SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Oblikovni obrasci u programskom okviru PyTorch

Petra Kelković

Voditelj: *prof.dr.sc. Siniša Šegvić*

Zagreb, lipanj, 2025.

Pregled poglavlja

1. Uvod

Oblikovni obrasci predstavljaju provjerena rješenja za česte probleme u razvoju softverskih sustava. Umjesto točnih uputa kako implementirati neko rješenje, oni nude strukturalne i konceptualne smjernice koje pomažu pri organizaciji koda, olakšavaju održavanje i omogućuju veću fleksibilnost sustava.

PyTorch je popularan programski okvir za duboko učenje koji podržava dinamičko definiranje modela, rad s kompleksnim strukturama podataka te jednostavnu integraciju vlastitih komponenti. U takvom okruženju jasno definirani obrasci dizajna olakšavaju izradu modularnih i čitljivih rješenja, posebice u slučajevima kada se grade složene arhitekture ili uvode vlastite apstrakcije.

U ovom seminarskom radu fokus je na stavljen na slijedećih šest obrazaca: Kompozit, Okvirna metoda, Strategija, Iterator, Prilagodnik i Tvornica. Tih šest obrasca posebno dolaze do izražaja u dijelovima PyTorcha koji se najviše koriste u praksi. Njihova je prisutnost najvidljivija upravo ondje gdje se modeli definiraju, treniraju i evaluiraju, što ih čini ključnim za razumijevanje načina na koji PyTorch funkcionira. Svaki od njih ima svoje mjesto u okviru PyTorcha — od organizacije slojeva unutar modela, preko odabira i kreacije optimizacijskih pristupa, pa do iteracije nad skupovima podataka bez potrebe za poznavanjem implementacijskih detalja komponenti i kreacije konkrentih objekata modela neuronskih mreža na uniforman način. Ovaj rad kroz primjere prikazuje kako se ovi obrasci uklapaju u dizajn PyTorch okvira te kako ih programer može iskoristiti za izgradnju jasnijih i kvalitetnijih rješenja.

2. Oblikovni obrazac Kompozit: klase nn.Sequential i torchvision.transforms.Compose

Oblikovni obrazac Kompozit jest strukturni obrazac koji omogućuje da se grupom objekata rukuje na jednak način kao i pojedinačnim objektom. Koristi se kada želimo implementirati strukturu objekata sličnu stablu ili kada želimo koristiti jednostavne i kompleksne objekte na uniforman način.

Sudionici ovog obrasca su Komponenta, Primitiv, Kompozit i Klijent.

PyTorch okvir primjenjuje ovaj obrazac u klasi nn.Module, baznoj klasi koja predstavlja sve module neuronskih mreža. U PyTorchu, modeli neuronskih mreža sastoje se od slojeva (nn.Linear, nn.ReLU, itd.), a slojevi su sami nn.Module objekti. Ovime se omogućuje da jedan modul (nn.Module) sadrži druge module kao podmodule, što znači da cijeli model i njegovi dijelovi imaju zajedničko sučelje.
Budući da moduli mogu sadržavati druge module (koji su također nn.Module), moguće je graditi hijerarhijske strukture proizvoljne dubine. To uvelike olakšava definiciju složenih mreža, primjenu rekurzivnog pristupa, učitavanje/spremanje stanja…

Osim u modelima neuronskih mreža, ovaj obrazac također se može prepoznati u načinu izvedbe klase torchvision.transforms.Compose. Klasa Compose omogućava kombinaciju više transformacija koje se zatim mogu primijeniti nad slikom ili podatcima.

Primjeri korištenja obrasca Kompozit u PyTorchu

Ulogu **Komponente** u primjeru s modelima neuronskih mreža ima upravo nn.Module, što je vidljivo iz činjenice da ju nasljeđuju svi moduli neuronskih mreža, bili oni Kompoziti ili Primitivi.

U primjeru s kombiniranjem transformacija, uloga **Komponente** je implicitna i ne postoji zajednička klasa koju sve transformacije i kompozit moraju naslijediti. Svaka komponenta mora imati implementirane sljedeće dvije metode: metodu *__call__(self, img),* koja preko argumenata prima sliku nad kojom obavlja transformaciju, i metodu *__repr__(self),* koja vraća znakovni niz koji predstavlja taj objekt.

Ulogu **Primitiva** u modelima neuronskih mreža imaju razne klase koje nasljeđuju nn.Module, od kojih su neke nn.Linear, nn.ReLu, nn.Conv2d. Svaka od ovih klasa nasljeđuje klasu nn.Module uz specifičnu namjenu koja joj je dodijeljena. Primjerice, nn.Linear je klasa koja nasljeđuje klasu Module i specijalizirana je za računanje afine linearne transformacije nad ulaznim podacima.

U primjeru s kombiniranjem transformacija, ulogu **Primitiva** imaju konkretne izvedbe transformacija koje implementiraju već spomenute funkcije *__call__()* i *__repr__().* To su primjerice klase torchvision.transforms.PILToTensor, koja pretvara PIL (*Python Imaging*

Library) sliku u tenzor jednakog tipa, klasa torchvision.transforms.CenterCrop, koja služi za izrezivanje centralnog dijela slike (ili tenzora), i klasa torchvision.transforms.Normalize, koja provodi normalizaciju danog tenzora slike sa zadanom srednjom vrijednosti i standardnom devijacijom.

U modelima neuronskih mreža, ulogu **Kompozita** može poprimiti modificirana instanca klase nn.Module. Klasa nije formalno apstraktna i može se instancirati ako joj se nadjača metoda forward(). Ona sadrži funkciju za dodavanje djece modula u objekt sa sljedećom deklaracijom: *add_module(name, module).* S obzirom na postojanje ovakve funkcije, jasno je da objekti klase nn.Module imaju mogućnost postati Kompoziti. Osim nn.Module, ulogu Kompozita ima klasa nn.Sequential koja nema drugih uloga u sustavu. Klasa nn.Sequential prima u konstruktoru listu objekata nn.Module od kojih će se sastojati te ne može postojati kao neovisan objekt.
U kombiniranju transformacija, ulogu kompozita ima klasa torchvision.transforms.Compose. Ona u konstruktoru prima listu transformacija od kojih će se sastojati. Njezina uloga jest isključivo omatanje liste transformacija koje će se redom primjenjivati nad slikama ili podatcima.

```
class Sequential(Module):
    @overload
    def __init__(self, *args: Module) -> None:
        ...

    @overload
    def __init__(self, arg: "OrderedDict[str, Module]") -> None:
        ...


  def forward(self, input):
        for module in self:
            input = module(input)
        return input
```

Kôd 1: isječak izvornog koda klase nn.Sequential programskog okvira PyTorch

U kôdu 1 mogu se vidjeti jedina dva oblika konstruktora koje klasa nn.Sequential ima te je vidljivo da obje inačice primaju skup nn.Module objekata. Metoda *forward* klase nn.Sequential daje dobar primjer „klasične“ kompozitne metode. U njoj je vidljivo da je objekt tipa nn.Sequential izvođenje vlastite funkcije delegira podmodulima iterativno, što je tipično ponašanje za funkcije unutar Kompozita.

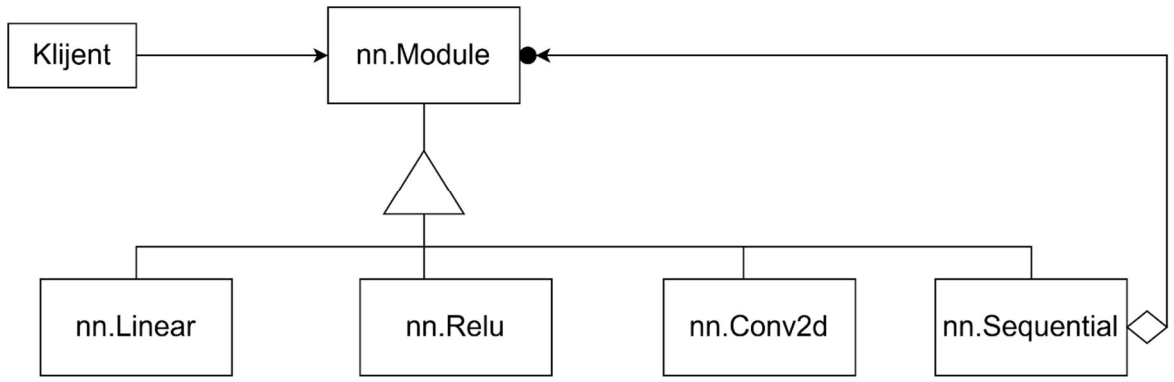
```
class Compose:
    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, img):
        for t in self.transforms:
```

```
            img = t(img)
        return img

    def __repr__(self):
        format_string = self.__class__.__name__ + '('
        for t in self.transforms:
            format_string += '\n'
            format_string += '    {0}'.format(t)
        format_string += '\n)'
        return format_string
```

Kôd 2: izvorni kod klase torchvision.transforms.Compose programskog okvira PyTorch

U kôdu 2 može se vidjeti da klasa Compose u konstruktoru prima listu transformacija *transforms* te da ne postoje verzije konstruktora koje bi kreirale objekt klase Compose bez liste transformacija. Također, u obje metode koje klasa Compose implementira mogu se vidjeti osobine „klasičnih" kompozitnih metoda – obje metode se sastoje od *for* petlje koja ide po svim transformacijama u listi i redom poziva njihove implementacije tih metoda.

Ulogu **Klijenta** preuzima korisnik okvira koji primjerice može koristiti model neuronskih mreža i pozivati metode poput metode *forward()* ili koji može koristiti kompoziciju transformacija i pozivati ju nad željenom slikom. Klijentu je omogućeno da po želji stvara objektnu hijerarhiju strukture stabla korištenjem nn.Module objekata. Klijent također može po želji stvarati objektnu hijerarhiju strukture stabla ugnježđivanjem objekata Compose, no to mu neće donijeti nikakve prednosti te se struktura transformacija najčešće zadržava u obliku liste.
Dijagram navedenih odnosa klasa modula neuronskih mreža prikazan je na Slici 1.



Slika 1: Dijagram odnosa klasa modula neuronskih mreža koji sudjeluju u OO Kompozit

Dijagram odnosa između klasa koje sudjeluju u kompoziciji transformacija teže je prikazati iz razloga što je uloga Komponente u sustavu implicitna, tj. ne postoji zajednička klasa koju bi i Kompozit i Primitivi morali naslijediti. Ako zamislimo zajedničku klasu kao skup funkcija koje transformacijske klase moraju implementirati, onda bi klasni dijagram izgledao jednako kao i dijagram klasa modula neuronskih mreža, u kojem bi ulogu Komponente imao taj skup funkcija koje su nužne za implementaciju transformacija, a to su funkcije *__call__()* i *__repr__()*.

Primjer korištenja klasa za modeliranje neuronskih mreža slijedi u nastavku.

```
import torch.nn as nn
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
model.train()
```

Kôd 3: jednostavan primjer korištenja obrasca Kompozit u modeliranju neuronskih mreža u programskom okviru PyTorch

Ovaj primjer demonstrira kako se korištenjem klase nn.Sequential može implementirati proizvoljna neuronska mreža čiji slojevi su također instance klase nn.Module. Ovaj model može koristiti iste funkcije koje bilo koja druga instanca klase nn.Module može koristiti, poput funkcije *train()*.

Primjer korištenja kompozicije transformacija nad slikom slijedi u nastavku.

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

image = Image.open('putanja/do/slike.jpg')
image = transform(image)
```

Kôd 4: jednostavan primjer korištenja obrasca Kompozit u kombiniranju transformacija u programskom okviruPyTorch

Kôd 4 demonstrira kako se korištenjem klase Compose može stvoriti kompozicija transformacija koje će se slijedno primjenjivati nad predanim objektom. U ovom slučaju, kompozit se sastoji od transformacija Resize, ToTensor i Normalize, a nakon toga se dobiveni objekt poziva imenom. U pozivu objekta predaje se objekt nad kojim će se transformacija obavljati, što je u ovom slučaju slika.

Posljedice korištenja obrasca Kompozit u oblikovanju odnosa između klasa su:

- Jedinstveno sučelje svih modula (jednostavnih i složenih) - svi imaju metode *forward, parameters, train, eval,…*
- Jednostavno dodavanje novih slojeva modulima
- Rekurzivna pohrana i učitavanje – funkcije poput *state_dict()* i *load_state_dict()* automatski pokrivaju i podmodule
- Jednostavan način stvaranja kombiniranih transformacija koje se pozivaju na isti način kao i sama transformacija

Primjena obrasca Kompozit povećava fleksibilnost i omogućuje primjenu Načela nadomjestivosti osnovnih razreda (*Liskov Substitution Principle*), budući da svi moduli dijele

isto sučelje bez obzira na složenost njihove unutarnje strukture, a sve transformacije implementiraju isti skup funkcija s pomoću kojih ih se jednostavno koristi.

3. Oblikovni obrazac Okvirna metoda: metoda *__call__()* klase nn.Module

Oblikovni obrazac Okvirna metoda jest ponašajni obrazac koji omogućuje definiranje kostura algoritma u nadređenoj klasi, dok konkretne korake algoritma prepušta izvedbenim klasama. Tako se omogućuje da različiti podrazredi implementiraju pojedine faze algoritma bez mijenjanja strukture algoritma kao cjeline.

Sudionici ovog obrasca su Apstraktni Razred i Konkretni razred.

U okviru PyTorch, ovaj obrazac se koristi u metodi *__call__()* u klasi nn.Module. Općenito, u programskom jeziku Python metoda *__call__()* omogućuje objektima da se mogu pozvati poput običnih funkcija. Kada korisnik stvori objekt klase koja ima definiranu metodu *__call__()* i tu instancu pozove na isti način na koji poziva funkcije, u pozadini se pokreće metoda *__call__()* nad tim objektom.

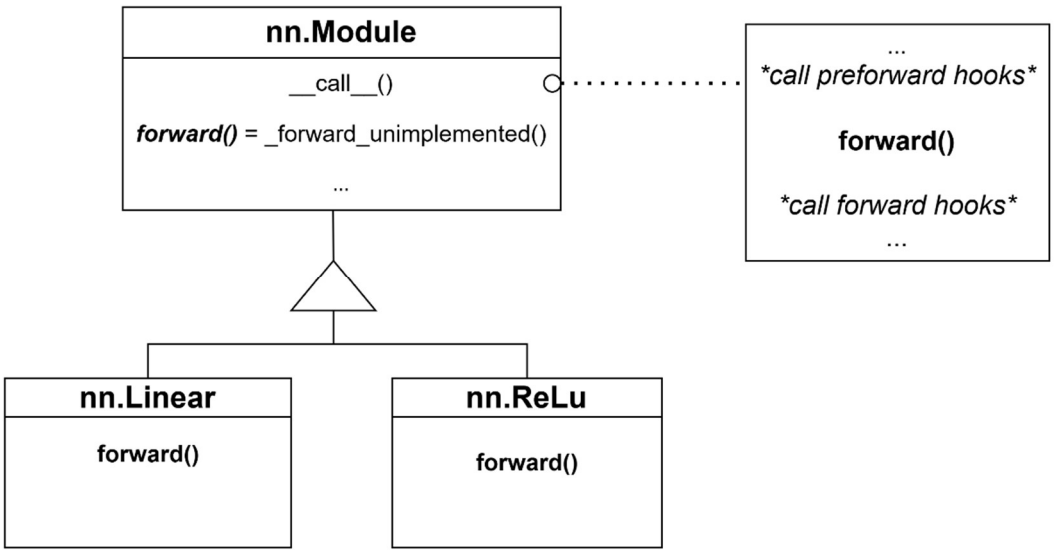
Bazna klasa neuronskih mreža nn.Module koristi obrazac Okvirna metoda upravo pri definiciji te metode. Tako klasa nn.Module pruža svojim nasljednicima okvirne korake koji se moraju provesti prilikom poziva funkcije *__call__(),* a konkretne izvedbe pojedinih koraka prepušta njima. Najvažniji korak koji svaka klasa izvedena iz nn.Module mora implementirati jest metoda *forward(),* koja određuje kako se od ulaznih podataka dobivaju izlazni podaci.

Primjer korištenja obrasca Okvirna metoda u PyTorchu

Ulogu **Apstraktne Klase** ima klasa nn.Module, koja definira strukturu okvirne metode *__call__().* Okvirna metoda najprije poziva *preforward hooks*, tj. funkcije koje su registrirane da ih se pozove točno prije pozivanja metode *forward(),* nakon čega se poziva sama metoda *forward(),* a zatim se pozivaju *forward hooks*, funkcije koje su registrirane da ih se pozove nakon što je metoda *forward* završila. *Hook* funkcije slične su promatračima iz OO Promatrač, no razlika je ta što se one ne pozivaju kada dođe do promjene stanja objekta, već u točno definiranom trenutku u tijeku izvođenja, što je u ovom slučaju prije i nakon izvršavanja metode *forward*. Klasa nn.Module ima definirane sve potrebne funkcionalnosti za upravljanje *hookovima*, što tim funkcionalnostima daje ulogu nadomjesnih metoda. S druge strane, klasa nn.Module ne pruža implementaciju metode *forward*, stoga je ona apstraktni primitiv.

Ulogu **Konkretnog razreda** preuzimaju svi razredi koji nasljeđuju klasu nn.Module. Svaki razred koji naslijedi klasu nn.Module ima obavezu definirati vlastitu implementaciju metode *forward(),* osim ako i on sam ima ulogu apstraktnog razreda. Konkretni nazivi razreda koji nasljeđuju klasu nn.Module spomenuti su u prošlom poglavlju, pa ih ovdje nećemo spominjati.

Dijagram navedenih odnosa može se vidjeti na Slici 2. Implementacija metode *__call__* napisana je u pseudokodu koji demonstrira redoslijed obavljanja koraka unutar te metode.



Slika 2: Dijagram odnosa PyTorch razreda koji sudjeluju u OO Okvirna metoda

Programski primjer definicije klase vlastitog modela slijedi u nastavku.

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.linear1 = nn.Linear(784, 128)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x

model = MyModel()
output = model(input_data)  # poziva model.__call__()
```

Kôd 5: Primjer definicije klase modela i korištenja okvirne metode __call__()

U ovom primjeru, definirana je klasa MyModel koja nasljeđuje klasu nn.Module i koja definira vlastitu implementaciju metode *forward*. Metoda *forward* u ovom primjeru implementira jednostavnu neuronsku mrežu koja daje svakom pojedinom sloju da obradi ulazne podatke, a zatim vraća izlaz zadnjeg sloja. Klasa se koristi tako da se najprije instancira njen objekt koji se pohranjuje u varijablu *model*, a zatim se varijabla *model* poziva s ulaznim podacima kao argumentom. Poziv objekta interno se prevodi u poziv okvirne metode *__call__()* klase nn.Module, koja poziva implementaciju metode *forward* klase MyModel. Metoda *forward* vraća izlazne podatke koji se spremaju u varijablu *output*.

Posljedice korištenja obrasca Okvirna metoda u PyTorchu su:

- Lako se dodaju nove izvedbe konkretnih klasa neuronskih mreža, što je sukladno Načelu nadogradnje bez promjene (NNBP) .
- Konkretne klase neuronskih mreža na jedostavan način mogu izmjeniti ponašanje izvršavanja modela, a da se ne moraju brinuti o upravljanju *hookovima.*
- Sve klase neuronskih mreža imaju jedinstvenu točku kroz koju se pokreće izvršavanje modela neovisno o njegovoj konkretnoj strukturi, što pojednostavljuje korištenje i testiranje modela.

Obrazac Okvirna metoda u PyTorchu omogućuje elegantno razdvajanje mehanizma od ponašanja, gdje korisnik ima punu kontrolu nad protokom podataka (kroz `forward()`), dok okvir brine o svemu ostalom.

4. Oblikovni obrazac Strategija: klase torch.nn.modules.loss._Loss i torch.optim.Optimizer

Oblikovni obrazac Strategija jest ponašajni obrazac koji omogućuje izmjenu ponašanja algoritma neovisno o klijentima koji te algoritma pozivaju. Umjesto da se određeno ponašanje ugrađuje u klasu putem grananja ili nasljeđivanja, Strategija omogućuje delegiranje ponašanja drugoj klasi. Na taj način mijenjanje ponašanja (tj. algoritma) ne zahtijeva izmjene izvornog koda osnovne klase.

Sudionici ovog obrasca su Kontekst, Strategija i Konkretna strategija.

U programskom okviru PyTorch, obrazac Strategija može se uočiti u implementaciji različitih funkcija gubitka (loss functions) i optimizacijskih algoritama (optimizers). Funkcija gubitka predstavlja Strategiju koja određuje kako model mjeri pogrešku, dok optimizator predstavlja Strategiju ažuriranja parametara modela.

Primjeri korištenja obrasca Strategija u PyTorchu

Ulogu **Strategije** u okviru PyTorch imaju bazne klase za odabir algoritma optimizacije i za odabir algoritma računanja gubitka. Klasa za odabir algoritma za računanje gubitka naziva se torch.nn.modules.loss._Loss, a klasa za odabir algoritma optimizacije naziva se torch.optim.Optimizer. Obje klase delegiraju posao definiranja osnovnih metoda Konkretnim strategijama. Klasa Loss ne definira osnovnu metodu *forward()* za algoritme računanja gubitka, nego očekuje da ju implementiraju klase koje ju naslijede. Isto tako, klasa Optimizer ima praznu varijantu funkcije *step(),* ključne funkcije za algoritme optimizacije, koja svojim pozivom vraća *NotImplementedError* grešku.

Ulogu **Konkretnih strategija** preuzimaju pojedinačne klasne implementacije funkcija gubitka i optimizatora. Na primjer, klase torch.nn.MSELoss i torch.nn.CrossEntropyLoss su Konkretne strategije za Strategiju torch.nn.modules.loss._Loss koje računaju gubitak na temelju srednje kvadratne pogreške i unakrsne entropije. Za Strategiju torch.optim.Optimizer možemo navesti Konkretne strategije torch.optim.SGD, koja koristi stohastički gradijentni spust i torch.optim.Adam, koja koristi Adam algoritam stohastičke optimizacije. Korisnik može u svakom trenutku zamijeniti jednu funkciju gubitka drugom ili zamijeniti algoritam optimizacije bez potrebe za izmjenom same strukture funkcija koje ih koriste.

Ulogu **Konteksta** u ovom obrascu preuzimaju funkcije klase nn.Module, za koje se u *main* funkciji odabire koju će Konkretnu strategiju upotrijebiti. Na primjer, funkcija za treniranje (*train()*) najčešće kroz argument prima Konkretnu strategiju optimizacije koju koristi. Time je postignuto da funkcije koje koriste optimizatore ne ovisi o konkretnim implementacijama optimizacijskih strategija, već koriste njihovo zajedničko sučelje koje definira klasa torch.optim.Optimizer.

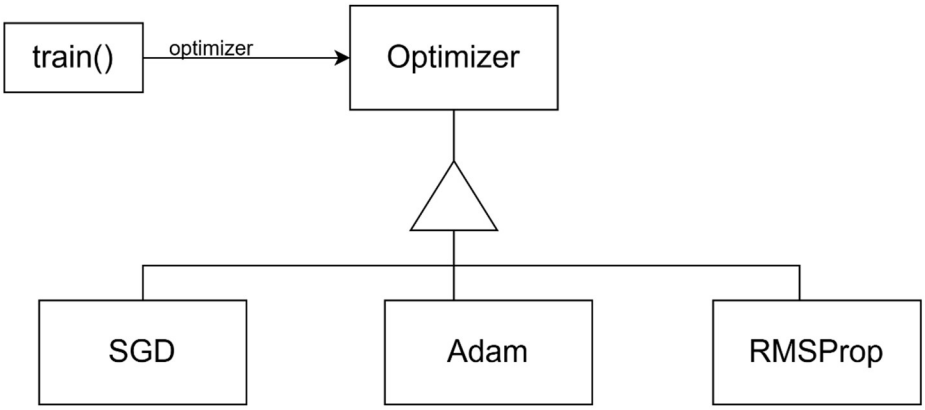
```
def train(model, data, optimizer, criterion, args):
  model.train()
  for batch_num, batch in enumerate(data):
    model.zero_grad()
```

```
    # ...
    logits = model(x)
    loss = criterion(logits, y)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), args.clip)
    optimizer.step()
    # ...

def evaluate(model, data, criterion, args):
  model.eval()
  with torch.no_grad():
    for batch_num, batch in enumerate(data):
      # ...
      logits = model(x)
      loss = criterion(logits, y)
      # ...

def main(args):
  seed = args.seed
  np.random.seed(seed)
  torch.manual_seed(seed)

  train_dataset, valid_dataset, test_dataset = load_dataset(...)
  model = initialize_model(args, ...)

  criterion = nn.BCEWithLogitsLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

  for epoch in range(args.epochs):
    train(…, …, optimizer, criterion, …)
    evaluate(…, …, criterion, …)
```

Kôd 6: Primjer struktura funkcija za treniranje i evaluaciju modela, uz primjer korištenja

U ovom programskom primjeru možemo vidjeti da funkcija *train()* prima argument *optimizer*, čija konkretna implementacija joj nije poznata, ali zna da objekt *optimizer* sigurno ima implementiranu funkciju *step*() koju može pozvati. Slično vrijedi i za argument *criterion*. Funkcija *train*() zna da može pozvati funkciju *criterion()* s argumentima *logits* i *y* iako ne zna koja je konkretna implementacija funkcije korištena.

Konkretne strategije koje će funkcija *train* koristiti za optimizaciju i za izračun gubitka zadaju se u funkciji *main*. U ovom primjeru koristi se upravo klasa Adam za optimizacijsku strategiju, dok se za strategiju računanja gubitka koristi klasa BCEWithLogitsLoss koja koristi kombinaciju sigmoidne aktivacijske funkcije i *Binary Cross Entropy* gubitka. Dijagram komponenti navedenih u primjeru prikazan je u nastavku.



Slika 3: Dijagram odnosa PyTorch razreda i funkcija koje sudjeluju u obrascu Strategija

Konkretniji primjer korištenja Strategija u PyTorchu slijedi u nastavku:

```
import torch
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(10, 1)          # jednostavni model

loss_fn = nn.MSELoss()             # strategija gubitka

optimizer = optim.SGD(model.parameters(), lr=0.01) # str. opt.

for input, target in data_loader:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

Kôd 7: Primjer korištenja obrazaca Strategija u PyTorchu

U ovom primjeru, funkcija gubitka MSELoss i optimizacijski algoritam SGD su zamjenjive strategije koje korisnik može prilagoditi prema potrebi. Na primjer, promjena iz MSELoss u CrossEntropyLoss ili iz SGD u Adam ne bi zahtijevala izmjenu drugih dijelova koda osim funkcije main gdje se te strategije zadaju.

(Pozitivne) posljedice korištenja obrasca Strategija u PyTorchu:

- Poštivanje Načela nadogradnje bez promjene (NNBP)– moguće je definirati vlastite funkcije gubitka i optimizatore nasljeđivanjem osnovnog sučelja.
- Primjena Načela nadomjestivosti osnovnih razreda (Liskov Substitution Principle) – sve funkcije gubitka i optimizatori mogu se koristiti zamjenjivo jer implementiraju zajedničko sučelje.
- Povećana ortogonalnost – klase neuronskih mreža ne ovise o konkretnim izvedbama postupaka optimizacije / računanja gubitka.

5. Oblikovni obrasci Iterator: klase Dataset, DataLoader i _BaseDataLoaderIter

Oblikovni obrazac Iterator pripada skupini ponašajnih obrazaca i omogućuje sekvencijalni pristup elementima kolekcije bez otkrivanja unutarnje reprezentacije te kolekcije. Svrha ovog obrasca je omogućiti klijentskom kôdu da prolazi kroz elemente kompleksne strukture na uniforman način, bez brige o tome kako su ti elementi strukturirani ili dohvaćeni. Također, iterator omogućuje višestruko obilaženje kolekcije u isto vrijeme.

Sudionici ovog obrasca su Iterator, Konkretni iterator, Apstraktni skupni objekt i Konkretni skupni objekt.

S obzirom na to da se funkcije PyTorch okvira vrlo često bave obradom podataka, PyTorch je osmislio dva primitiva koja olakšavaju rukovanje podatcima. Ta dva primitiva su torch.utils.data.DataLoader i torch.utils.data.Dataset, klase koje se bave pohranom i iteriranjem po podacima. Klasa Dataset služi za pohranu uzoraka i njihovih oznaka, dok klasa DataLoader ima ulogu iterabilnog objekta koju se omotava oko Dataseta kako bi omogućio pristup podacima na jednostavan i uniforman način.

Primjer korištenja obrasca Iterator u PyTorchu

Ulogu **Apstraktnog skupnog objekta** ima klasa torch.utils.data.Dataset. Klasa Dataset je apstraktna klasa koja se koristi za pohranu podataka, iz čega je odmah lako zaključiti njezinu ulogu u obrascu. Sve klase koje nasljeđuju Dataset moraju implementirati funkciju *__getitem__* koja za dani ključ podatka vraća sam podatak, a preporučuje se implementacija funkcije *__len__,* koja vraća broj podataka u setu.

Ulogu **Konkretnog skupnog objekta** imaju razne klase koje su naslijedile klasu Dataset, a koje dolaze u *preloadanom* stanju, što znači da ih je PyTorch već ispunio podatcima nad kojima programeri mogu trenirati svoje modele. Primjeri takvih klasa su mogu se pronaći u paketu torchvision.dataset, a to su primjerice torchvision.dataset.ImageNet, torchvision.dataset.Country211 i torchvision.dataset.FER2013 :)

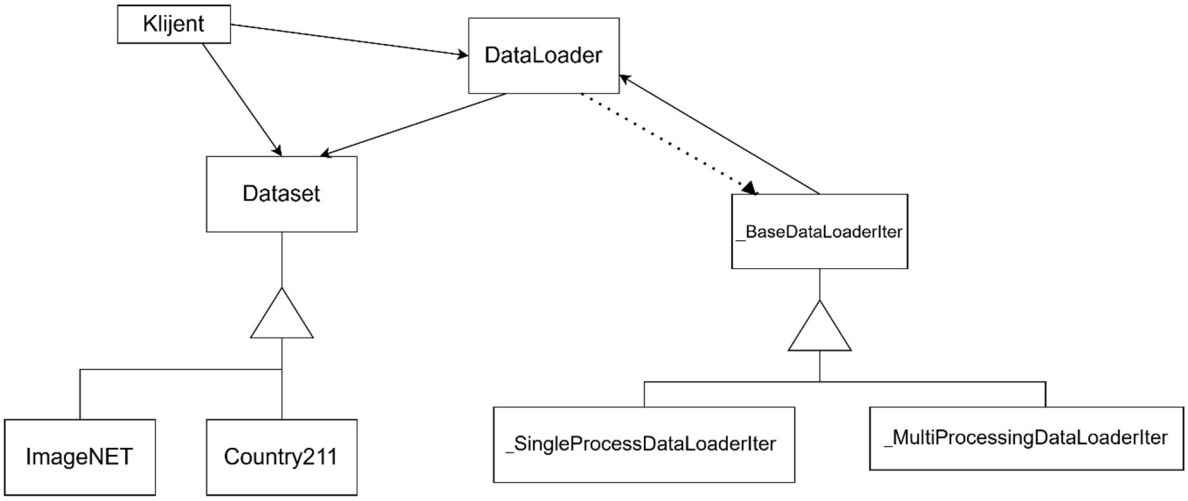
Klasa DataLoader u okviru PyTorch nema ulogu Iteratora, iako se vrlo često koristi upravo za iteriranje po skupu podataka. Naime, sama klasa nema metode koje iteratori uobičajeno implementiraju (npr. metodu *next*()), a poziv funkcije *__get_iterator* vraća objekt apstraktne klase _BaseDataLoaderIter. Iz tog razloga je klasa DataLoader smatrana iterabilnim omotačem nad objektom klase Dataset – ona vraća sve što je korisniku potrebno za iteriranje po skupu podataka, a da sama *nije* iterator. DataLoader na ovaj način stvara dodatan sloj između korisnika i samog iteratora, skrivajući unutarnju implementaciju iteratorskih funkcija, što je upravo glavna uloga sudionika prilagodnik u OO Prilagodnik. Klasa DataLoader i njezina uloga prilagodnika u programskom okviru PyTorch detaljno je objašnjena u poglavlju 6. Oblikovni obrazac Prilagodnik.

Dakle, ulogu **Iteratora** u PyTorchu ima apstraktna klasa _BaseDataLoaderIter. Iako ova klasa nije apstraktna u pravom smislu riječi (nema apstraktne metode niti ne nasljeđuje baznu apstraktnu klasu abc.ABC, no to gotovo niti jedna klasa u Pythonu nema), ona je smatrana

apstraktnom zato što nema implementirane funkcije poput *_next_data*(), već poziv te funkcije baca *NotImplementedError* grešku. Klasa _BaseDataLoaderIter koristi se kao bazna klasa za sve konkretne DataLoader iteratore. Ona ima deklarirane funkcije poput *__next__*, *__iter__* i *__len__* koje karakteriziraju iteratore u programskom jeziku Python, što dokazuje njezinu ulogu Iteratora u objektnom obrascu Iterator.

Ulogu **Konkretnih iteratora** imaju dvije klase koje nasljeđuju _BaseDataLoaderIter, a to su _SingleProcessDataLoaderIter i _MultiProcessingDataLoaderIter. _SingleProcessDataLoaderIter klasa se koristi kada se učitavanje i obrada podataka odvija u glavnom procesu PyTorcha, a _MultiProcessingDataLoaderIter se koristi kada postoje dodatni procesi radnici koji preuzimaju zadatak učitavanje i obrade podataka. Ta informacija se sprema u parametru *num_workers* – ako je on jednak 0 koristit će se prva klasa, a inače će se koristiti druga. Obje ove klase imaju implementiranu funkciju *__next_data*() koju je roditeljska klasa _BaseDataLoaderIter propustila implementirati.

Odnosi navedenih klasa prikazani su na Slici 4 koja slijedi u nastavku.



Slika 4: Dijagram odnosa PyTorch razreda koji sudjeluju u OO Iterator (i OO Prilagodnik)

Na Slici 4 možemo vidjeti da su obitelji Skupnog objekta i Iteratora odvojene jedna od druge klasom DataLoader. Klasa DataLoader ima ulogu Prilagodnika između ove dvije obitelji, čime je postignuto da Konkretni iteratori ne ovise o Konkretnim skupnim objektima, kao što je to uobičajeno u OO Iterator.

```
batch_size = 2
shuffle = False
train_dataset = NLPDataset.from_file('data/sst_train_raw.csv')

my_dataloader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True
)


for batch_idx, (data_batch, labels_batch) in enumerate(my_dataloader):
```

```
    # obrada za svaki batch
```

Kôd 8: Primjer korištenja obrazaca Iterator (i Prilagodnik) u PyTorchu

U kôdu 8 može se vidjeti primjer korištenja klasa NPLDataset, nasljednika klase Dataset, i klase DataLoader koje se koriste za iteriranje po skupovima podataka u for petlji. Unutar for petlje može se obavljati proizvoljna radnja sa skupom podataka *data_batch*.

Posljedice korištenja OO Iterator u PyTorchu su:

- Apstrakcija pristupa podacima – interna struktura podataka ostaje sakrivena, a podatcima se jednostavno pristupa korištenjem iterabilnog objekta klase DataLoader.
- Učinkovito rukovanje velikim skupovima podataka – podatci ce dijele u skupine (eng. batch), što je ključno za učinkovito korištenje grafičkog procesora.
- Poštuje se Načelo jedinstvene odgovornosti (NJO) – dio klasa koji pohranjuje podatke ne mora razmišljati o načinu pristupa podacima, već se za to brinu iteratorske klase.
- Uvođenjem klase DataLoader je riješen problem cirkularne ovisnosti koji se javlja u OO Iterator.

Oblikovni obrazac Iterator je u programskom okviru PyTorch našao idealnu primjenu s obzirom na prirodu namjene programskog okvira PyTorch.

6. Oblikovni obrazac Prilagodnik: klasa DataLoader

Oblikovni obrazac Prilagodnik pripada skupini strukturnih dijagrama i njegova namjena jest prilagoditi postojeće razrede sučelju koje očekuju klijenti tih razreda. On omogućava pristup raznorodnim resursima na uniforman i transparentan način, što pospješuje inverziju ovisnosti.

Sudionici ovog obrasca su Ciljno sučelje, Klijent, Vanjska komponenta i Prilagodnik.

Kao što je već bilo spominjano u prethodnom poglavlju, obrazac Prilagodnik koristi se u implementaciji klase torch.utils.data.DataLoader. Klasa DataLoader djeluje kao Prilagodnik između Dataset klase, koja služi za pohranu uzoraka, i sučelja potrebnog za iteraciju po uzorcima, _BaseDataLoaderIter.

Primjer korištenja obrasca Prilagodnik u PyTorchu

Ulogu **Vanjske komponente** preuzima apstraktna klasa torch.utils.data.Dataset. Apstraktna klasa Dataset ne nudi funkcionalnost iteriranja kroz podatke, ali definira metodu *__get_item(self, index),* koja se koristi za dohvaćanje podatka na mjestu *index*. Klijent koji koristi ovu klasu želi imati mogućnost jednostavne i uniformne iteracije po podacima koje Dataset čuva, stoga se javila potreba za Prilagodnikom koji će Klijentu pružiti željenu funkcionalnost.

Ulogu **Ciljanog sučelja** ima apstraktna klasa _BaseDataLoaderIter. Korisnik ima zahtjev za jednostavnim načinom iteracije kroz podatke unutar Dataseta, no nije mu bitno koja konkretna klasa će ispuniti taj zahtjev. Korisnikov zahtjev zadovoljio bi Prilagodnik koji koristi bilo koju vrstu iteratora, a u ovom slučaju PyTorch koristi upravo _BaseDataLoaderIter. Stoga možemo zaključiti da ciljno sučelje u ovom slučaju nije specifično definirano, nego je definirana specifična funkcionalnost koja se želi ostvariti, a ta funkcionalnost je ostvarena upravo kroz klasu _BaseDataLoaderIter.

Ulogu **Prilagodnika** ima klasa DataLoader. Klasa DataLoader prima instancu Dataset klase u svom konstruktoru. Umjesto da klijent ručno poziva *dataset.__getitem__()* za svaki uzorak i sam ga spaja u *batcheve* podataka, DataLoader pruža jednostavno, iterabilno sučelje koje "adaptira" ponašanje Dataseta kako bi odgovaralo potrebama klijenta za iteracijom po *batchevima* podataka. Za ostvarivanje te funkcionalnosti DataLoader koristi već spomenutu apstraktnu klasu _BaseDataLoaderIter.

Ulogu **Klijenta** preuzima korisnik PyTorcha, točnije programska petlja koja želi iterirati kroz *batcheve* podataka. Klijent može ostvariti iteraciju po podacima unutar Dataseta interaktirajući isključivo s DataLoader objektom, pozivajući ga u for petlji. Tako je Klijent potpuno nesvjestan načina na koji DataLoader interno komunicira s Dataset i _BaseDataLoaderIter objektima kako bi dobio i obradio podatke.

Odnosi navedenih komponenata već su prikazani na Slici 4, a ovdje ćemo ih ponovo analizirati kroz njihove uloge u obrascu Prilagodnik.

Klasa DataLoader ima ulogu Prilagodnika između ove dvije obitelji – ona enkapsulira stvaranje iteratora, čime je njegova implementacija skrivena i od Klijenta i od Vanjske komponente. S druge strane, klasa _BaseDataLoaderIter pristupa podacima nad kojima vrši iteraciju upravo preko klase DataLoader, čiju instancu prima u konstruktoru. Dakle, osim što klasa DataLoader enkapsulira pristup Iteratoru od Dataseta, ona također enkapsulira pristup Datasetu od Iteratora. Jedino što povezuje Ciljno sučelje i Vanjsku komponentu jest upravo klasa DataLoader, što je uvelike unaprijedilo inverziju ovisnosti unutar ovog odnosa.

Također, s obzirom na usku povezanost klasa koje sudjeluju u OO Iterator i OO Prilagodnik, primjer korištenja OO Prilagodnik u okviru PyTorch može se vidjeti u kôdu 8, u kojem smo promatrali način korištenja OO Iterator.
U primjeru možemo vidjeti da klijent koristi instancu klase DataLoader poput iteratora unutar for petlje te da instanca DataLoadera omogućuje jednostavno iteriranje po *batchevima* podataka. Klijent u trenutku korištenja instance DataLoadera u for petlji nije svjestan unutrašnje implementacije sustava, nego samo zna da je ono što će mu iterator vratiti *batch* podataka, kojeg on onda može na željeni način obraditi.

Posljedice korištenja OO Prilagodnik u programskom okviru PyTorch su slijedeće:

- Pospješena inverzija ovisnosti – klasa DataLoader odvaja Dataset od konkretne implementacije iteracije (_BaseDataLoaderIter) i obrnuto. Time se omogućuje da komponente više razine ne ovise o detaljima implementacije nižih razina.
- Smanjena duplikacija koda – klasa DataLoader može se koristiti nad različitim tipovima Datasetova, bez potrebe za pisanjem posebnih petlji za učitavanje, čime je postignuta ponovna iskoristivost koda.
- Klasa DataLoader skriva unutarnju implementaciju iteratora od Klijenta, a da i dalje pruža funkcionalnost iteriranja, što sustav čini jednostavnim za korištenje, a kod je pregledniji i čišći.

Na temelju svega ovoga možemo zaključiti da oblikovni obrasci Iterator i Prilagodnik zajedničkim snagama čine moćan par koji uvelike unaprjeđuje i olakšava rad s ovim programskim okvirom.

7. Oblikovni obrazac Tvornica: metoda *torch.hub.load()* i modul torchvision.models

Oblikovni obrazac Tvornica pripada skupini kreacijskih obrazaca. Njegova glavna svrha je apstrahirati proces stvaranja objekata tako da klijent ne mora biti upoznat s detaljima kako se objekti konkretno instanciraju. Umjesto toga, objekt se stvara kroz definirano sučelje, čime se omogućuje lakše proširivanje sustava, a kod postaje zatvoreniji za promjene.

Sudionici ovog obrasca su Tvornica, Apstraktni Proizvod i Konkretni Proizvod.

Iako ovaj obrazac ima jasnu strukturu u mnogim programskim jezicima, u okviru PyTorcha često se koriste idiomatski i fleksibilniji pristupi temeljenim na dinamičkom stvaranju objekata. Takvi pristupi imaju jednaku namjeru kao i klasična tvornica, no teže je identificirati klasične uloge oblikovnog obrasca.

Dva pristupa dinamičkom stvaranju objekata u Pytorchu su korištenje funkcije torch.hub.load() i korištenje modula torchvision.models, koji pružaju sličnu razinu apstrakcije i fleksibilnosti kao i klasična tvornica.

Primjeri korištenja obrasca Tvornica u PyTorchu

Prvi primjer koji će se proučavati jest tvornička metoda *torch.hub.load().* Ova metoda omogućuje dinamičko učitavanje modela neuronskih mreža iz udaljenih *GitHub* repozitorija ili iz lokalnih direktorija. Za pravilno učitavanje modela, funkciji je potrebno predati nekoliko argumenata: naziv izvora (npr. *'pytorch/vision:v0.10.0'*), ime modela (npr. *'resnet50'*), te opcionalno listu argumenata za inicijalizaciju modela, vrstu izvora (*'github'* ili *'local'*), kao i dodatne logičke parametre koji definiraju način učitavanja.

Ova metoda enkapsulira proces pronalaženja, dohvaćanja i instanciranja modela te ga klijentu izlaže kroz jednostavno i ujednačeno sučelje. Ulogu **Tvornice** u ovom slučaju preuzima sama metoda *torch.hub.load(),* dok se konkretni proizvodi specificiraju putem string identifikatora modela, poput *'resnet50'*. Budući da metoda uvijek vraća instancu modela neuronske mreže, ulogu **Apstraktnog Proizvoda** ima bazna klasa torch.nn.Module. **Konkretni proizvodi** su pojedinačne arhitekture modela koje nasljeđuju tu klasu, a koje su već prethodno spominjane u seminaru.

Jednostavni primjer korištenja tvorničke metode *torch.hub.load()* slijedi u nastavku.

```
model = torch.hub.load(
    "pytorch/vision",
    "resnet50",
    weights="ResNet50_Weights.IMAGENET1K_V1"
)

model.eval()
```

Kôd 9: Primjer korištenja metode torch.hub.load() u programskom okviru Pytorch

U kôdu 9 može se vidjeti primjer učitavanja unaprijed istreniranog modela ResNet-50 iz Pytorchevog torchvision repozitorija na GitHubu. U ovom primjeru, korisnik na jednostavan način stvara objekt željenog modela pozivom metode torch.hub.load, bez potrebe za stvaranjem ovisnosti između njega i modela ResNet-50 kojeg je odlučio koristiti.

Drugi primjer koji ilustrira oblikovni obrazac Tvornica u okviru PyTorch biblioteke jest modul torchvision.models. Ovaj modul nudi pristup velikom broju unaprijed definiranih arhitektura neuronskih mreža. Modul je organiziran tako da korisniku omogućuje stvaranje različitih modela na standardiziran način, bez potrebe za ručnim instanciranjem konkretnih klasa ili eksplicitnim definiranjem arhitekture. Time se postiže visoka razina apstrakcije, što je osnovna svrha obrasca Tvornica.

U kontekstu ovog modula, **Tvornicu** predstavlja sam modul *torchvision.models*, jer on objedinjuje i pruža pristup različitim funkcijama koje instanciraju modele. Iako ovdje nije riječ o klasičnoj objektno-orijentiranoj tvornici s metodama poput *createModel()*, funkcije unutar modula ispunjavaju istu ulogu, jer omogućuju kreaciju konkretnih objekata na temelju znakovnog identifikatora ili funkcijskih poziva.

Ključna funkcija u ovom sustavu jest funkcija *get_model()*. Ova funkcija prima ime modela kao string (npr. 'resnet50', 'mobilenet_v3_small', 'vit_b_16'), kao i dodatne parametre poput argumenta *weights* kojim se može specificirati koje unaprijed naučene težine koristiti. Pozivom funkcije *get_model('resnet50', weights='DEFAULT')*, korisnik dobiva u potpunosti inicijaliziran model spreman za treniranje ili inferenciju, bez potrebe za ručnim definiranjem njegove arhitekture. Time funkcija *get_model()* preuzima ulogu **Tvornice**, jer enkapsulira logiku instanciranja, te omogućuje dinamičko stvaranje objekata.

Ulogu **Apstraktnog Proizvoda** preuzima bazna klasa torch.nn.Module, koja je zajednička svim modelima unutar modula torchvision.models, baš kao i u prošlom primjeru. Bez obzira radi li se o modelima kao što su ResNet, VGG, EfficientNet ili ViT, svi oni nasljeđuju klasu nn.Module i ponašaju se sukladno definiranoj apstrakciji.

Pojedinačni modeli koji nasljeđuju nn.Module predstavljaju **Konkretne Proizvode**. To mogu biti primjerice klase torchvision.models.resnet.ResNet, torchvision.models.vgg.VGG, torchvision.models.alexnet.AlexNet i drugi.

Dodatnu fleksibilnost u upravljanju modelima pruža funkcija *get_model_weights()*, koja omogućuje dohvat svih dostupnih skupova težina za određeni model. Time korisnik može, osim zadane vrijednosti *'DEFAULT'*, odabrati alternativne verzije istog modela koje su trenirane na različitim skupovima podataka ili s različitim optimizacijama. Na primjer, pozivom *get_model_weights('resnet50')* moguće je dobiti sve dostupne varijante težina za model ResNet-50.

Za pregled svih dostupnih modela u okviru modula, funkcija *list_models()* vraća listu podržanih string identifikatora modela koji se mogu koristiti kao ulaz u funkciju *get_model()*. Ova funkcija dodatno jača ulogu modula torchvision.models kao Tvornice, jer omogućuje programsko istraživanje prostora proizvoda i dinamički odabir modela temeljen na vanjskim uvjetima, poput korisničkog unosa ili konfiguracijskih datoteka.

Iako modul torchvision.models ne koristi klasičnu objektno-orijentiranu hijerarhiju tvornica, modul savršeno utjelovljuje temeljne ideje oblikovnog obrasca Tvornica: apstrakciju stvaranja objekata, skrivanje složenosti instanciranja i lako proširivu strukturu proizvoda. Novi modeli se mogu lako dodavati u modul, bez potrebe za izmjenama postojećeg sučelja, što omogućuje održivost i skalabilnost sustava.

```
# Ispis svih podržanih imena modela
available_models = list_models()
print(available_models)

# Dohvati dostupne težine za ResNet50
weights = get_model_weights('resnet50')
print(weights)

# Kreiraj model ResNet50 s unaprijed treniranim težinama
model = get_model('resnet50', weights='DEFAULT')

model.eval()
```

Kôd 10: Primjer korištenja metoda list_models, get_model_weights i get_model u programskom okviru Pytorch

U ovom programskom primjeru možemo vidjeti primjer jednostavnog korištenja prethodno opisanih funkcija. Funkcija *list_models* koristi se za ispis imena svih dostupnih modula koji se mogu kreirati, a funkcija *get_model_weights* koristi se za ispisivanje svih dostupnih težina konkretnog modela, koji je u ovom slučaju ResNet50. Predzadnja linija programa pokazuje kako se na temelju imena modela može stvoriti konkretan objekt te klase, za što se koristi tvornička funkcija *get_model*.

Posljedice korištenja OO Tvornica u okviru PyTorch su:

- Apstrakcija procesa stvaranja objekata —klijent ne mora poznavati niti upravljati složenim detaljima inicijalizacije modela, što pojednostavljuje korištenje i smanjuje mogućnost pogrešaka.
- Jednostavna proširivost — dodavanje novih modela ne zahtijeva izmjene u postojećem kôdu tvornice, već se novi modeli jednostavno dodaju kao zasebne klase unutar modula torchvision.models, što omogućuje laku nadogradnju i održavanje.
- Poštivanje Načela nadogradnje bez promjene — postojeći kod tvornice i klijenta ne mijenja se prilikom dodavanja novih modela.

Primjena OO Tvornica u PyTorch okviru omogućava jednostavno i lako održivo upravljanje složenim procesom stvaranja modela na dva različita načina, što korisnicima olakšava rad i unaprjeđuje kvalitetu koda.

8. Zaključak

Oblikovni obrasci predstavljaju važan alat u razvoju softverskih sustava jer omogućuju fleksibilniju, pregledniju i proširivu arhitekturu. U ovom radu analizirani su obrasci Kompozit, Okvirna metoda, Strategija, Iterator, Prilagodnik i Tvornica, u kontekstu programskog okvira PyTorch. Svaki od njih koristi se za rješavanje specifičnog problema: Kompozit za rad s hijerarhijskim modelima, Okvirna metoda za definiciju okvirnih koraka funkcije poziva modela, Strategija za zamjenjivost različitih algoritama, poput algoritma optimizacije i algoritma za računanje pogreške, Iterator i Prilagodnik za efikasno i standardizirano iteriranje nad skupovima podataka te Tvornica za jednostavan i apstraktan način stvaranja modela neuronskih mreža.

Programski primjeri pokazuju da su ovi obrasci sustavno primijenjeni i u skladu su s načelima objektno orijentiranog dizajna. Njihova primjena u okviru PyTorch pojednostavljuje rad s kompleksnim modelima, omogućuje ponovnu upotrebu koda i olakšava proširenja. Znanje o ovim obrascima korisno je i za razumijevanje unutarnjeg funkcioniranja PyTorcha, ali i za pisanje vlastitih modula i rješenja koja se na njega oslanjaju.

9. Literatura

PyTorch dokumentacija. Dostupno na: https://docs.PyTorch.org/docs/stable/index.html (pristupljeno u lipnju 2025.)

PyTorch. *GitHub repozitorij – PyTorch/PyTorch*. Dostupno na: https://github.com/PyTorch/PyTorch (pristupljeno u lipnju 2025.)

PyTorch tutorial o DataLoader i Dataset klasi. Dostupno na: https://docs.PyTorch.org/tutorials/beginner/basics/data_tutorial.html#iterate-through-the-dataloader (pristupljeno u lipnju 2025.)

Refactoring Guru: *Design Patterns – Refactoring.Guru*. Dostupno na: https://refactoring.guru/design-patterns (pristupljeno u lipnju 2025.)

Neslužbene stranice predmeta Duboko učenje 1. Dostupno na https://www.zemris.fer.hr/~ssegvic/du/ (pristupljeno u lipnju 2025.)

Materijali predmeta Oblikovni obrasci u programiranju. Dostupno na https://www.fer.unizg.hr/predmet/ooup/materijali (pristupljeno u lipnju 2025.)

Alat za crtanje dijagrama diagrams.net (Draw.io). Dostupno na: https://app.diagrams.net/ (pristupljeno u lipnju 2025.)