SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Programska podrška za duboko učenje

Frano Rajič

Voditelj: Siniša Šegvić

Zagreb, travanj, 2019

Sadržaj

1. Uvod

Smjestimo za početak duboko učenje u odgovarajući kontekst umjetne inteligencije i strojnog učenja. Umjetna inteligencija je dio računalne znanosti koji se bavi razvojem sposobnosti računala da obavljaju zadaće za koje je potreban određeni oblik inteligencije, što znači da se mogu snalaziti u novim prilikama, učiti nove koncepte, donositi zaključke, razumjeti prirodni jezik, raspoznavati prizore i dr.

Nadalje, strojno učenje je dio umjetne inteligencije koja se bavi oblikovanjem algoritama kojima umjetna inteligencija može učiti, odnosno svoju učinkovitost poboljšavaju na temelju empirijskih podataka.

Konačno, duboko učenje je grana strojnog učenja koja koristi neuronske mreže da bi obavljala svoje zadaće i učila. Temelji se na predstavljanju podataka složenim reprezentacijama do kojih se dolazi slijedom naučenih nelinearnih transformacija. Metode dubokog učenja svoju primjenu pronalaze u izazovnim područjima gdje je dimenzionalnost podataka iznimno velika: računalnom vidu, obradi prirodnog jezika ili razumijevanju govora. Najčešće se dijeli u tri široke kategorije:

1) Nadzirano učenje - Računalu je dan ulaz i željeni izlaz, a cilj je pronaći pravila koja preslikavaju ulaze u izlaze.
2) Nenadzirano učenje - Predani su podaci bez ciljne vrijednosti, treba pronaći pravilnost u podacima.
3) Podržano/ojačano učenje - Učenje optimalne strategije na temelju pokušaja s odgođenom nagradom. Računalni program mora izvršiti određeni cilj bez da mu se sugerira je li došao blizu cilja.

Zahvaljujući svojoj opsežnosti te jednostavnosti pisanja programa, programski jezik Python jedan je od idealnih izbora za pisanje programa vezanih za strojno učenje općenito. Stoga su za jednostavnije korištenje algoritama dubokog učenja stvorene brojne Python biblioteke koje korisniku nude kako visoku efikasnost tako i vrlo jednostavnu primjenu algoritama u svojem programu. Te biblioteke pružaju funkcionalnost vektorske obrade podataka, učitavanja i obrade slika, automatsku diferencijaciju, podršku za GPU i iscrtavanje rezultata. U nastavku ovog rada se iznosi pregled poznatih biblioteka za spomenute funkcionalnosti, a uzgred tomu opisuje minimalistički program za klasifikaciju na FashionMNIST skupu podataka.

2. Opis programske podrške

Programska podrška je u Pythonu ostvarena kroz skup raznih biblioteka. Među njima su najpopularnije i najkorištenije sljedeće:

- numpy za vektorsku obradu podataka
- libjpeg-turbo i pillow-simd za učitavanje i obradu slika
- PyTorch za automatsku diferencijaciju
- cudnn za GPU podršku
- matplotlib za iscrtavanje rezultata.

Slika 2.1 Python logo

2.1 Biblioteka Numpy

Programski jezik Python osigurava bogat skup podatkovnih struktura visoke razine: liste za enumeraciju kolekcija objekata, rječnike za izgradnju hash tablica, itd. Međutim, te strukture nisu prikladne za numeričko računanje visokih performansi pa je upravo zbog toga biblioteka Numpy postala standardna reprezentacija numeričkih podataka u Pythonu. Ona nudi podršku za velike, višedimenzionalne nizove i matrice, kao i izvođenje brojnih matematičkih funkcija nad njima. Implementirana je tako da osigurava brzo i efikasno izvođenje tih funkcija i time pruža znatno bolje performanse u odnosu na standardne Python strukture. Tri tehnike koje nadasve koristi da bi u tome uspjela su: vektorska obrada podataka, izbjegavanje kopiranja podatka u memoriji i minimiziranje broja operacija.

Osnovna funkcionalnost biblioteke je struktura podataka "ndarray" koja predstavlja homogeno, višedimenzionalno polje. Neke od najvažnijih operacija koje se mogu izvršavati nad poljem uključuju: množenje i transponiranje matrica, rješavanje sustava jednadžbi, množenje i normalizacija vektora.

Slika 2.2 NumPy logo

2.1.1 Stvaranje polja i osnovne operacije

Primjer u nastavku demonstrira stvaranje polja te neke od osnovnih operacija koje pruža NumPy biblioteka.

```
>>> ################################################################
>>> ### PRIMJER KORIŠTENJA BIBLIOTEKE NUMPY - OSNOVNE OPERACIJE ###
>>> ################################################################
>>>
>>> #Učitavanje biblioteke
>>> import numpy as np
>>>
>>> #Stvaranje vektora u obliku retka
>>> vector_row = np.array([1,2,3]) #np.array stvara primjerak od ndarray
```

```
>>>
>>> #Stvaranje vektora u obliku stupca
>>> vector_column = np.array([[1],[2],[3]])
>>>
>>> print(vector_row[2]) #Odaberi treći element vektora
3
>>> print(vector_row[:]) #Odaberi sve elemente vektora
[1 2 3]
>>> print(vector_row[0:2]) #Odaberi elemente vektora od prvog do drugog, bez trećeg
[1 2]
>>> print(vector_row[-1]) #Odaberi posljednji element vektora
3
>>>
>>> #Stvaranje dvodimenzionalne matrice
>>> #Moguće je naznačiti tip podataka dodatnim argumentom dtype
>>> matrix = np.array([[1,2,3],[4,5,6]], dtype=np.int8)
>>> print(matrix)
[[1 2 3]
 [4 5 6]]
>>>
>>> print(matrix[1,1]) #Odaberi 2. redak i 2. stupac matrice
5
>>> print(matrix[:,1:2]) #Odaberi sve retke i drugi stupac
[[2]
 [5]]
>>>
>>> #Dohvaćanje informacija o matrici
>>> print(matrix.shape) #Dohvati broj redaka i stupaca
(2, 3)
>>> print(matrix.size) #Dohvati broj elemenata (reci*stupci)
6
>>> print(matrix.ndim) #Dohvati broj dimenzija (kod nas dvodimenzionalno polje)
2
>>>
>>> #Preoblikovanje polja
>>> print(matrix.reshape(6,1)) #Preoblikovanje iz 2x3 u 6x1
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
>>> print(matrix.reshape(1,-1))#Preoblikovanje gdje 1 znači “koliko god je potrebno”
[[1 2 3 4 5 6]]
>>> print(matrix.flatten()) #Za pretvaranje u 1D polje, isto kao reshape(-1,1)
[1 2 3 4 5 6]
>>>
>>> #Operacije nad elementima
>>> print(matrix+1) #Dodaj 1 na svaki element matrice
[[2 3 4]
 [5 6 7]]
>>> print(matrix*2) #Pomnoži svaki element matrice s 2
[[ 2  4  6]
 [ 8 10 12]]
>>> print(np.sin(matrix)) #Izračunaj sinus svakog elementa matrice
[[ 0.8413  0.909   0.1411]
 [-0.757  -0.959  -0.2793]]
```

```
>>> print(matrix + 2*matrix) #Zbrajanje dviju matrica - samo ako su kompatibilne!
[[ 3  6  9]
 [12 15 18]]
>>> for i in np.ndindex(matrix.shape): #Iteracija nad svim elementima
...     print(i, " --> ", matrix[i])
...
(0, 0)  -->  1
(0, 1)  -->  2
(0, 2)  -->  3
(1, 0)  -->  4
(1, 1)  -->  5
(1, 2)  -->  6
>>> #Definiranje novih operacija
>>> add_5 =lambda i: i+5 #Funckija definirana lambdom koja dodaje 5 na dani broj
>>> vectorized_add_5= np.vectorize(add_5) #Pretvaranje u funkciju nad vektorima
>>> print(vectorized_add_5(matrix)) #Primjeni funkciju na sve elemente matrice
[[ 6  7  8]
 [ 9 10 11]]
>>> print(matrix) #Matrica je nepromjenjena jer nismo pridodjelili novu vrijednost
[[1 2 3]
 [4 5 6]]
>>>
>>> #U potrazi za min i max vrijednostima
>>> print(np.max(matrix)) #Pronađi najveći element
6
>>> print(np.argmax(matrix)) #Pronađi index najvećeg elementa
5
>>> print(np.max(matrix,axis=0)) #Pronađi najveći element u svakom stupcu
[4 5 6]
>>> print(np.max(matrix,axis=1)) #Pronađi najveći element u svakom redku
[3 6]
>>> print(np.min(matrix)) #Pronađi najmanji element
1
>>>
>>> #Malo statistike
>>> print(np.mean(matrix)) #Srednja vrijednost
3.5
>>> print(np.std(matrix)) #Standardna devijacija
1.707825127659933
>>> print(np.var(matrix)) #Varijanca
2.9166666666666665
```

Primjer 2.1 Korištenje osnovne funkcionalnosti koju osigurava biblioteka NumPy

2.1.2 Linearna algebra

Biblioteka NumPy pruža velik broj funkcija za rad s dvodimenzionalnim poljima. U nastavku je primjer korištenja osnovnih funkcija.

```
>>> ################################################################
>>> ### PRIMJER KORIŠTENJA BIBLIOTEKE NUMPY - LINEARNA ALGEBRA ###
>>> ################################################################
>>>
>>> #Učitavanje biblioteke numpy
>>> import numpy as np
>>> #Učitavanje dodatnih funkcija iz biblioteke numpy
>>> from numpy.random import rand
>>> from numpy.linalg import solve, inv
>>>
>>> #Stvaranje polja
>>> a = np.array([[1, 2, 3], [3, 4, 6.7], [5, 9.0, 5]]) #Polje dimenzija 3x3
>>> b = np.array([3, 2, 1]) #Polje dimenzija 1x3
>>> c = rand(3, 3) #Polje 3x3 s random vrijednostima
>>>
>>> #Što je u polju c?
>>> print(c)
[[0.41947571 0.52826777 0.84974037]
 [0.78311044 0.37670877 0.61183364]
 [0.26541033 0.28880773 0.82091482]]
>>>
>>> #Transponiranje, inverz i rješavanje sustava
>>> print(a.transpose()) #Transponiranje matrice
[[1.  3.  5. ]
 [2.  4.  9. ]
 [3.  6.7 5. ]]
>>> print(inv(a)) #Računanje inverza matrice
[[-2.27683616  0.96045198  0.07909605]
 [ 1.04519774 -0.56497175  0.1299435 ]
 [ 0.39548023  0.05649718 -0.11299435]]
>>> print(solve(a, b)) #Rješavanje jednadžbe ax = b
[-4.83050847  2.13559322  1.18644068]
>>>
>>> #Množenje matrica, determinanta i rang matrice
>>> print(np.dot(a, c)) #Množenje dviju matrica (dot product)
[[ 2.78192757  2.14810851  4.5361521 ]
 [ 6.16911808  5.0266502  10.49668493]
 [10.47242412  7.47575642 13.85977868]]
>>> print(a@c) #Također množenje matrica
[[ 2.78192757  2.14810851  4.5361521 ]
 [ 6.16911808  5.0266502  10.49668493]
 [10.47242412  7.47575642 13.85977868]]
>>> print(np.linalg.det(a)) #Izračunaj determinantu matrice
17.700000000000006
>>> print(np.linalg.matrix_rank(a)) #Izračunaj rang matrice
3
```

Primjer 2.2 Korištenje funkcija linearne algebre biblioteke NumPy

2.2 Biblioteka pillow-simd i libjpeg-turbo

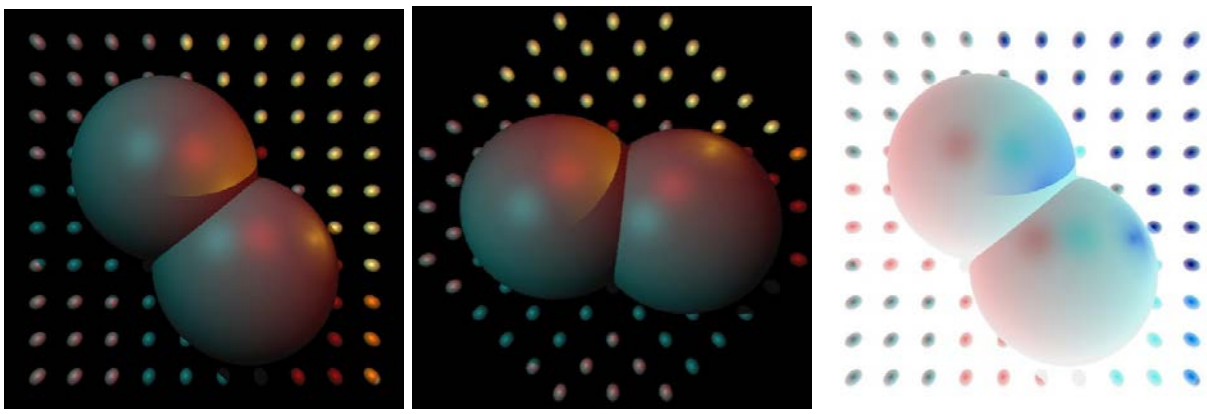
Kao podrška za procesiranje slike u Pythonu postoji biblioteka pillow-simd. Ona omogućuje brzo izvođenje obrade nad slikama kao što je rotacija, mijenjanje veličine i afina transformacija, vršenje operacija nad pikselima i filtriranje. Veliku brzinu izvođenja zahvaljuje korištenju SIMD pristupa (Single Instruction, Multiple Data) u kojemu se više podataka istovremeno procesira u jednoj instrukciji procesora.

Kako bi imao veliku brzinu učitavanja slika, pillow-simd koristi besplatnu biblioteku libjpeg-turbo koja implementira JPEG kodek. Svoju funkcionalnost enkodiranja i dekodiranja JPEG slika pri visokim performansama libjpeg-turbo također zahvaljuje korištenju SIMD pristupa.

U nastavku je jednostavan primjer korištenja biblioteke pillow-simd.

```
1.  ################################################
2.  ### PRIMJER KORIŠTENJA BIBLIOTEKE PILLOW-SIMD ###
3.  ################################################
4.
5.  # Učitavanje potrebnih dijelova biblioteke
6.  from PIL import Image
7.  import PIL.ImageOps
8.
9.  colorImage = Image.open("C:/Users/Frano/ana3/raytracer.jpg") # Stvori Image objekt
10. colorImage.show() # Prikaži sliku
11.
12. rotated = colorImage.rotate(45) # Rotiraj za 45 stupnjeva
13. rotated.show() # Prikaži rotiranu sliku
14.
15. inverted = PIL.ImageOps.invert(colorImage) # Invertiraj pixele slike
16. inverted.show() # Prikaži invertiranu sliku
```

Primjer 2.3 Korištenje biblioteke pillow-simd



Slika 2.3.Slike prkazuju redom originalnu, rotiranu i sliku s invertiranim pikselima

2.3 Biblioteka PyTorch

PyTorch je programski okvir za duboko učenje te paket za znanstveno računanje u Pythonu koji sadrži sve potrebno za jednostavnu implementaciju dubokog učenja uz akceleraciju računanja korištenjem grafičke procesorske jedinice. Aspekt znanstvenog računarstva je prije svega rezultat PyTorch-ove „tensor“ biblioteke i operacija nad njom.

Slika 2.4. PyTorch logo

2.3.1 Duboko učenje i PyTorch

Tenzor je višedimenzionalni niz i vrlo je sličan ndarray-u biblioteke NumPy. Na konceptualnoj razini su identična stvar, a u praksi se bez problema u jednoj liniji koda prebaci iz jednog u drugi, kao što se može vidjeti u primjeru 2.4. Štoviše, tenzor je uveden kao zamjena za ndarray koja koristi GPU snagu.

```
>>> import numpy as np       #Učitavanje biblioteke NumPy
>>> import torch             #Učitavanje biblioteke PyTorch
>>>
>>> n = np.array([1,2,3])    #Stvaranje ndarray objekta
>>> print(n)
[1 2 3]
>>>
>>> t = torch.as_tensor(n) #Stvaranje tenzora iz ndarraya
>>> print(t)
tensor([1, 2, 3], dtype=torch.int32)
>>>
>>> n_again = t.numpy() #Tenzor pretvoren u ndarray
>>> print(n_again)
[1 2 3]
```

Primjer 2.4. Primjer prikazuje prelazak iz ndarray objekta u tensor objekt, odnosno iz tensor objekta u ndarray.

PyTorch tenzori imaju ugrađenu GPU podršku te se operacije nad tenzorima mogu izvoditi na grafičkim procesorskim jedinicama. Sasvim je jednostavno premjestiti tenzore na GPU jezgre kao i skinuti ih s njih, ako je podržan GPU sustava. Koliko to znači i zašto bi koristili GPU za računanje s tenzorima je opisano u poglavlju 2.4.

Tenzori su vrlo bitni za duboko učenje i neuronske mreže jer su osnovna podatkovna struktura koja se koristi za izgradnju i treniranje neuronskih mreža. Pored njih, PyTorch naravno nudi mnogo više u smislu izgradnje i treniranja neuronskih mreža.

Sljedeća tablica prikazuje listu PyTorch paketa i njihovih odgovarajućih opisa.

Paket	Opis
torch	Paket najviše razine, sadrži tenzor biblioteku.
torch.nn	Podpaket koji sadrži module i proširive klase za izgradnju neuronskih mreža.
torch.autograd	Podpaket koji podržava operacije diferencijacije.
torch.nn.functional	Funkcionalno sučelje koje sadrži tipične operacije korištene za izgradnju neurosnkih mreža kao što su funkcije gubitaka, aktivacijske funkcije i operacije konvolucija.
torch.optim	Podpakat koji je sadrži standardne operacije optimizacije kao što su SGD i Adam.
torch.utils	Podpaket koji sadrži utility klase za primjerice učitavanje podataka, koji olakšavaju predobradu podataka.
torchvision	Paket koji omogućuje pristup popularnim skupovima podataka, arhitekturama modela i transformacijama slika za računalni vid.

Tablica 2.1. Opis paketa biblioteke PyTorch

U nastavku je dan primjer korištenja razreda Tensor i osnovnih operacija nad njim.

In [1]:
```
from __future__ import print_function
import torch
```

Konstruiranje neinicijalizirane 5x3 matrice:

In [2]:
```
x = torch.empty(5, 3)
print(x)
```

```
tensor([[1.0561e-38, 1.0653e-38, 4.1327e-39],
        [8.9082e-39, 9.8265e-39, 9.4592e-39],
        [1.0561e-38, 1.0653e-38, 1.0469e-38],
        [9.5510e-39, 9.0000e-39, 9.0919e-39],
        [9.2755e-39, 9.1837e-39, 1.6115e-43]])
```

Konstruiranje nasumično inicijalizirane matrice:

In [3]:
```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.2087, 0.0946, 0.2116],
        [0.9154, 0.1330, 0.0731],
        [0.7495, 0.8692, 0.1935],
        [0.6271, 0.9219, 0.7988],
        [0.8370, 0.0054, 0.0176]])
```

Konstruiranje matrice inicijalizirane nulama, koja sadrži podatke tipa "long":

In [4]:
```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

Konstruiranje tenzora s danim vrijednostima:

In [5]:
```
x = torch.tensor([5.5, 3])
print(x)
```

```
tensor([5.5000, 3.0000])
```

Stvaranje tenzora na temelju postojećeg:

In [6]:
```
x = x.new_ones(5, 3, dtype=torch.double)
print(x)

x = torch.randn_like(x, dtype=torch.float)
print(x)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[-1.2752, -0.5818, -1.5521],
        [ 0.7898, -0.2537,  1.0751],
        [ 1.4452,  0.8169,  1.3914],
        [-1.1131,  0.2407, -0.9046],
        [ 0.3668, -0.2557,  1.8421]])
```

Dohvati dimenzije tenzora:

In [7]:
```
print(x.shape)
```

```
torch.Size([5, 3])
```

Zbrajanje tenzora:

In [8]:
```
y = torch.rand(5, 3)
print(x + y)
```

```
tensor([[-0.9925, -0.0962, -0.6375],
        [ 1.1984,  0.3334,  1.3040],
        [ 1.8555,  1.5317,  1.9106],
        [-1.0971,  0.2870, -0.6430],
        [ 1.1576,  0.5039,  2.1575]])
```

Zbroji x i y te pohrani u y:

In [9]:
```
y.add_(x)
print(y)
```

```
tensor([[-0.9925, -0.0962, -0.6375],
        [ 1.1984,  0.3334,  1.3040],
        [ 1.8555,  1.5317,  1.9106],
        [-1.0971,  0.2870, -0.6430],
        [ 1.1576,  0.5039,  2.1575]])
```

> Operacije koje mijenaju tenzor su naznačene s _ . Tako primjerice i ove operacije mijenjaju tenzor y: „y.copy_(x)“, „y.t_()“.

Za mijenjaje dimenzija tenzora koristimo metodu `view` :

In [10]:
```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)  # -1 označava da se dimenzija određuje na temelju ostalih
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

Kod tenzora s jednim elementom možemo taj element dohvatiti pozivom `.item()` :

In [11]:
```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([-0.5117])
-0.5117341876029968
```

Primjer 2.5. Korištenje razreda Tensor te osnovne operacije nad njim

Slično kao i NumPy ndarray objekti, PyTorch tenzori osiguravaju velik broj funkcionalnosti kao što su transponiranje matrice, razni načini indeksiranja elemenata, mnogobrojne matematičke operacije, linearna algebra itd. Kako je primjer dan za ndarray u prethodnom poglavlju za te stvari, tako ovdje neće biti ponovljen.

2.3.2 Kratki pogled u prošlost

Inicijalno izdanje PyTorch-a bilo je u listopadu 2016., a prije stvaranja PyTorch-a, postojao je i još uvijek postoji okvir zvan Torch. Torch je programski okvir za strojno učenje koji postoji već duže vrijeme i temelji se na Lua programskom jeziku. Veza između PyTorch-a i Lua verzije postoji jer mnogi programeri koji održavaju Lua verziju su pojedinci koji su stvorili PyTorch.

Soumith Chintala je zaslužan za pokretanje PyTorch projekta, a njegov razlog za stvaranje PyTorch-a je prilično jednostavan - Lua verzija Torcha postaje zastarjela pa je tako potrebna novija verzija napisana u Pythonu. Kako je Soumith Chintala pri pokretanju PyTorcha radio u Facebook AI istraživačkoj skupini (i na dan pisanja ovog rada također radi), PyTorch se često naziva Facebookovim. Međutim, postoji mnoge druge organizacije koje koriste PyTorch, primjerice Twitter i Salesforce.

Slika 2.5. Soumith Chintala

Slika 2.6. Facebook AI Research group logo

2.3.3 Zašto PyTorch

Kada gradimo neuronske mreže s PyTorchom, vrlo smo blizu programiranju neuronskih mreža od nule. Iskustvo programiranja u PyTorchu je najbliže onome što se stvarno događa u neuronskim mrežama. Nakon korištenja PyTorcha, lako je vidjeti kako proces funkcionira krene li se iz ničega u čistom Pythonu, zbog čega je PyTorch odličan za početnike i daje dublje razumijevanje neutronskih mreža i dubokog učenja u cjelini.

Za takvu prirodu biblioteke je zaslužna filozofija koje se programeri iza PyTorcha drže, a koja u središte pozornosti postavlja neuronske mreže, ne sam okvir, i može se sažeti u ovih par točaka:

- **Stay out of the way** (skloni se s puta programeru što je više moguće te dopusti da se fokusira na neuralne mreže umjesto na framework)
- **Cater to the impatient** (pobrinuti se za nestrpljive)
- **Promote linear code-flow** (promicanje linearnog protoka koda)
- **Full interop with the Python ecosystem** (potpuna povezanost s Python ekosustavom)
- **Be as fast as anything else** (biti brzi kao bilo što drugo ponuđeno)

S perspektive ulaganja znanja, PyTorch se može smatrati sigurnijom opcijom zato što ga Facebook podupire i izgrađen je za Python, jezik s velikom i dalje rastućom zajednicom dubokog učenja. Uz visoku integriranost s Pythonom, PyTorch je lagan programski okvir u pogledu prethodno opisane kombinacije apstrakcije i jednostavnosti te se može lakše prilagoditi brzo razvijajućem svijetu dubokog učenja. Stoga je razumna tvrdnja da će PyTorch dugovječno ostati korišteni programski okvir za duboko učenje.

Dodatna PyTorch karakteristika je prikladnost za istraživanje koju duguje svojem tehničkom dizajnu. PyTorch je izgrađen tako da u osnovi koristi dinamičke računske grafove za provođenje računanja na tenzorima u neuronskoj mreži. Kako bismo optimizirali neuronske mreže, moramo računati derivacije, a da bismo to učinili pomoću računala, programski okviri dubokog učenja koriste upravo računske grafovima. Dinamički računski grafovi se stvaraju tijekom računanja, u odnosu na statičke koji su u potpunosti određeni još prije samih operacija. Najnovije istraživačke teme dubokog učenja zahtijevaju ili imaju veliku korist upravo od korištenja dinamičkih grafova

2.4 CUDA i biblioteka cuDNN

cuDNN je biblioteka za duboke neuronske mreže koja pruža akceleraciju računanja korištenjem grafičke procesorske jedinice. Za razumijevanje što cuDNN odnosno CUDA nudi, najprije ću se dotaknuti paralelizacije i grafičkih procesorska jedinica.

2.4.1 Grafička procesorska jedinica i sramotno lako paraleliziranje

GPU je procesor namijenjen specijaliziranim izračunima, dok s druge strane centralna procesorska jedinica (CPU) dobro izvodi opće proračune. Zbog toga GPU može biti mnogo brži u proračunavanju od CPU-a, ovisno o vrsti izračuna koji se izvodi. Tip računanja koji je najprikladniji za GPU je izračun koji se može obaviti paralelno.

Paralelizacija je vrsta računanja gdje se određeni račun može razbiti u nezavisne manje izračune, koji se zatim mogu izvoditi istovremeno. Rezultati manjih izračuna se zatim kombiniraju (ili sinkroniziraju), da bi se dobio rezultat izvornog većeg izračuna. Kako bi razbijanje većeg zadatka imalo smisla, mora postojati dovoljan broj jezgri koje bi zadužili za manje proračune, a broj jezgri ovisi o korištenom hardvera. Jezgre su jedinice koje zapravo računaju unutar procesora, a današnji CPU obično ima 4, 8 ili 16 jezgri, za razliku od GPU koji imaju na tisuće. Mnogobrojnost jezgri čini grafičke procesorske jedinice posebno prikladnima za paralelno obavljanje zadataka. Ako se računanje može obaviti paralelno, možemo ga ubrzati pomoću pristupa paralelnog programiranja i GPU-ova.

Neuronske mreže pa i duboko učenje koje ih koristi se sramotno lako paraleliziraju. Sramotno lako paralelizirajući proračunski zadatak je onaj gdje je potrebno malo ili nimalo napora kako bi se cjelokupni zadatak razdvojio na skup manjih zadataka koji će se paralelno izračunati. Kod takvih zadataka se lako vidi da su svi zadatci u skupu manjih zadataka neovisni jedni u odnosu na druge. Kako se neuronske mreže sramotno lako paraleliziraju, grafičke procesorske jedinice su osobito pogodne za potrebne izračune u dubokom učenju.

2.4.2 Nvidia i GPGPU

U početku su glavne zadaće koje su ubrzane pomoću GPU-a bile usko vezane uz računalnu grafiku pa se otuda i pojavljuje naziv grafičke procesne jedinice, no posljednjih se godina pojavilo mnogo više paralelnih zadataka. Jedan od takvih zadataka je duboko učenje. Stoga funkcionalnost GPU-a da obavlja i operacije opće namjene odnosno GPGPU (general purpose graphical processing unit) stječe sve veću važnost.

Nvidia je tehnološka tvrtka koja dizajnira GPU-ove. Pionir je u prostoru grafičkih procesorskih jedinica opće namjene (GPGPU). Međutim, Nvidia se prema GPU

računanju opće namjene odnosi kao da je istovjetno GPU računanju, ističući važnost i očekujući da se u budućnosti to i podrazumijeva pod GPU.

Slika 2.7. Nvidia logo

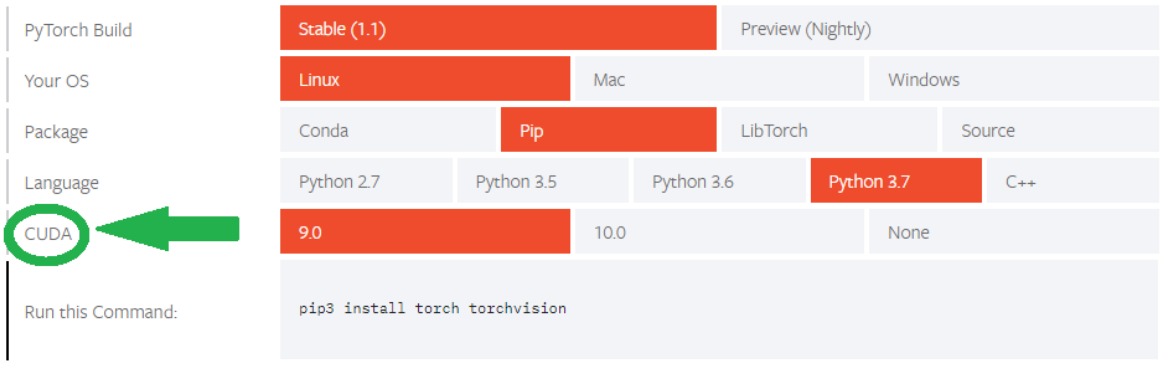
CEO Jensen Huang's predvidio je GPU računanje vrlo rano, zbog čega je softverska platforma CUDA nastala prije gotovo 10 godina. CUDA je kreirana kao platforma koja se spaja s Nvidia GPU hardverom i programerima olakšava izradu softvera koji ubrzava računanje koristeći paralelnu procesorsku snagu Nvidia GPU-a.

Nvidia GPU je hardver koji omogućuje paralelne računanje, dok je CUDA softver koji pruža API programerima za korištenje. Kao rezultat toga je za korištenje CUDA-e potreban Nvidia GPU, a CUDA se može besplatno preuzeti i instalirati s Nvidia web stranice. Programeri koriste CUDA preuzimanjem CUDA alata. Uz alat dolaze specijalizirane biblioteke poput cuDNN (*CUDA Deep Neural Network)* koja sadrži implementacije algoritama dubokog učenja velike brzine.

2.4.3 PyTorch dolazi s CUDA-om

Jedna od prednosti korištenja PyTorch-a ili bilo kojeg drugog API-ja za neuronske mreže jest da je korištenje paralelizma unaprijed ugrađeno. Zbog toga se programer neuronske mreže može više usredotočiti na izgradnju neuronskih mreža, a manje na probleme s performansama.

U PyTorch se uključivanje CUDA nudi već pri samom početku pri preuzimanju PyTorcha, nema dodatnih preuzimanja. Sve što trebamo je imati podržan Nvidia GPU i možemo iskoristiti CUDA koristeći PyTorch. Ne moramo izravno znati koristiti CUDA API. Kako izgleda preuzimanje odnosno instalacija PyTorcha sa službene stranice je prikazano na slici ispod.



Slika 2.8. Instalacija PyTorcha nudi odmah odabir željene CUDA verzije

Iskoristiti CUDA je izuzetno lako u PyTorch-u. Ako želimo da se određeno računanje izvrši na GPU-u, možemo narediti PyTorch-u da to učini pozivanjem funkcije cuda() na željenom tenzoru.

```
1.  >>> import numpy as np       #Učitavanje biblioteke NumPy
2.  >>> import torch             #Učitavanje biblioteke PyTorch
3.  >>>
4.  >>> t = torch.tensor([1,2,3]) #Tenzor se predpostavljeno stvori na CPU
5.  >>> t
6.  tensor([1, 2, 3])
7.  >>>
8.  >>> t = t.cuda()      #Za prebacivanje računanja s tim tenzorom na GPU
9.  >>> t
10. tensor([1, 2, 3], device='cuda:0')  #Računanje se izvodi na GPU s indeksom 0
```

Primjer X Korištenje GPU u PyTorchu

2.4.4 GPU može biti sporiji od CPU-a

Vidjeli smo da računanje možemo provesti proizvoljno na CPU ili GPU, no zašto jednostavno sve ne proračunavamo na GPU?

Odgovor na to pitanje je da je GPU brži samo za specifične zadatke. Možemo primjerice susresti situaciju da će usko grlo biti prijenos podataka s CPU na GPU, što će značajno usporiti izvršavanje zadatka. Frekventno pomicanje relativno malih proračunskih zadataka na GPU primjerice uzrokuje tu situaciju.

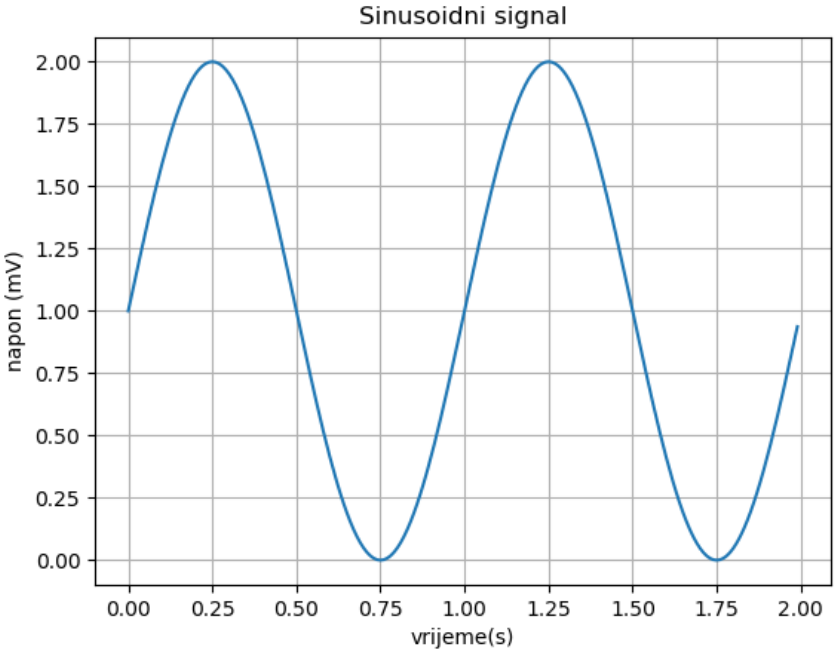
Stoga ćemo radije koristiti CPU za jednostavnije zadatke, a susretnemo li zadatke proračunski teške za izvest koji se mogu paralelizirati, onda ćemo njih radije premjestiti na GPU.

2.5 Biblioteka matplotlib

Matplotlib je open-source biblioteka za iscrtavanje 2D figura. Dizajnirana je za jednostavnu upotrebu, ali pruža velik broj mogućnosti kroz objektno orijentirani API za ugradbu grafova u aplikacije kroz grafička sučelja poput Tkinter. Qt, itd. U okviru dubokog učenja se često koristi za vizualiziranje rezultata

Primjer jednostavnog programa koji prikazuje graf sinusoidnog signala te ga sprema kao sliku „test.png“ dan je u nastavku.

```
1.  import matplotlib
2.  import matplotlib.pyplot as plt
3.  import numpy as np
4.
5.  # Podatci za crtanje
6.  t = np.arange(0.0, 2.0, 0.01)
7.  s = 1 + np.sin(2 * np.pi * t)
8.
9.  fig, ax = plt.subplots()
10. ax.plot(t, s)
11.
12. ax.set(xlabel='vrijeme(s)', ylabel='napon (mV)',
13.        title='Sinusoidni signal')
14. ax.grid()
15.
16. fig.savefig("test.png")
17. plt.show()
```



Slika 2.7. Iscrtani graf za prethodno dani isječak koda

3. Opis minimalističkog programa

Za stvaranje umjetne duboke neuronske mreže su nužne dvije komponente. Prva je programski kod koji opisuje kako neuronska mreža sveukupno funkcionira, a druga je skup podataka odnosno dataset. Podatci su ono na temelju čega umjetna neuronska mreža uči i napreduje te bez njih treniranje mreže ne bi bilo moguće. Podatci se kasnije također predaju mreži kako bi riješila zadaću za koju je izgrađena. Zanimljivo je primijetiti da je najteži dio dubokog učenja upravo prikupljanje skupa podataka.

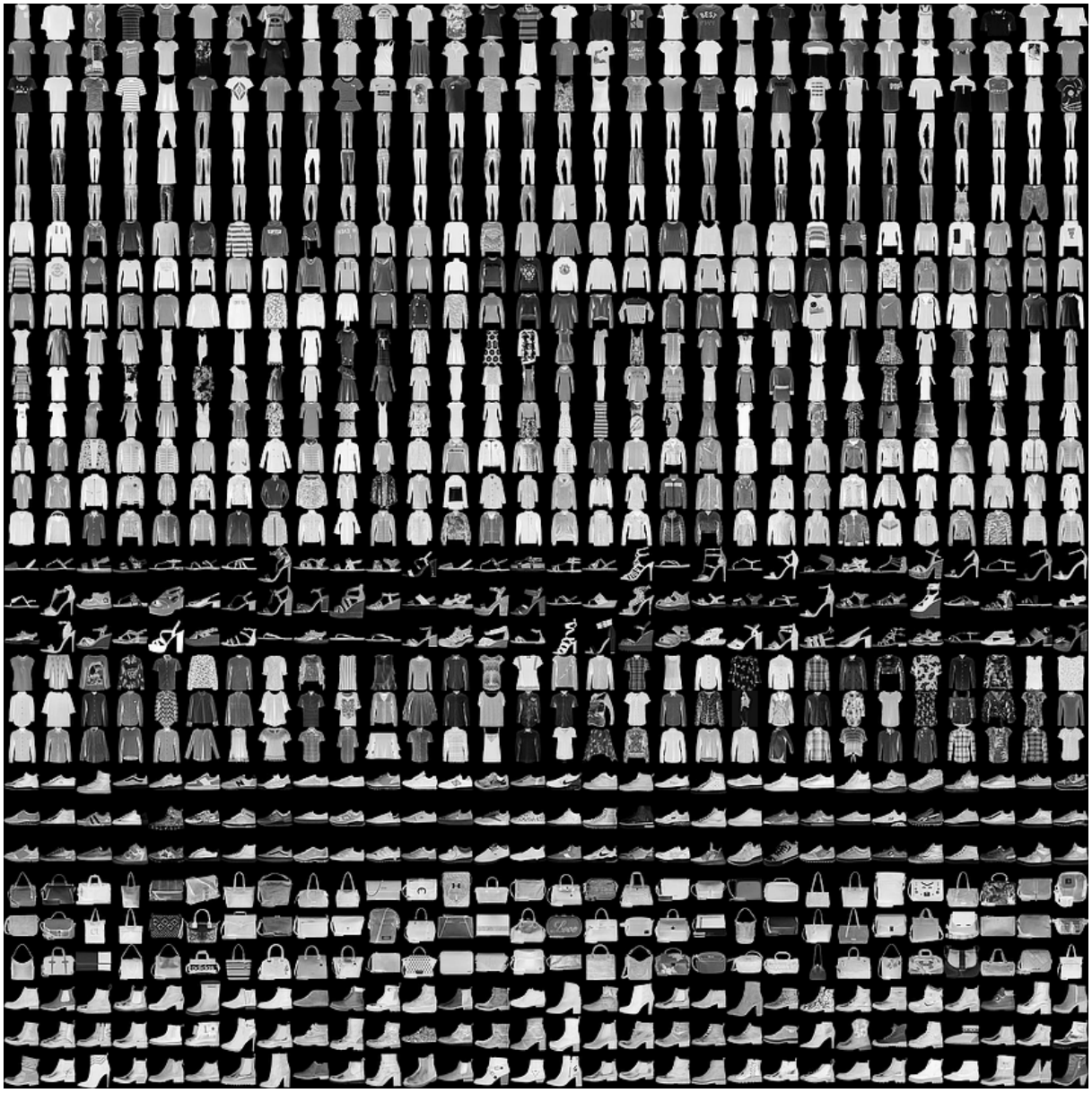
Prije nego nastavim na programski kod minimalističkog programa, opisati ću kratko MNIST i FashionMNIST skup podataka jer će podatke iz FashionMNISU-a koristiti neuronska mreža iz minimalističkog programa za treniranje. Skup podataka MNIST (*Modified National Institute of Standards and Technology database*) je poznat dataset ručno napisanih znamenki koji je sakupljen 1980-ih. Svojevrstan je „Hello world“ u svijetu dubokog učenja jer je najčešće prvi dataset koji programer dubokog učenja iskoristi. Nudi 60 tisuća slika u skupu podataka za treniranje mreže te dodatnih 10 tisuća slika u skupu podataka za testiranje razvijene neuronske mreže. Sve slike su klasificirane u kategorije od 0 do 9, ovisno o znamenki koja je na prikazana na slici. Homogeno su raspoređene, što znači da je svaka znamenka zastupljena u jednakom broju. Dio MNIST-a je prikazan na slici 3.1.

Slika 3.1. Dio slika iz MNIST dataseta

FashionMNIST je za razliku od MNIST-a znatno noviji. Objavljen je u kolovozu 2017. i bazira se na slikama artikala odjeće i obuće iz njemačke online trgovine zalando.de. Tim Zalando Research koji je dio zaladno.de trgovine je sastavio FashionMNIST s namjerom da postupno zamijeni MNIST, zbog čega ima isti broj slika u skupu za treniranje i testiranje, jednak oblik slika te jednak broj klasa. 10 klasa koje čine FashionMNIST su navedene u tablici 2.2., a isječak iz skupa slika je prikazan na slici 3.2.

Oznaka	Opis
0	Majica
1	Hlače
2	Džemper
3	Haljina
4	Kaput
5	Sandale
6	Košulja
7	Tenisice
8	Torba
9	Čizma za gležanj

Tablica 2.2. Oznake predmeta u Fashion MNIST datasetu



Slika 3.2. Dio slika iz FashionMNIST dataseta

Razlozi za zamjenu MNIST-a su sljedeći:

- MNIST je postao prelagan dataset na kojem pojedine konvolucijske mreže dostižu čak 99.7% točnosti na testiranju
- MNIST nije reprezentativan skup podataka za moderne probleme računalnog vida.

FashionMNIST je kao i MNIST moguće dohvatiti putem Python biblioteke torchvision. Torchvision uz biblioteku torch čini PyTorcha, a odvojena je od torcha jer sadrži gotove skupove podataka, gotove arhitekture modela neuronskih mreža te specifične transformacije nad slikama korištene općenito u računalnom vidu.

Konačno, krenimo na opis minimalističkog programa. Program postupno stvara, trenira i testira jedan model duboke neuronske mreže. Radni tijek ovog programa kao i općenito razvijanja neuronske mreže je sljedeći:

1. Pronaći skup ili stvoriti skup podataka s kojima želimo da neuronska mreža radi. Dataset učitati te stvoriti dataloader koji će olakšati kasnije učitavanje podataka.
2. Modeliranje arhitekture neuronske mreže, odnosno definiranje slojeva koji ju grade i načina na koji se će se ulaz kretati kroz slojeve mreže. Potom i instanciranje definirane mreže.
3. Pripremiti treniranje definiranjem funkcije treniranja, što uključuje odabir optimizacijskog algoritma, funkciju gubitka, itd.
4. Trenirati mrežu – hrana su podatci iz odabranog skupa podataka koje su učitani dataloaderom.
5. Koristiti istreniranu mrežu najprije za testiranje kako bi se vidjela njena uspješnost tj. točnost.
6. Na temelju rezultata testiranja ponavljati korake 2. – 5. dok ne budemo zadovoljni s dobivenim rezultatima istrenirane neuronske mreže.
7. Koristiti dobro istreniranu mrežu za rješavanje novih problema, primjerice za prepoznavanje slika koje nisu do sada bile u skupu podataka.

Programski kod programa koji sam napisao i smjestio u nastavku se bavi koracima 1. – 5. te naposljetku razvije mrežu koja daje točnost od 85.233% na testnom datasetu. Kod je popraćen komentarima koji objašnjavaju što se to točno događa.

0. korak - učitavanje potrebnih biblioteka

Učitavanje svih potrebnih PyTorch biblioteka

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import torchvision
import torchvision.transforms as transforms
```

Učitavanje standardnih Python paketa za data science

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.metrics import confusion_matrix
#from plotcm import plot_confusion_matrix

import pdb

torch.set_printoptions(linewidth=120)
```

1. korak - dataset i dataloader

Pored njihovog stvaranja ćemo također nešto detaljnije zaviriti u dataset i pogledati kako su slike spremljene u njemu. Za stvaranje dataseta i dataloadera koristimo PyTorch razrede:

- torch.utils.data.Dataset
- torch.utils.data.DataLoader

Za istancu FashionMNIST dataseta koristimo torchvision koji sam dohvaća dataset koji zatražimo:

In [3]:
```
train_set = torchvision.datasets.FashionMNIST(
    root='./data'     #Lokacija gdje želimo spremati dohvaćene podatke
    ,train=True       #True ako je dataset za treniranje
    ,download=True    #True ukoliko bi dataset trebao biti skinut
    ,transform=transforms.Compose([
        #Kompozicija transformacija koje će se izvesti nad elementom dataseta
        #Ovdje želimo sve učitane slike pretvoriti u tensor objekte
        transforms.ToTensor()
    ])
)
```

Stvorimo Dataloader wrapper za učitani training set. Dataloader omata dataset te osigurava kasnije potrebnu funkcionalnost pristupa podatcima.

In [4]:
```
train_loader = torch.utils.data.DataLoader(train_set
    ,batch_size=1000
    ,shuffle=True
)
```

Pogledajmo najprije što je u train_set. Koja je duljina tog tenzora? Koji su labeli unutar njega? Je li dataset homogen - odnosno je li ima jednak broj slika svake klase?

In [5]:
```
print(len(train_set)) #Duljina seta
print(train_set.targets)  #Koji su labeli točne klasifikacije
print(train_set.targets.bincount()) # Provjerimo je li set balansiran
```

```
60000
tensor([9, 0, 0,  ..., 3, 0, 5])
tensor([6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000])
```

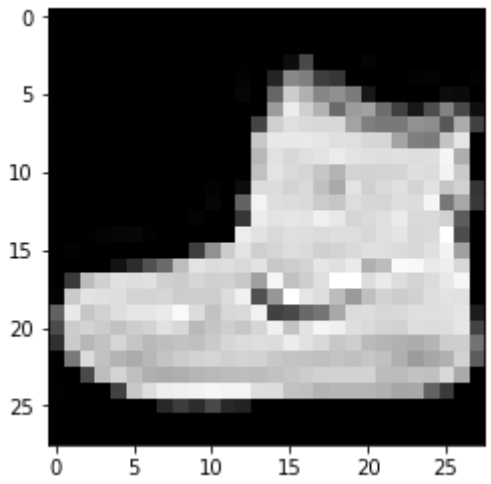
Zavirimo u prvu sliku iz seta. Pytonov način za dobiti prvi element iz nečega iterabilnog je općenito next(iter(neki_iterabilni_objekt)), gdje iter() dohvati iterator nad iterabilnim objektom, a next uzme sljedeći element iteratora. Zanima nas kako izgledaju pohranjeni podatci u datasetu i zato dobiveni uzorak *sample* razdvajamo najprije na njegove dvije komponente - sliku i label, a dalje doznamo da je slika tenzor dimenzija 1x28x28, a label obični integer gdje broj 9 označava da slika prikazuje gležnjaču. Sliku iscrtavamo pomoću biblioteke matplotlib.

In [6]:
```
sample = next(iter(train_set))

image, label = sample #Na prvom mjesto uzorka je slika, na drugom label
print('types:', type(image), type(label)) #Koje su vrste slika odnosno label
print('shapes:', image.shape, torch.tensor(label).shape) #Kojeg su oblika
#Label je samo broj (int) pa ga moramo najprije upakirati da bi doznali shape
print(image.squeeze().shape) #Primjetimo da squeeze izbaci jediničnu dimenziju

#Iscrtavanje uzorka pomoću matplotlib biblioteke
plt.imshow(image.squeeze(), cmap="gray")
print("Ova slika je klase: ", torch.tensor(label))
```

```
types: <class 'torch.Tensor'> <class 'int'>
shapes: torch.Size([1, 28, 28]) torch.Size([])
torch.Size([28, 28])
Ova slika je klase:  tensor(9)
```



Pogledajmo također kako izgleda jedan batch podataka. Batch podataka se odnosi na veću skupinu slika koje ćemo kasnije htjeti zajedno obraditi. Preciznije, govori broj slika koje koristimo u svakoj iteraciji pri treniranju ili testiranju modela mreže. Ovdje ćemo prikazati samo batch od 10 slika, a kasnije ćemo koristiti batch od po 1000 slika.

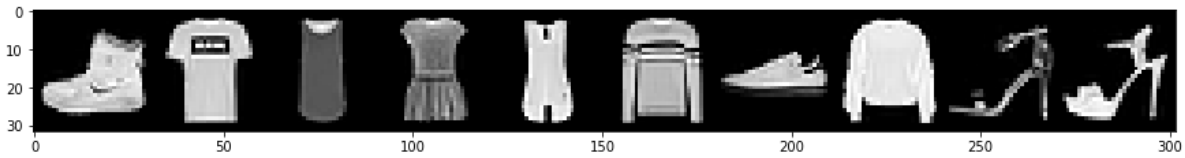
In [7]:
```
display_loader = torch.utils.data.DataLoader( train_set, batch_size=10 )
batch = next(iter(display_loader)) # uzmimo prvi batch podataka od loadera

images, labels = batch #Raspakirajmo batch na slike i labele
print('types:', type(images), type(labels)) #Tipovi slika i labela
print('shapes:', images.shape, labels.shape) #Oblici slika i labela
print(images[0].shape) # oblik prve slike
print(labels[0]) #Klasifikacija prve slike

#Stvorimo grid koji možemo iscrtati uz pomoć torchvision.utils
grid = torchvision.utils.make_grid(images, nrow=10)
plt.figure(figsize=(15,15))
plt.imshow(np.transpose(grid, (1,2,0)))

print('labels:', labels) #ispišimo i pripadajuće labele
```

```
types: <class 'torch.Tensor'> <class 'torch.Tensor'>
shapes: torch.Size([10, 1, 28, 28]) torch.Size([10])
torch.Size([1, 28, 28])
tensor(9)
labels: tensor([9, 0, 0, 3, 0, 2, 7, 2, 5, 5])
```



2. korak - modeliranje neuronske mreže

Nakon što smo vidjeli kako učitati potrebne podatke, krenimo na izgradnju neuronske mreže. Prvi korak u tome je izgradnja arhitekture, odnosno definiranje slojeva koji je sačinjavaju i definiranje propagacije unaprijed koju određuje metoda forward.

Sve neuronske mreže moraju nasljediti razred torch.nn.Module. Slojevi mreže također nasljeđuju taj razred. U ovom kodu koristimo dvije vrste slojeva - konvolucijski sloj modeliran razredom torch.nn.Conv2d i linearni tj. potpuno povezani sloj modeliran razredom torch.nn.Linear.

Pri definiranju propagacije unaprijed, bitan je pojam funkcija aktivacije. Funckije aktivacije određuju kako pojedinačni neuroni reagiraju na dospjele "podražaje", odnosno govore koliko će se neuron aktivirati za dospjele signale iz prethodnog sloja. Kao funkcija aktivacije se najčešće koriste ReLU i sigmoida.

In [8]:

```
class Network(nn.Module):
    def _init__(self):
        super(Network, self)._init__()
        #Stvorimo dva konvolucijska sloja
        #konvolucijski slojevi su modelirani razredom torch.nn.Conv2d
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)

        #Stvorimo dva linearna sloja
        #fc kao "fully connected layer", a ima i drugi naziv - "linear layer"
        #linearni slojevi su modelirani razredom torch.nn.Linear
        self.fc1 = nn.Linear(in_features=12 * 4 * 4, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=60)
        #Stvaranje izlaznog sloja - sloja gdje vidimo rezultate predikcije
        self.out = nn.Linear(in_features=60, out_features=10)

    def forward(self, t):
        # (1) input layer
        t = t

        # (2) hidden conv layer
        t = self.conv1(t)
        t = F.relu(t) #funkcija aktivacije
        t = F.max_pool2d(t, kernel_size=2, stride=2)

        # (3) hidden conv layer
        t = self.conv2(t)
        t = F.relu(t) #funkcija aktivacije
        t = F.max_pool2d(t, kernel_size=2, stride=2)

        # (4) hidden linear layer
        t = t.reshape(-1, 12 * 4 * 4)
        t = self.fc1(t)
        t = F.relu(t) #funkcija aktivacije

        # (5) hidden linear layer
        t = self.fc2(t)
        t = F.relu(t) #funkcija aktivacije

        # (6) output layer
        t = self.out(t)
        #t = F.softmax(t, dim=1)

        return t
```

Neke parametre koje smo koristili pri definiranju arhitekture smo sami odabrali,a njih nazivamo hiperparametri. Hiperparametri su parametri čije se vrijednosti biraju ručno i proizvoljno. Programer neuronskih mreža bira te vrijednosti uglavnom na temelju pokušaja i pogreške, promatrajući koji paramteri će dati bolje rezultate i vodeći se vrijednostima koje su se pokazale uspješnima u prošlosti. Za izgradnju arhitekture definirane konvolucijske mreže smo sada odabrali sljedeće hiperparametre: kernel_size, out_channels i out_features. Već smo ranije odabrali i hiperparametar veličinu batcha. Vrijednosti hiperparametara jednostavno sami odabiremo te testiramo i podešavamo kako bismo pronašli vrijednosti koje najbolje funkcioniraju.

Kratak opis spomenutih parametara:

- kernel_size - Postavlja veličinu filtra u korištenog u konvolucijskom sloju. Riječi kernel i filter su međusobno zamjenjive. kernel_size=5 odgovara filtru dimenzija 5x5
- out_channels - Postavlja broj filtara. Jedan filtar proizvodi jedan izlazni kanal.
- out_features - Postavlja veličinu izlaznog tensora.

Stvorimo instancu neuronske mreže te ju ispišimo. PyTorch osigurava da ispis modela neuronske mreže prikaže i opiše slojeve koji čine tu mrežu

In [9]:
```
network = Network()
print(network)
```

```
Network(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 12, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=192, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=60, bias=True)
  (out): Linear(in_features=60, out_features=10, bias=True)
)
```

3. korak - definirati funkciju treniranja

Treniranje se ugrubo odvija u ovim koracima:

1. Dohvati batch
2. Predaj dohvaćeni batch mreži
3. Izračunaj koliko je mreža pogriješila pomoću funkcije gubitka
4. Izračunaj gradijent funkcije gubitke s obzirom na parametre u slojevima mreža (težine)
5. Ažuriraj parametre mreže na temelju odgovarajućeg gradijenta i to tako da napraviš korak određene veličine u negativnom smjeru gradijenta
6. Ponavljaj korake 1-5 dok se ne dovrši jedna epoha, a epoha znači preći jednom preko svih podataka što su u datasetu za treniranje
7. Ponavljaj korake 1-6 dok se ne odradi zadani broj epoha

Pri definiranju funkcije treniranja ključni su sljedeći hiperparametri:

- funckija gubitka (loss) - funkcija koja za predikcije mreže očitane na izlaznom sloju kaže koliko je mreža pogriješila, da bi se mreža na temelju toga i korištenjem diferencijacije popravila. Neke funkcije gubitka su MSE (*Mean Square Error*), MAE (*Mean Absolute Error*), MBE (*Mean Bias Error*)
- funkcija optimizacije (optimizer) - da bi mreža učila, koristi propagaciju unazad - dakle od izlaznog sloja krene prema ulaznom sloju. Pritom na svaki parametar svakog sloja primjeni funkciju optimizacije kojom napravi korak određene veličine u negativnom smjeru gradijenta. Najkorištenije funkcije optimizacije su Adam (*Adaptive Moment Estimation*), SGD (*Stochastic gradient descent*) i RMSProp (*Root Mean Squared Propagation*)
- learning rate - početna veličina koraka koju mreža napravi u negativnom smjeru derivacije funkcije gubitka za svaki parametar koji se nalazi u težinama slojeva mreže kada se mreža optimizira. Veličina koraka se tijekom učenja prilagođava, točnije smanjuje kako se bliži minimumu funckije gubitka.
- broj epoha - koliko puta ćemo preći preko svih podataka iz dataseta

In [10]:
```
loss_f = nn.CrossEntropyLoss() # funkcija gubitka
optimizer_f = torch.optim.Adam # funkcija optimizacije
learning_rate = 0.005 # početna veličina koraka
epochs = 6 #broj epoha

def train():
    model = Network()
    optimizer = optimizer_f(model.parameters(), lr = learning_rate)
    criterion = loss_f
    for epoch in range(1, epochs):
        for batch_id, (image, label) in enumerate(train_loader):
            label, image = label, image
            output = model(image)
            loss = criterion(output, label)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if batch_id % 10 == 0:
                # print tek da vidimo kako teče treniranje
                print('Loss :{:.4f} Epoch[{}/{}]'.format(loss.item(),epoch,6))
    return model
```

4. korak - treniranje mreže

Pogledajmo najprije koje su predikcije neistrenirane mreže za prije izvađeni uzorak - sliku gležnjače koja ima label 9. Nakon što provučemo izlaz kroz softmax funckiju, dobit ćemo vjerojatnosti u postotcima. Primjetimo da su vjerojatnosti nasumično raspoređene jer su trenutni parametri mreže također nasumično uzeti pri inicijalizaciji mreže. Postotci s kreću oko 10% za sve klase, što je za očekivati jer je ukupno 10 klasa, a 1/10 = 0.1 --> 10%

In [11]:
```
pred = network(image.unsqueeze(0))
print(pred)
print(F.softmax(pred, dim=1))
```

```
tensor([[ 0.0153,  0.0589, -0.0193,  0.0642,  0.0707, -0.0489,  0.0202,  0.10
51,  0.0718,  0.0426]],
       grad_fn=<AddmmBackward>)
tensor([[0.0976, 0.1020, 0.0943, 0.1026, 0.1032, 0.0916, 0.0981, 0.1068, 0.10
33, 0.1004]], grad_fn=<SoftmaxBackward>)
```

Trenirajmo mrežu tako što pridodjelimo istreniranu mrežu prije definiranom modelu:

In [12]:
```
model = train()
```

```
Loss :2.3044 Epoch[1/6]
Loss :1.2073 Epoch[1/6]
Loss :1.0005 Epoch[1/6]
Loss :0.8542 Epoch[1/6]
Loss :0.7324 Epoch[1/6]
Loss :0.6961 Epoch[1/6]
Loss :0.7034 Epoch[2/6]
Loss :0.6200 Epoch[2/6]
Loss :0.6092 Epoch[2/6]
Loss :0.5755 Epoch[2/6]
Loss :0.5533 Epoch[2/6]
Loss :0.5620 Epoch[2/6]
Loss :0.5832 Epoch[3/6]
Loss :0.5129 Epoch[3/6]
Loss :0.5209 Epoch[3/6]
Loss :0.5013 Epoch[3/6]
Loss :0.5088 Epoch[3/6]
Loss :0.5054 Epoch[3/6]
Loss :0.4789 Epoch[4/6]
Loss :0.4548 Epoch[4/6]
Loss :0.5368 Epoch[4/6]
Loss :0.4341 Epoch[4/6]
Loss :0.4299 Epoch[4/6]
Loss :0.4149 Epoch[4/6]
Loss :0.4351 Epoch[5/6]
Loss :0.3945 Epoch[5/6]
Loss :0.4937 Epoch[5/6]
Loss :0.4481 Epoch[5/6]
Loss :0.3892 Epoch[5/6]
Loss :0.4328 Epoch[5/6]
```

Provjerimo koje predikcije istrenirane mreže na istu onu sliku gležnjače od prije. Primjetimo da je vjerojatnost koju sada predviđa mreža za gležnjaču značajno veća i točnija za label 9 koji odgovara klasi gležnjače te iznosi 99.561%

In [13]:
```
pred = model(image.unsqueeze(0))
print(pred)
print(F.softmax(pred, dim=1))
```

```
tensor([[ -1.4345, -11.6473,  -9.9104, -11.2662, -10.4171,   7.3630,  -6.235
6,   8.1757,   2.9816,  13.9716]],
       grad_fn=<AddmmBackward>)
tensor([[2.0291e-07, 7.4462e-12, 4.2290e-11, 1.0900e-11, 2.5480e-11, 1.3427e-
03, 1.6680e-09, 3.0266e-03, 1.6795e-05,
         9.9561e-01]], grad_fn=<SoftmaxBackward>)
```

5. korak - testiranje istrenirane mreže

Najprije dohvatimo dataset za testiranje, a potom definiramo funkciju koja će odraditi testiranje:

In [14]:
```
test_set = torchvision.datasets.FashionMNIST(
    root='./data'
    ,train=False
    ,download=True
    ,transform=transforms.Compose([
        transforms.ToTensor()
    ])
)

test_loader = torch.utils.data.DataLoader(train_set
    ,batch_size=1000
    ,shuffle=True
)
```

In [15]:
```
def test(model):
    with torch.no_grad():
        correct = 0
        total = 0
        for image, label in test_loader:
            image = image#.to(device)
            label = label#.to(device)
            outputs = model(image)
            predicted = torch.argmax(outputs,dim=1)
            total += label.size(0)
            correct += (predicted == label).sum().item()
        msg='Test Accuracy of the model on the test images: {} %'
        print(msg.format(100 * correct / total))
```

Testirajmo mrežu:

In [16]:
```
print("Testiranje krenulo...")
test(model)
```

```
Testiranje krenulo...
Test Accuracy of the model on the test images: 85.23333333333333 %
```

4. Zaključak

Za jednostavnije korištenje algoritama dubokog učenja stvorene su brojne Python biblioteke koje korisniku nude visoku efikasnost i vrlo jednostavnu primjenu algoritama. Te biblioteke pružaju funkcionalnost vektorske obrade podataka, učitavanja i obrade slika, automatsku diferencijaciju, podršku za GPU i iscrtavanje rezultata. Za GPU podršku je prije svega zaslužan CUDA API koji je razvila Nvidia i kojim je omogućila jednostavno korištenje jezgara grafičke procesorske jedinice za provođenje proračuna opće namjene kao što je paraleliziranje zadataka u dubokom učenju.

Od samog projektiranja i izgradnje arhitekture umjetne neuronske mreže, mnogo je teže pronaći i prikupiti odgovarajući skup podataka na kojem bi se modelirana mreža trenirala. Na internetu su dostupni skupovi podataka poput MNIST-a i FashionMNIST-a koje programeri često koriste pri vježbanju dubokog učenja, a znanstvenici kao benchmark osmišljenih arhitektura neuronskih mreža.

5. Sažetak

U ovom seminarskom radu opisane su najpoznatije biblioteke za ostvarivanje dubokog učenja korištene u programskom jeziku Python.

U Pythonu je programska podrška za duboko učenje osobito dobro razvijena. Stvorene su biblioteke koje osiguravaju visoke performanse izvođenja operacija vektorske obrade podataka, učitavanja i obrade slika, automatske diferencijacije i jednostavnog iscrtavanja rezultata te biblioteke koje pružaju GPU podršku. Među najkorištenijim i najpopularnijim bibliotekama za spomenute stvari su:

- NumPy za vektorsku obradu podataka
- libjpeg-turbo i pillow-simd za učitavanje i obradu slika
- PyTorch za automatsku diferencijaciju
- cuDNN za GPU podršku
- matplotlib za iscrtavanje rezultata

Preciznije je reći da je PyTorch programski okvir (framework) za duboko učenje koji sadrži sve potrebno za jednostavnu implementaciju algoritama dubokog učenja uz akceleraciju računanja korištenjem grafičke procesorske jedinice.

Pored programskog koda važni su skupovi podataka s kojima neuronska mreža radi. Jednostavni program od 200 linija koda koji je opisan u seminaru koristi FashionMNIST skup podataka da bi trenirao jednostavnu konvolucijsku neuronsku mrežu.

6. Literatura

http://www.zemris.fer.hr/~ssegvic/du/
http://www.zemris.fer.hr/~ssegvic/project/pubs/kralj16sem.pdf
https://www.python.org/
https://github.com/numpy/numpy
https://github.com/zalandoresearch/fashion-mnist

https://pytorch.org/
http://deeplizard.com/
https://arxiv.org/pdf/1102.1523.pdf
https://docs.fast.ai/performance.html
https://matplotlib.org