שלמדנו בגישה לאיבר ברשימה.

>>> arr0d = np.array(42)

>>> arr1d = np.array([1, 2, 3, 4, 5])

>>> arr2d = np.array([[1, 2, 3], [4, 5, 6]])

>>> arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

>>> arr2d.shape()

(2, 3)

>>> arr3d.ndim()

2

>>> arr3d.dtype()

dtype('int32')

>>> arr2d.size()

6

## פעולות על ndarray

ברשימות רגילות, פעולות אריתמטיות, כמו חיבור וכפל, רק מגדילות את הרשימה. לעומת זאת פעולות אריתמטיות על ndarray מפעיל את הפעולה לכל איבר בנפרד בהתאמה.

>>> np.array([2, 4, 5]) + np.array([7, 3, 2])

array([9, 7, 7])

>>> np.array([2, 4, 5]) \* np.array([7, 3, 2])

array([14, 12, 10])

* np.dot(<np.array>, <np.array>) - מקבלת שני ndarray ומחזירה את המכפלה הפנימית שלהם. עבור שני מערכים חד ממדיים פעולה זו שקולה למכפלת וקטורים, ועבור שני מערכים דו-ממדיים פעולה זו שקולה להכפלת מטריצות.
* np.mean(<np.array>) – מקבלת וקטור ומחזירה את הממוצע של כל הרכיבים בו.
* np.linalg.inv(<np.array>) - מקבלת מערך דו-ממדי (מטריצה) ומחזירה את המטריצה ההופכית שלה, כך שאם יכפילו בין שניהם תתקבל מטריצת היחידה.