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.

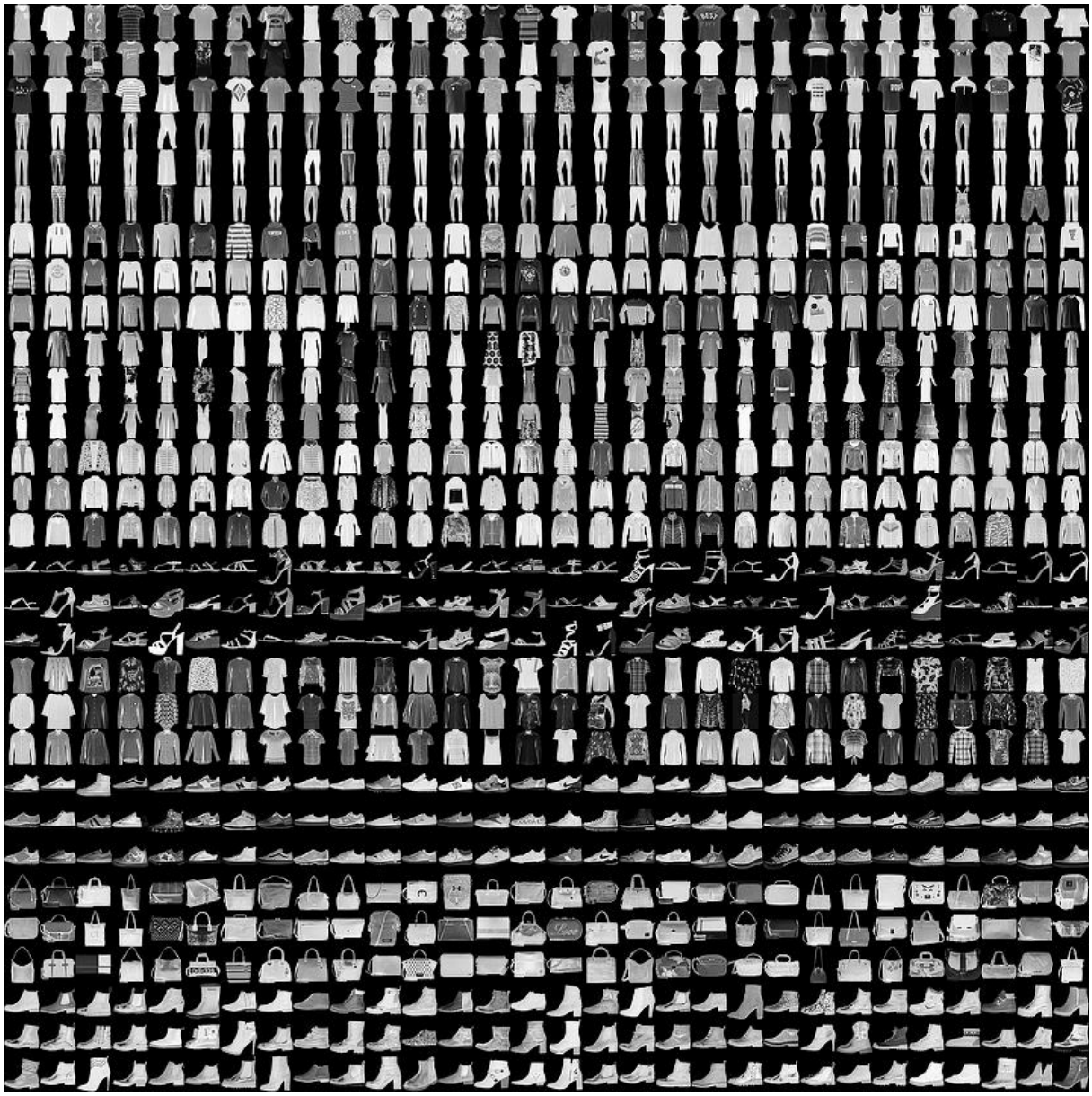
Prije nego nastavim na programski kod minimalističkog programa, opisat ću kratko MNIST i FashionMNIST skup podataka jer će podatke iz FashionMNISU-a koristiti neuronska mreža iz minimalističkog programa za treniranje. Skup podataka MNIST (*Modified National Institute of Standards and Technology database*) 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])
```

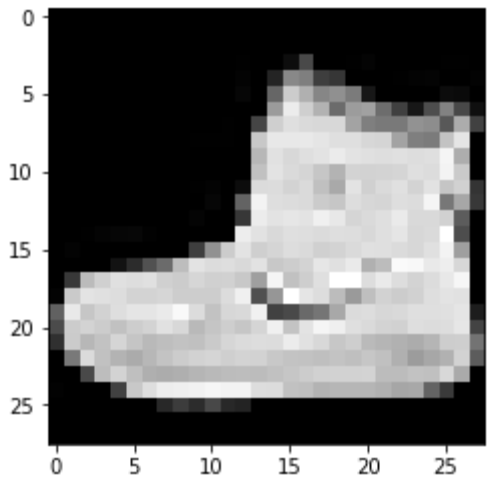
Zavirimo u prvu sliku iz seta. Pytonov način za dobiti prvi element iz nečega iterabilnog je općenito next(iter(neki_iterabilni_objekt)), gdje iter() dohvati iterator nad iterabilnim objektom, a next uzme sljedeći element iteratora. Zanima nas kako izgledaju pohranjeni podatci u datasetu i zato dobiveni 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.

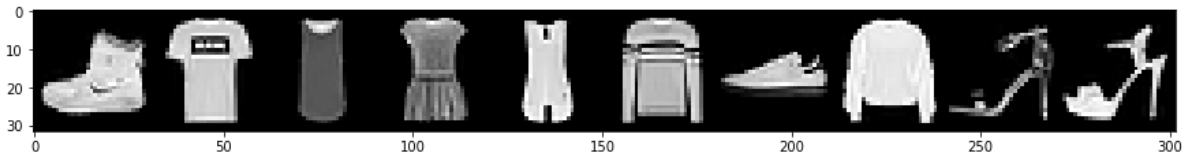
In [7]:
```
display_loader = torch.utils.data.DataLoader( train_set, batch_size=10 )
batch = next(iter(display_loader)) # uzmimo prvi batch podataka od loadera

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 %
```