**Univerzitet u Nišu, Elektronski fakultet**
**Katedra za računarstvo**

# Maskirano jezičko modelovanje

Nemanja Petrović, 1499

**Predmet: Duboko učenje**

**Profesor: prof. dr Aleksandar Milosavljević**

**Niš, 2023**

# Sadržaj

## Table of Contents

# 1 Uvod i cilj projekta

U današnjem digitalnom dobu, jezičko modelovanje postaje sve značajnije, kako za akademsku zajednicu tako i za industriju. Razvoj i primena naprednih jezičkih modela otvaraju nove mogućnosti u oblastima kao što su prevođenje, generisanje teksta, analiza sentimenta i mnoge druge. Jedan od jezičkih modela koji se koristi u ove svrhe je **BERT** (Bidirectional Encoder Representations from Transformers), koji je razvijen od strane Google-a.

BERT je vrsta jezičkog modela koji se zasniva na transformer arhitekturi, koja je postala popularna zbog svoje efikasnosti u obradi prirodnog jezika. Ono što BERT čini posebnim je njegova sposobnost da shvati kontekst i značenje reči u rečenici putem dvosmernog kodiranja. Ovo znači da BERT analizira kontekstualne informacije kako sa leve tako i sa desne strane reči, što omogućava bolje razumevanje jezičkih nijansi i složenosti.

Jedna od ključnih tehnika koju BERT koristi je maskirano jezičko modelovanje. Ova tehnika se koristi za predviđanje nepoznatih reči ili tokena na osnovu konteksta rečenice. Na primer, BERT može da nauči kako da predvidi nedostajuću reč u rečenici na osnovu prethodnih reči i konteksta u kojem se rečenica nalazi. Ovo je posebno korisno u situacijama gde je potrebno popuniti praznine u tekstu ili generisati rečenica.

Konkretnije, u ovom projektu će se koristiti **RoBERTa** (Robustly Optimized BERT pretraining approach) model. RoBERTa je verzija BERT-a koja je poboljšana finim podešavanjem parametara i dodatnom obukom na većem skupu podataka. Ovaj model se pokazao veoma efikasnim u razumevanju složenih jezičkih konstrukcija, kao i u različitim zadacima obrade prirodnog jezika.

Cilj ovog projekta je istraživanje i implementacija maskiranog jezičkog modela, sa fokusom na korišćenju RoBERTa modela i to sa ciljem da se on nauči da razume novi jezik, konkretno srpski jezik, kao i da se dodatno obuči da razume pravne tekstove na srpskom jeziku .

Kroz analizu postojećih radova i primenu relevantnih tehnika mašinskog učenja, projekat ima za cilj da pruži dublje razumevanje procesa maskiranog jezičkog modelovanja, posebno u kontekstu učenja novog jezika i razumevanju pravnih tekstova. Kroz fine tuning RoBERTa modela na specifičnom skupu podataka koji predstavljaju pravne tekstove, cilj je postići poboljšanja u preciznosti i tačnosti u razumevanju pravnih tekstova.

U nastavku izveštaja će biti detaljno opisan metodologija korišćena tokom projekta, rezultati eksperimenata, kao i diskusija o potencijalnim unapređenjima i daljim istraživanjima na ovom polju.

# 2 Korišćene tehnologije

## 2.1 BERT model

**BERT (Bidirectional Encoder Representations from Transformers)** je jezički model razvijen od strane Google-a. Ovaj model je predstavljen 2018. godine i predstavlja veliki korak napred u oblasti obrade prirodnog jezika.

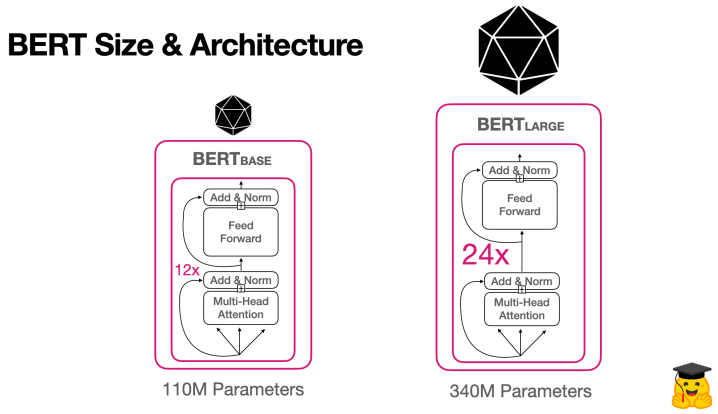
BERT se zasniva na transformer arhitekturi koja je postala popularna u jezičkom modelovanju. Transformer arhitektura koristi mehanizam pažnje kako bi model mogao efikasno obrađivati sekvencijalne podatke, poput rečenica. Mehanizam pažnje omogućava modelu da obraća posebnu pažnju na određene delove ulaznih podataka prilikom generisanja izlaza.

Ono što čini BERT posebnim je njegova sposobnost da shvati kontekst i značenje reči u rečenici putem dvosmernog kodiranja. To znači da BERT analizira kontekstualne informacije kako sa leve tako i sa desne strane reči. Za razliku od prethodnih jezičkih modela koji su se oslanjali samo na levo ili desno kontekstualno kodiranje, BERT je prvi model koji koristi bidirekcionalno kodiranje, što omogućava bolje razumevanje jezičkih nijansi i složenosti.

BERT je treniran na velikom korpusu podataka koji su prikupljeni sa interneta. Koristi se koncept preobuke (pretraining) na velikim količinama neoznačenih podataka, što mu omogućava da stekne široko razumevanje jezičkih konstrukcija. Nakon preobuke, BERT se može dalje fino podesiti (fine tuning) za specifične zadatke obrade prirodnog jezika, kao što su prevođenje, generisanje teksta, analiza sentimenta, prepoznavanje entiteta i drugi.

BERT se koristi u raznim zadacima obrade prirodnog jezika i postigao je izvanredne rezultate. Njegova sposobnost da nauči kontekstualno značenje reči omogućava mu da bolje razume složene jezičke konstrukcije, kao i da prepozna sličnosti i razlike između reči i rečenica. Ovaj model se takođe koristi za generisanje visokokvalitetnih i coerentnih tekstova na osnovu zadatih početnih reči ili rečenica.

Na *slici 1* data je arhitektura i veličina dva originalna BERT modela, u pitanju su osnovni BERT model i veliki BERT model.




*Slika 1: Arhitektura i veličina dva originalna BERT modela*

## 2.2 RoBERTa model

**RoBERTa (Robustly Optimized BERT pretraining approach)** je unapređena verzija BERT-a koja je razvijena kako bi se poboljšala efikasnost i performanse jezičkog modelovanja. Ovaj model predstavlja rezultat istraživanja i eksperimenata u cilju optimizacije BERT-a.

RoBERTa se zasniva na istoj transformer arhitekturi kao i BERT. Ova arhitektura koristi mehanizam pažnje za efikasno kodiranje sekvencijalnih podataka, kao što su rečenice, omogućavajući modelu da obraća posebnu pažnju na relevantne delove ulaznih podataka prilikom generisanja izlaza.

Jedna od ključnih razlika između RoBERTa i BERT-a je u načinu preobuke (pretraining). RoBERTa je trenirana na još većem skupu podataka i sa dužim vremenskim trajanjem treniranja. Korišćenje većeg skupa podataka i duže preobuke omogućava RoBERTi da stekne dublje razumevanje jezičkih konstrukcija i da nauči suptilnosti u značenju reči i konteksta.

RoBERTa takođe primenjuje nekoliko optimizacija u odnosu na originalni BERT. Ove optimizacije uključuju promene u arhitekturi, fine tuning hiperparametara i druge tehnike koje doprinose poboljšanju performansi modela.

Kao i BERT, RoBERTa ima široku primenu u raznim zadacima obrade prirodnog jezika. Ovaj model je pokazao izvanredne rezultate u prevođenju, generisanju teksta, analizi sentimenta, prepoznavanju entiteta i drugim zadacima. RoBERTa ima sposobnost da dublje razume kontekstualno značenje reči, što je ključno za precizno i coerentno obradu prirodnog jezika.

Kombinacija veće količine podataka, duže preobuke i optimizacija u RoBERTa modelu čine ga snažnim alatom za razumevanje i generisanje teksta. Ovaj model otvara nove mogućnosti u razumevanju složenijih jezičkih konstrukcija i pruža temelj za dalji napredak u obradi prirodnog jezika.

## 2.3 OSCAR skup podataka

**OSCAR (Open Super-large Crawled ALMAnaCH coRpus)** je veliki skup podataka koji se koristi za preobuku jezičkih modela u oblasti obrade prirodnog jezika. Ovaj skup podataka je razvijen u cilju pružanja obimnog i raznovrsnog korpusa koji obuhvata tekstove na više jezika i različitih domena.

OSCAR koristi web skrejping (web crawling) tehnike kako bi prikupio tekstove sa javno dostupnih web stranica. Skup podataka obuhvata preko 100 jezika, sa ukupno preko 100 terabajta podataka. To ga čini jednim od najvećih dostupnih korpusa tekstova za preobuku jezičkih modela.

Jedna od ključnih karakteristika OSCAR-a je da uključuje tekstove iz različitih domena, kao što su vesti, članci, blogovi, forumi i još mnogo toga. Ovo pruža raznolikost u sadržaju i kontekstu, što je važno za razvijanje jezičkih modela koji su sposobni da obrađuju tekstove iz različitih izvora i sa različitim stilovima pisanja.

OSCAR se koristi za preobuku jezičkih modela, uključujući BERT, RoBERTa i druge. Preobuka na OSCAR-u omogućava modelima da steknu široko razumevanje jezičkih konstrukcija, značenja reči i konteksta. Veći korpus podataka kao što je OSCAR pruža veću raznolikost i obuhvata šire jezičko znanje, što može dovesti do boljih rezultata prilikom primene tih modela na različite zadatke obrade prirodnog jezika.

OSCAR se smatra vrednim resursom u istraživanju i primeni jezičkih modela. Veliki obim podataka, višejezičnost i raznolikost domena čine OSCAR korisnim za različite primene, uključujući prevođenje, generisanje teksta, analizu sentimenta, prepoznavanje entiteta i još mnogo toga.

| | | | | |
|---|---|---|---|---|
| sr | Serbian | 1,677,896 | 632,781,822 | 7.7 GB |

*Slika 2: Podaci o OSCAR setu za srpski jezik*

Na *slici 2* vidimo podatke o OSCAR setu za srpski jezik, u pitanju su podaci na ćirilici a brojevi govore o tome da set sadrži:

- 1677896 dokumenata
- 632781822 reči
- 7.7 GB podataka

## 2.4 PyTorch

Za izradu projekta će biti korišćen **Python** programski jezik, a jedan od ključnih alata koji će se koristiti je PyTorch.

**PyTorch** je otvorenokodni računarski okvir za mašinsko učenje i duboko učenje koji je popularan među istraživačima i praktičarima. Ovaj okvir omogućava razvoj i primenu složenih modela mašinskog učenja uz jednostavnu sintaksu i fleksibilnost.

Jedna od ključnih prednosti PyTorch-a je njegova jednostavnost i intuitivnost. PyTorch koristi dinamički računski graf koji omogućava korisnicima da definišu i modifikuju graf tokom izvršavanja. Ovo olakšava eksperimentisanje, debagovanje i prilagođavanje modela, kao i brzo pravljenje prototipova novih ideja. Python je fleksibilan programski jezik koji ima bogat ekosistem biblioteka, što dodatno olakšava razvoj složenih modela uz korišćenje PyTorch-a.

PyTorch takođe pruža visoku performansu zahvaljujući podršci za GPU (Graphics Processing Unit) akceleraciju. Ovo omogućava paralelno izvršavanje operacija na velikim tenzorima, što ubrzava procesiranje i treniranje složenih modela. Osim toga, PyTorch pruža i različite alate za optimizaciju performansi, kao što su automatsko diferenciranje (autograd) i mogućnost raspoređivanja na više uređaja.

## 2.5 FastAPI

Kada je istreniran model za obradu prirodnog jezika spreman za upotrebu, važno je omogućiti pristup modelu putem API-ja kako bi drugi korisnici mogli iskoristiti njegove sposobnosti. API (Application Programming Interface) pruža standardizovan način za komunikaciju između različitih softverskih komponenti.

Za izgradnju API-ja koji će izložiti istrenirani model, jedan od popularnih izbora je **FastAPI**. FastAPI je moderan, brz i jednostavan okvir za razvoj API-ja koristeći Python programski jezik. On kombinuje snagu Pythona sa brzinom i performansama zasnovanim na asinhronom programiranju.

FastAPI koristi Python tipove podataka, što olakšava pisanje čitljivog i pouzdanog koda. Takođe podržava automatsku generaciju interaktivne dokumentacije zahvaljujući ugrađenoj podršci za

OpenAPI i Swagger standardima. Ovo čini dokumentaciju API-ja jasnom, pristupačnom i lakom za korišćenje.

Jedna od ključnih prednosti FastAPI-ja je njegova brzina. Zahvaljujući asinhronom programiranju i upotrebi uvodjenja (coroutines), FastAPI može da obrađuje veliki broj zahteva paralelno, što doprinosi izuzetnoj skalabilnosti i performansama.

FastAPI takođe pruža podršku za validaciju podataka, autentifikaciju, autorizaciju i mnoge druge sigurnosne funkcionalnosti.

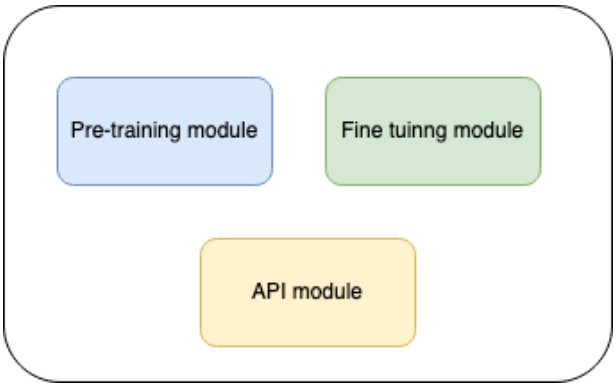
```python
@app.get("")

def read_root():

    return {"Hello": "World"}



@app.get("/items/{item_id}")

def read_item(item_id: int, q: Union[str, None] = None):

    return {"item_id": item_id, "q": q}
```

*Slika 3: Primer FastAPI koda*

# 3 Arhitektura projekta

## 3.1 Generalna arhitektura projekta

Projekat se sastoji iz tri modula koji zajedno čine kompletnu aplikaciju. Pojednostavljeni izgled arhitekture dat je na *slici 4*.




*Slika 4: Pojednostavljeni izgled arhitekture*

Prvi modul, modul za preobuku, fokusira se na proces treniranja modela na velikom skupu podataka. Za ovaj projekat, korišćen je RoBERTa model, opisan u prethodnoj sekciji. RoBERTa se preobučava na raznovrsnom korpusu tekstova, a za ovu svrhu se koristi OSCAR set za srpski jezi, takođe opisan u prethodnoj sekciji.

Drugi modul, modul za fine tuning, se fokusira na prilagođavanje prethodno preobučenog modela specifičnom zadatku obrade prirodnog jezika. U ovom projektu, fine tuning se vrši sa ciljem da model nauči da razume pravne tekstove. Za fine tuning se koristi odgovarajući skup podataka koji sadrži parvne tekstove, konrektno, pravni tekstovi su preuzeti sa pravno/informacionog sistema Republike Srbije. Ovaj proces omogućava modelu da se usmeri na specifičan zadatak i poboljša svoju sposobnost razumevanja pravnih tekstova.

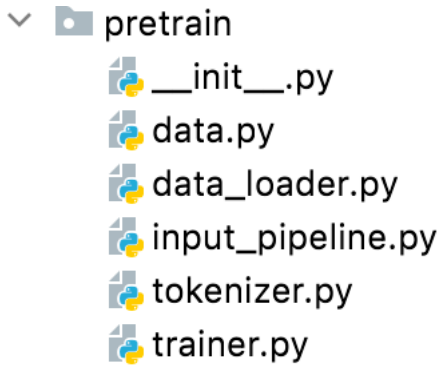
Treći modul, modul za izlaganje modela putem API-ja, omogućava drugim korisnicima pristup istreniranom modelu i korišćenje njegovih funkcionalnosti. Za izgradnju API-ja u ovom projektu je izabran FastAPI jer omogućava jednostavno definisanje rutera i kontrolera za obrađivanje zahteva, podržava validaciju podataka, autentifikaciju i autorizaciju, i pruža visoke performanse zahvaljujući asinhronom programiranju. Kroz API, korisnici će moći da koriste istrenirani model za obradu parvnih tekstova.

## 3.2 Pre-training modul

Kao što je rečeno, ovaj modul će biti korišćen za preobuku modela za razumevanje samog srpskog jezika. Taj proces se sastoji iz više koraka, koji moraju da budu izvršeni u sledećem redosledu:

1. Povlačenje OSCAR seta podataka
2. Kreiranje tokenizatora
3. Kreiranje ulaznih tenzora
4. Kreiranje dataloader-a
5. Treniranje modela

Više informacija o svim ovim koracima biće u narednim sekcijama, a strukturu fajlova ovog modula možemo videti na *slici 5*.



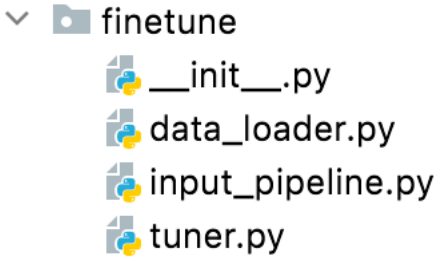

*Slika 5: Struktura modula za preobuku*

## 3.3 Fine tuning modul

Kada već postoji model koji razume srpski jezik, a to bi trebalo da posotji nakon završetka svih koraka definisanih u prethodnom modulu, potrebno je taj model dodatno istrenirati nad pravnim tekstovima.
Za to treniranje potrebno je odraditi sledeće korake:

1. Učitati model koji već razume srpski jezik
2. Kreirati tenzore za dodatno treniranje na osnovu pravnih tekstova
3. Pokrenuti dodatno treniranje modela

Struktura fajlova ovog modula data je na *slici 6*:




*Slika 6: Struktura fajlova u modulu za fine tune*

## 3.4 API modul

Ovaj modul je jako jednostavan jer postoji samo jedan endpoint čiji je cilj da omogući nekome da testira istrenirani model. Endpoint prihvata rečenicu gde je neka reč maskirana i kao odgovor daje predikciju koja bi reč tu trebalo da se nađe.


*Slika 7: Struktura fajlova u API modulu*

# 4 Implementacioni detalji

## 4.1 Pre-training

### 4.1.1 Priprema podataka

Kao što je već rečeno, za potrebe preobuke koristiće se OSCAR set podataka, specifično set za na srpskom jeziku. *Na slici 8* nalazi se kod za povlačenje seta podataka.

```
dataset = load_dataset("oscar-corpus/OSCAR-2301",
                       cache_dir="dataset_cache",
                       use_auth_token=hugging_face_token,
                       language="sr",
                       streaming=False)
```

*Slika 8: Kod za povlačenje OSCAR seta*

Funkciji za povlačenje seta podataka potrebno je proslediti nekoliko parametara, prosleđeni parametri od prvog do poslednjeg govore da:

- se preuzima najnovija verzija OSCAR seta podataka: OSCAR-2301
- se podaci keširaju u navedenom folderu
- se šalje token koji se nalazi u navedenoj promenljoj
- se preuzima set na srpskom jeziku
- se ne radi stream-ing podataka

Nakon što se podaci preuzmu, potrebno je da se podele u fajlove za lakšu obradu, to se radi tako što prvo uklonimo karaktere za označavanje novog reda, a zatim se obrađeni tekst dodaje u niz dok se ne dobije niz sa 10 hiljada primeraka teksta, taj niz se na kraju sačuva u fajl. Kod koji radi navedeno nalazi se na *slici 9*:

```
for sample in tqdm(dataset['train']):

    sample = sample['text'].replace('\n', '')
    text_data.append(sample)

    if len(text_data) == 10_000:
        with open(f'sr_{file_count}.txt', 'w', encoding='utf-8') as fp:
            fp.write('\n'.join(text_data))
        text_data = []
        file_count += 1

# after saving in 10K chunks, we have to add leftovers
with open(f'sr_{file_count}.txt', 'w', encoding='utf-8') as fp:
    fp.write('\n'.join(text_data))
```

*Slika 9: Kod za kreiranje fajlova sa podacima za trening*

### 4.1.2 Tokenizacija

Tokenizacija predstavlja proces konvertovanja neobrađenog teksta u manje jedinice koje se nazivaju tokenima. Ovi tokeni se zatim mogu mapirati na brojeve kako bi se dalje mogli dovesti na ulaz nekog NLP modela.
Vodeći se arhitekturom i principima koje je postavila RoBERTa i ovaj projekat koristiće isti algoritam tokenizacije pod nazivom *ByteLevelBPETokenizer* i njegovu implementaciju od strane *HuggingFace* biblioteke.

ByteLevelBPETokenizer je metoda tokenizacije teksta koja se koristi za pretvaranje ulaznog teksta u nizove "tokena" ili jedinica, kao što su pojedinačne reči ili podreči. Ova metoda koristi Byte Pair Encoding (BPE) algoritam za tokenizaciju.

BPE je algoritam kompresije bez gubitaka koji se koristi za segmentaciju teksta na manje jedinice, koje se zatim koriste kao tokeni. Ovaj algoritam gradi rečnik tokena na osnovu statistike o ponavljanju parova bajtova u tekstu. BPE algoritam iterativno spaja najčešće parove bajtova kako bi stvorio nove, duže tokene. Ovaj proces se ponavlja sve dok se ne dostigne željeni broj tokena ili dok se ne dostigne kraj iteracija.

ByteLevelBPETokenizer implementira BPE tokenizaciju na nivou bajtova. To znači da tekst prvo razbija na niz bajtova, a zatim primenjuje BPE algoritam na nivou tih bajtova. Ovaj pristup je koristan kada se radi sa tekstom na jezicima koji koriste različite skupove znakova ili slova koje nisu deo standardnog ASCII seta.

```
paths = [str(x) for x in Path('./').glob('*.txt')]
paths = paths[0:50]
print("Starting tokenizer training")
tokenizer = ByteLevelBPETokenizer()
tokenizer.train(
    files=paths,
    vocab_size=30_522,
    min_frequency=2, show_progress=True,
    special_tokens=[
        '<s>', '<pad>', '</s>', '<unk>', '<mask>'
    ]
)
```

*Slika 10: Kod za tokenizaciju*

Na slici 10 se vidi kod koji se koristi za tokenizaciju, bitno je objasniti konfiguraciju tokenizatora:

- **files**: Tokenizacija se vrši nad delom generisanih fajlova, konkretno 50 od 84
- **vocab_size**: Veličina vokabulara, 30522 je preporučena veličina za manju implementaciju RoBERTa-e
- **min_frequency**: Minalni broj pojalvjivanja neke reči koji je potreban da bi s eta reč uzela u obzir za tokenizaciju
- **special_tokens**: Specijalni karakteri, najbitniji od njih je <mask> karakter koji predstavlja maskiranu reč koju je potrebni pogoditi

### 4.1.3 Kreiranje tenzora

Tenzori su osnovna struktura podataka u mašinskom učenju. Oni su višedimenzionalni nizovi koji sadrže brojeve. U kontekstu mašinskog učenja, tenzori se koriste za reprezentaciju ulaznih podataka (kao što je tekst) i izlaza ili rezultata modela. Tenzori omogućavaju manipulaciju, obradu i transformaciju podataka, kao i izvršavanje različitih operacija koje su neophodne za rad modela, kao što je izračunavanje gradijenata i optimizacija parametara.

Za kreiranje tenzora potreban je tokenizator koji je istreniran u prethodnom koraku i koristeći njega kreiramo 3 liste tenzora.

```
input_ids = []
mask = []  # attention mask
labels = []

print("Iterating through files and creating tensors")
for path in tqdm(paths):
    print("File: " + path)
    with open(path, 'r', encoding='utf-8') as f:
        lines = f.read().split('\n')
    sample = tokenizer_srberta(lines, max_length=512, padding='max_length', truncation=True, return_tensors='pt')
    labels.append(sample.input_ids)
    mask.append(sample.attention_mask)
    input_ids.append(mlm(sample.input_ids.detach().clone()))

input_ids = torch.cat(input_ids)
mask = torch.cat(mask)
labels = torch.cat(labels)
```

*Slika 11: Kreiranje tenzora*

Kao što se vidi na *slici 11*, najpre se kreiraju 3 prazne liste koje će sadržati tenzore koji predstavljaju ulazne identifikatore tokena (*input_ids*), maske pažnje (*masks*), i labele (*labels*).
Zatim se prolazi kroz sve fajlove i koristeći tokenizator kreira se tenzor maksimalne dužine 512. Na kraju se kreirani tenzori spajaju koristeći *torch.cat*.

### 4.1.4 Učitavanje podataka

Pre početka treninga modela, potrebno je učitati tenzore, za to je potrebno odrediti batch size, on se određuje na osnovu resursa dosptupnih za trening, veći batch size znači brži trening ali i korišćenje više resursa. Za treniranje ovog modela korišćen je batch size 8 jer je to maksimalni batch size koji je mogao da se koristi na dostpunoj grafičkoj kartici.

```
encodings = {
    'input_ids': input_ids,
    'attention_mask': mask,
    'labels': labels
}
dataset = Dataset(encodings)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

*Slika 12: Učitavanje tenzora*

Na *slici 12*, vidi se učitavanje tenzora koristeći Dataloader iz pytorch-a.

### 4.1.5 Pre-trainig modela

Pre nego da se započne sa procesom treniranja modela, potrebno je definisati različite hiper-parametre. Hiper-parametri izabrani za trneiranje ovog modela dati su na *slici 11*, izabrani su na osnovu dostpunih resursa kao i na osnovu preporučenih vrednosti za RoBERTa model.

```
config = RobertaConfig(
    vocab_size=tokenizer_srberta.vocab_size,
    max_position_embeddings=514,
    hidden_size=768,
    num_attention_heads=12,
    num_hidden_layers=6,
    type_vocab_size=1,
)
```

*Slika 13: Hiper-parametri*

- Veličina vokabulara - Postavljanje veličine vokabulara direktno utiče na arhitekturu mreže tako što, između ostalog, definiše veličinu vektora rezultata. Naime, na izlazu iz *RobertaForMaskedLM* mreže dobija se *MaskedLMOutput* koji u podrazumevanom obliku sadrži rezultat funkcije gubitaka i tenzor pod nazivom *logits*.
- Maksimalni broj pozicionih embedinga – Ovaj parametar se, prema dokumentaciji koju nudi *HuggingFace,* postavlja na vrednost maksimalne veličine ulazne sekvence plus 2.
- Veličina *hidden_size* - Dimenzionalnost enkoder slojeva, podrazumevano se postavlja na vrednost 768. Ono što zapravo ova veličina predstavlja jeste dimenzionalnost vektora skrivenog stanja, koji odgovara svakom tokenu iz ulazne sekvence.
- Broj glava mehanizma pažnje – Ova vrednost odnosi se na broj glava mehanizma pažnje u svakom sloju Pažnje u okviru transformer enkodera i postavljena je na podrazumevanu vrednost 12.
- Broj skrivenih slojeva – Broj skrivenih slojeva u transformer enkoderu. Vrednost postavljena na 6 ukazuje na to da koristimo manju verziju arhitekture ovog modela.
- Tip veličine vokabulara – Postoje dve veličine vokabulara, veliki koji poseduje 50 hiljada tokena i mali sa oko 30 hiljada tokena. S obzirom da je u ovom projektu korišćen rečnik manjih dimenzija, vrednost parametra *tip veličine vokabulara* postavljena je na vrednost 1.

Nakon podešavanja svih parametara, može se pokrenuti treniranje modela. Treniranje modela odvija se u određenom broju epoha, u svakoj epohi prolazi se kroz sve podatke iz dataloader-a koji je definisan u prethodnom koraku.

- Prvo se postavljaju gradijenti na nulu pozivom `optim.zero_grad()`. To je važno jer želimo da gradijenti budu izračunati samo za trenutni batch podataka.

- Ulazni podaci se prenose na odgovarajući uređaj (npr. GPU) pozivom `.to(device)` kako bi se ubrzao proces treniranja.

- Varijable `input_ids`, `mask` i `labels` se izdvajaju iz `batch` objekta, što su ulazni identifikatori tokena, maske pažnje i oznake za treniranje modela.

- Poziv `model(input_ids, attention_mask=mask, labels=labels)` šalje ulazne podatke modelu. Model generiše predikcije na osnovu ulaza i izračunava gubitak (loss) na osnovu stvarnih oznaka. `outputs` objekat sadrži predikcije modela i gubitak.

- Poziv `loss.backward()` izračunava gradijente gubitka u odnosu na parametre modela.

- Poziv `optim.step()` ažurira parametre modela koristeći optimizator (`optim`), na osnovu izračunatih gradijenata.

Ceo opisani proces nalazi se u kodu na *slici 14*. dok se na *slici 15* nalazi kod koji se izvršava na kraju procesa treniranja i koji čuva istrenirani model.

```
for epoch in range(epochs):
    step = 0
    loop = tqdm(dataloader, leave=True)
    for batch in loop:
        optim.zero_grad()
        input_ids = batch['input_ids'].to(device)
        mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(input_ids, attention_mask=mask, labels=labels)
        loss = outputs.loss

        loss.backward()
        optim.step()

        loop.set_description(f'Epoch: {epoch}')
        loop.set_postfix(loss=loss.item())

        writer.add_scalar("Loss/train", loss, step)
        writer.flush()
        step += 1
```

*Slika 14: Treniranje modela*

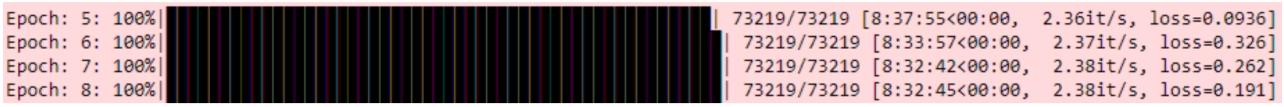
```
torch.save({
    'optimizer_state_dict': optim.state_dict()
}, 'optimizer.pt')

model.save_pretrained("./srberta_model")
save(model, optim)
```

*Slika 15: Čuvanje istreniranog modela*

### 4.1.6 Proces treniranja

Kao što je rečeno proces treniranja sastoji se iz određenog broja epoha. U slučaju treniranja ovog modela taj broj je bio 19. Kako se došlo do tog broja?
Pre nego da se dođe do objašnjenja zašto je trening zaustavljen nakon 19 epohe, potrebno je reći nešto o mašini na kojoj je treniran model.

Model je treniran na grafičkoj kartici Nvidia Quadro RTX 4000 sa 8 GB memorije, pa na toj kartici model nakon 19 epoha nije više pokazivao da uči i treniranje je tu zaustavljeno a samo treniranje trajalo je 6 dana, 17 sati i 30 minuta, a vremena treninga za neke od epoha data su na slici 16:




*Slika 15: Vreme treniranja*

## 4.2 Fino podešavanje

### 4.2.1 Podaci

Fino podešavanje se radi nad podacima dobijenih sa sajta pravnog sistema Republike Srbije i tu spadaju:

- Ustav
- Zakoni
- Odredbe
- Rešenja

Podaci se pripremaju na isti način kao i kod procesa preobuke, pa neće biti dodatno opisani, potrebno je samo reći da se radilo 10% teksta izdvojilo sa strane kako bi se koristio za validaciju nakon završenog treniranja.

### 4.2.2 Treniranje - fino podešavanje

Kao i kod obrade podataka i proces treniranja je isti kao i ranije opisan uz par bitnih razlika kao što su:

- Ne inicijalizuje se novi model već se učitava istrenirani model koji već razume srpski jezik, samim tim nije potrebno postaviti hiper-parametre, oni će biti isti kao i kod preobuke
- Kod pripreme ulaznih tenzora postavljen je batch size na 2 a ne na 8, nemamo toliko podataka kao kod preobuke pa nam nije potreban veći batch size

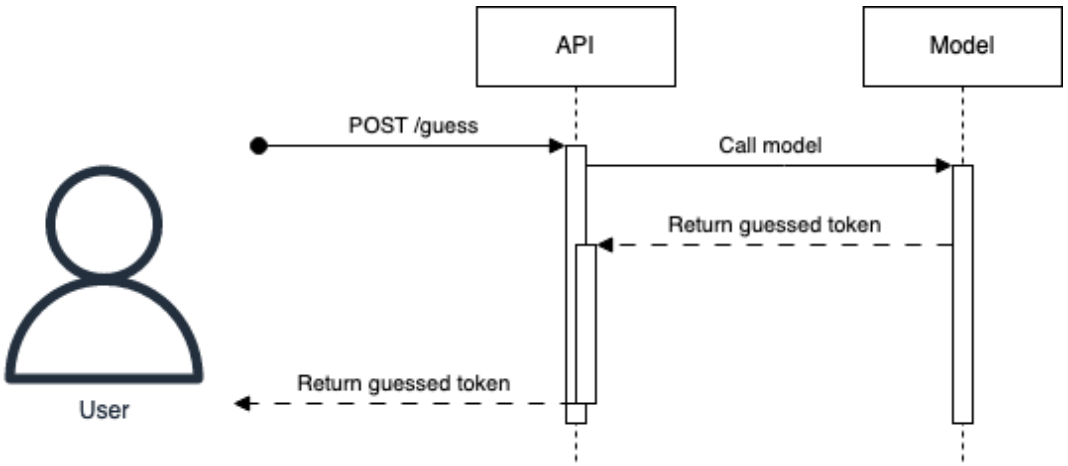
S obzirom da je proces dalje isti kao i kod preobuke, nije potrebno ponovo ga objasniti, učitavanje postojećeg modela dato je na slici 16, dok je petlja za treniranje identična kao kod preobuke.

```
from transformers import RobertaForMaskedLM
model = RobertaForMaskedLM.from_pretrained(f"./pre_trained/srberta_model")
```

*Slika 16: Učitavanje već istreniranog modela*

## 4.3 API

Kako bi istrenirani model bio dostupan za korišćenje, on je izložen putem API-ja. Kao što je rečeno za ovo je korišćena FastAPI biblioteka.
Proces pozivanja i dobijanja odgovora od modela je zamišljen da radi na sledeći način *(slika 17):*




*Slika 17: Proces pozivanja modela putem API-ja*

Kao što se vidi, korisnik poziva POST metodu na putanji */guess* koja poziva model i rezultat koji model predvidi za maskirani token se vraća korisniku
Metoda očekije da dobije objekat koji u sebi sadrži tekstualno polje, izgled tog objekta dat je na *slici 18*:

```
class Question(BaseModel):
    sentence: str
```

*Slika 18: Izgled objekta koji API očekuje*

Pre nego što se uopšte kreira POST metoda, potrebno je učitati model koji je dobijen treniranjem nad pravnim tekstovima, kao i tokenizator (slika 19).

```
tokenizer = RobertaTokenizerFast.from_pretrained("srberta_tokenizer")
model = RobertaForMaskedLM.from_pretrained("./srberta_law_model")
model.to('cpu')
```

*Slika 19: Učitavanje modela i tokenizatora*

FastAPI dolazi sa serverom u sebi pa je vrlo lako kreirati API za bilo koju potrebu. Nakon instaliranja FastAPI biblioteke, potrebno je inicijlizovati server i kreirati endpoint.

Takođe potrebno je omogućiti pristup serveru sa bilo koje adrese kako bi moglo nesmetano da se testira, naravno a ozbiljnije stvari ovde je ipak potrebno ograničiti pristup. Na narednim slikama dat je kod za sve navedene korake.

```
app = FastAPI()
origins = ["*"]
```

*Slika 20: Inicijalizacija servera*

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

*Slika 21: Podešavanje pristupa, dozvola pristupa sa svih adresa*

```
@app.post("/guess")
async def search(question: Question):
    fill = pipeline('fill-mask', model=model, tokenizer=tokenizer)
    fill(question.sentence)

    return fill
```

*Slika 22: POST endpoint i pozivanje modela*

Na slici 22 dat je izgled metode koja se koristi za obradu teksta i pogađanje tokena koje korisnik posašlje. Metoda je vrlo jednostavna:

- Prvo pripremi Pytorch pipeline i to fill-mask pipeline čiji je zadatak da pogodi maskirani token
- Njemu se proslede model i tokenizator
- Zatim mu se prosledi pitanje koje se nalazi unutar objekta poslatog od strane korisnika
- Na kraju se taj rezultat vrati korisniku

Ako se API-ju prosledi tekst kao na *slici 23*, dobija se odgovor kao na *slici 24*.

```
{
    "sentence": "ако код послодавца није основан синдика или ниједан синдикат не
        испуњава услове репрезентативности или није закључен <mask> о удруживању у
        складу са овим законом"
}
```

*Slika 23: Rečenica sa tokenom koji treba pogoditi*

```
[{'score': 0.5952515602111816,
  'token': 3187,
  'token_str': ' уговор',
  'sequence': '1) ако код послодавца није основан синдикат или ниједан синдикат не испуњава услове репрезентативности или није
закључен уговор о удруживању у складу са овим законом;'},
 {'score': 0.36653608083724976,
  'token': 4185,
  'token_str': ' споразум',
  'sequence': '1) ако код послодавца није основан синдикат или ниједан синдикат не испуњава услове репрезентативности или није
закључен споразум о удруживању у складу са овим законом;'},
```

*Slika 24: Objekat koji se vraća*

Na *slici 24* se vidi da je model vratio više opcija, na prvom mestu dao je reč **ugovor**, što je i tačna reč.

# 5 Analiza dobijenih rezultata

Prva stvar se odnosi na proces testiranja i ima za cilj da naglasi činjenicu da je testiranje jezičkog modelovanja vršeno tako što su uračunavana kao tačna samo ona predviđanja mreže koja se apsolutno poklapaju sa labelom koja stoji iza maskirane reči, pritom posmatrajući 5 najboljih skorova mreže na izlazu, za svaku maskiranu reč.
Treba se prisetiti da će BERT transformer mreža za analizu formmreža na izlazu za svaki od tokena iz ulazne sekvence, samim tim i za svaki maskirani token, dati skorove jednake dimenziji rečnika.

## 5.1 Rezultati modela za srpski jezik

Preobuka je rađena kao što je ranije rečeno u 19 epoha, rezultati metrike tačnosti date su na *slici 25*. Nakon 19 epoha treniranje je prekinuto jer više nije bilo značajnog napretka u tačnosti modela.

| Model iz epohe treniranja: | Vrednost metrike tačnosti |
|---|---|
| **0** | 56.2% |
| **1** | 63.6% |
| **2** | 66.4% |
| **4** | 69.3% |
| **6** | 70.8% |
| **8** | 71.9% |
| **10** | 72.5% |
| **12** | 73.1% |
| **14** | 73.3% |
| **16** | 73.5% |
| **17** | 73.7% |
| **18** | 73.7% |

*Slika 25: Metrike tačnosti po epohama za pre-training*

Ono što se može primetiti je da model najviše napreduje na početku treniranja, kako proces treniranja odmiče tako se i napredak smanjuje i jednom trenutku počne da stagnira.

Ukoliko posedujemo tekst na srpskom jeziku čije su reči u 15% slučajeva maskirane, naš model će biti u stanju da u 73.7% slučajeva predvidi identičnu reč (token) koji se krije iza nje, pri čemu će u mnogim drugim slučajevima biti u stanju da predloži i druge potencijalne zamene za maskiranu reč, naučene iz konteksta raznih dokumenata kroz koje je model prošao 19 puta u toku svoje obuke.

## 5.2 Rezultati modela nad pravnim tekstovima

Pre prikazivanja rezultata, važno je objasniti kako je do njih došlo. Merenje je vršeno na isti način kao u prethodnoj sekciji a model je fino podešavan na modelu iz epohe 19 sa preobuke. Da bi se objasnilo kako i ovde model napreduje, takođe je zbog eksperimenta odrađeno fino podešavanje nad više različitih modela sa preobuke, rezultati metrike tačnosti dati su na slici 26.

| SRBerta model iz epohe | Vrednost metrike tačnosti |
|---|---|
| **8** | 80.7% |
| **10** | 81.2% |
| **12** | 81.5% |
| **14** | 81.6% |
| **15** | 81.7% |
| **17** | 81.9% |
| **18** | 81.9% |

Slika 26: Metrike tačnosti za fino podešen model

Ono što se može zaključiti je: **Što je veća tačnost inicijalnog modela to će biti veća tačnost fino podešenog modela**.

# 6 Zaključak

Do sada u ljudskoj istoriji, jezik je bio privilegija samo ljudi - njegovo poznavanje, razumevanje i korišćenje. Međutim, brz razvoj veštačke inteligencije i obrade prirodnog jezika donosi promene u našem svakodnevnom životu. Internet pretraživači poput Google Search Engine-a, društvene mreže i različite aplikacije postaju sve napredniji zahvaljujući dubokom učenju. Ove tehnologije koriste složene modele kao što su BERT transformeri koji su prethodno analizirani i koriste mehanizme sopstvene pažnje.

Inspirisani prethodnim istraživanjima i eksperimentima, odlučili smo da istreniramo mrežu koja će razumeti srpski jezik i kontekst rečenica. Ovaj projekat, koji smo nazvali SRBerta, ima dvostruki cilj. Prvo, želimo da pokažemo uspeh mreže nakon treniranja na srpskom jeziku. Drugo, želimo da pružimo rešenje za analizu i ispravljanje tekstova zakona.

Rezultati su impresivni. Mreža SRBerta postigla je tačnost od skoro 85% prilikom testiranja na zakonodavnim tekstovima. Ovo snažno potvrđuje našu početnu pretpostavku da je moguće olakšati i automatizovati tradicionalne zadatke u Republici Srbiji. Ovaj sistem ne samo da uzima u obzir specifičnosti i pravila srpskog jezika, već i formalne tekstove pisane ćiriličnim pismom.

# 7 Literatura

[1] Literatura sa časova
[2] TensorFlow
[3] Pytorch
[4] https://huggingface.co/docs/transformers/main/tasks/masked_language_modeling
[5] https://huggingface.co/docs/transformers/index
[6] https://huggingface.co/docs/transformers/model_doc/roberta
[7] https://huggingface.co/blog/how-to-train
[8] https://towardsdatascience.com/news-category-classifica-on-fine-tuning-roberta-on-tpus-withtensorflow-f057c37b093