Transformer arhitektura

Salha, Ramal

Sveučilište Josipa Jurja Strossmayera u Osijeku
Fakultet primijenjene matematike i informatike
Sveučilišni prijediplomski studij Matematika i računarstvo

Ramal Salha

Transformer arhitektura

Završni rad

Osijek, 2023.

Sveučilište Josipa Jurja Strossmayera u Osijeku
Fakultet primijenjene matematike i informatike
Sveučilišni prijediplomski studij Matematika i računarstvo

Ramal Salha

Transformer arhitektura

Završni rad

Mentor: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2023.

Sažetak: U ovom ćemo radu objasniti kako radi transformer arhitektura. Kako možemo pojedine slojeve arhitekture sami implementirati u poznatoj biblioteci za izradu neuronskih mreža **PyTorch**. Nakon toga pokazat ćemo kako možemo iskoristiti našu implementaciju da izgradimo prevoditelj hrvatskog na engleski jezik.

Ključne riječi: Transformer arhitektura, PyTorch, Mehanizam pažnje

Transformer architecture

Abstract: In this work, we will explain how the transformer architecture works. We will also show how we can implement individual layers of the architecture ourselves in the well-known neural network library, **PyTorch**. Afterward, we will demonstrate how we can use our implementation to build a translator from Croatian to English.

Keywords: Transformer architecture, PyTorch, Attention

Sadržaj

1. Uvod

U posljednjem desetljeću, polje umjetne inteligencije svjedočilo je revolucionarnom napretku zahvaljujući razvoju različitih arhitektura neuronskih mreža. Jedna od najznačajnijih inovacija u tom kontekstu je transformer arhitektura, koja je postala ključna komponenta brojnih naprednih modela za obradu prirodnog jezika i drugih sekvencijalnih podataka. Transformer je posebno važan jer se pokazao boljim u usporedbi s tradicionalnim modelima poput rekurentnih neuronskih mreža **(RNN)** i dugotrajne memorije s povratnim petljama **(LSTM)** u obradi sekvencijalnih podataka.

Razlog zašto je **Transformer arhitektura** bolja od **RNN** i **LSTM** je u nizu ključnih inovacija koje donosi. Transformer koristi mehanizam pažnje s više glava koji omogućava modelu da paralelno obrađuje sve elemente sekvence, za razliku od **RNN** i **LSTM** koji obrađuju podatke sekvencijalno. Mehanizam ubrzava proces učenja i predviđanja. Također omogućava modelu da nauči složene odnose između različitih dijelova sekvence, čime se poboljšava sposobnost modela za razumijevanje konteksta.

U ovom radu ćemo prikazati kako implementirati transformer arhitekturu koristeći biblioteku PyTorch. Projekt možete pronaći na sljedećem linku github.com/RamalS/Transformer.

2. Arhitektura

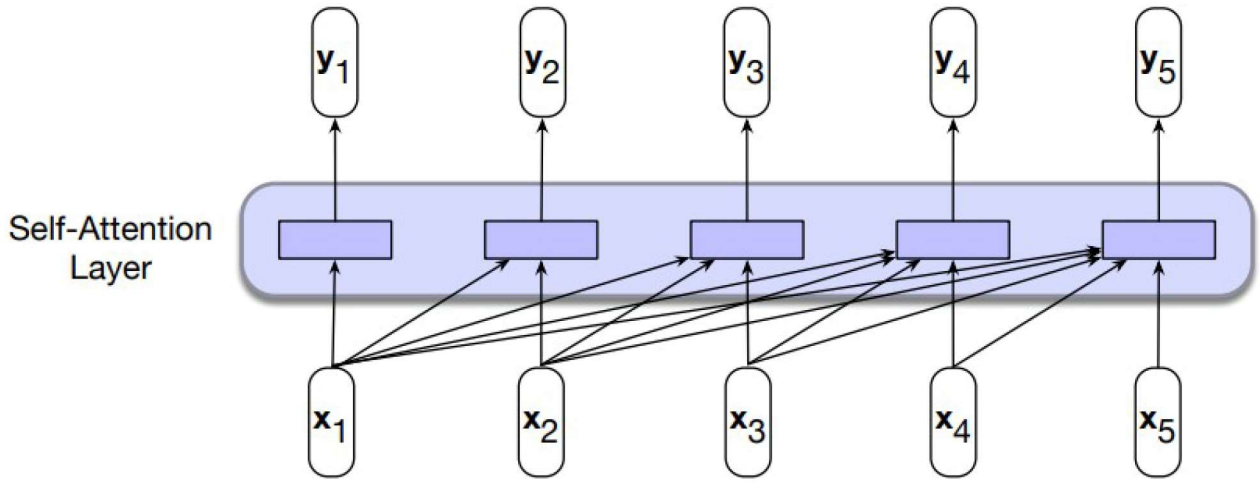
Transformer arhitektura [1, p. 2] se sastoji transformer blokova, gdje je svaki neuronska mreža napravljena od konbinacije linearnih slojeva, mreže prosljeđivanja i mehanizma pažnje što je ključna inovacija transformer arhitekture. Koristi se za sekvenca u sekvenca (eng. sequence to sequence) zadatke kao što su strojno prevođenje, sažimanje teksta, itd.. Kao i većina popularnih sekvencijalnih modela, transformer se sastoji od encoder-decoder strukture, gdje enkoder preslikava ulaz sekvenci $(x_1, ..., x_n)$ u niz $z = (z_1, ..., z_n)$. Za dani z, dekoder (eng. decoder) generira izlaz sekvenci $(y_1, ..., y_m)$. U svakom koraku model je auto-regresivan, odnosno koristi prijašnje generirane tokene kao dodatan ulaz pri generiranju sljedećeg.

3. Mehanizam pažnje

Mehanizam pažnje [1, p. 3] je ključna komponenta transformer arhitekture. Ima temeljnu ulogu u odlučivanju ovisnosti i odnosa između različitih elemenata u nizu podataka, kao što su riječi u rečenici. Slojevi mehanizama pažnje preslikavaju ulaz sekvence $(x_1, ..., x_n)$ u izlaznu sekvencu iste veličine $(y_1, ..., y_n)$.

Tijekom obrade svakog podatka na ulazu, model ima pristup svim elementima do i uključujući onaj koji se trenutno razmatra, ali nema pristup informacijama o kasnijim ulazima. Osim toga, računanje koje se obavlja za svaki element neovisno je o svim ostalim računanjima. Prvo svojstvo osigurava da ovaj pristup možemo koristiti za stvaranje jezičnih modela i koristiti ih za autoregresivnu generaciju, a drugo svojstvo nam omogućava da lagana paraleliziramo prosljeđivanje (eng. forward) i treniranje.

Srž pristupa baziranom na pažnji je mogućnost uspoređivanja jednog elementa sa drugim elementima na način da otkriva njihovu važnost u trenutnom kontekstu. U slučaju mehanizma pažnje, skup usporedbi odnosi se na druge elemente unutar danih sekvenci. Rezultat ovih usporedbi zatim se koristi za izračun izlaza za trenutni unos.



Slika 1: Prikaz kako informacije teku kroz mehanizam pažnje. Slika preuzeta s [3, p. 3]

Na primjer izračun y_3 je baziran na usporedbama između ulaza x_1, x_2 i x_3. Najjednostavniji oblik usporedbe između elemenata u mehanizmu pažnje je unutarnji produkt. Rezultat unutarnjeg produkta je skalar vrijednosti između $-\infty$ do ∞, gdje veći rezultat znači da su sličniji vektori koji se uspoređuju. Da bi bolje iskoristili ove ocjene, normaliziramo ih softmax-om kako bi napravili vektore težina α_{ij}, koji prikazuju proporcijalnu značajnost svakog ulaznog elementa j naspram odabranog i koji je trenutni fokus pažnje.

$$\alpha_{ij} = \text{softmax}(x_i \cdot x_j), \forall j \leq i$$

Uz dobivene proporcionalne ocjene u λ-ama, generiramo izlaznu vrijednost y_i tako da uzmemo sumu svih ulaza koje smo dosada vidjeli i njihovih odgovarajućih λ.

$$y_i = \sum_{j \leq i} \lambda_{ij} x_j$$

3.1. Skalirani unutarnji produkt (eng. Scaled Dot-Product Attention)

Prethodni koraci su srž mehanizma pažnje, no kako bi ju transformer mogao trenirati, sadrži sljedeće parametre u obliku matrica težina koje označavamo sa W^Q, W^K i W^V.

- Upit (eng. Query) trenutni fokus pažnje.
- Ključ (eng. Key) uspoređujemo sa upitom.
- Vrijednost (eng. Value) služi za računanje izlaza upita.

Ove težine se koriste za izračunavanje linearnih transformacija ulaza x u njihovim odgovarajućim ulogama.

$$q_i = W^Q x_i$$
$$k_i = W^K x_i$$
$$v_i = W^V x_i$$

Ocjena elementa u trenutnom fokusu, x_i i elementa iz prethodnog konteksta x_j je sastavljena od vektorskog produkta između vektora upita q_i i vektora ključa k_j prethodnog elementa. Sada usporedbu računamo kao unutarnji produkt q_i i k_j, a izlaz računanja y_i je sada baziran na težinskoj sumi preko vektora vrijednosti v.

$$y_i = \sum_{j \leq i} \lambda_{ij} v_j$$

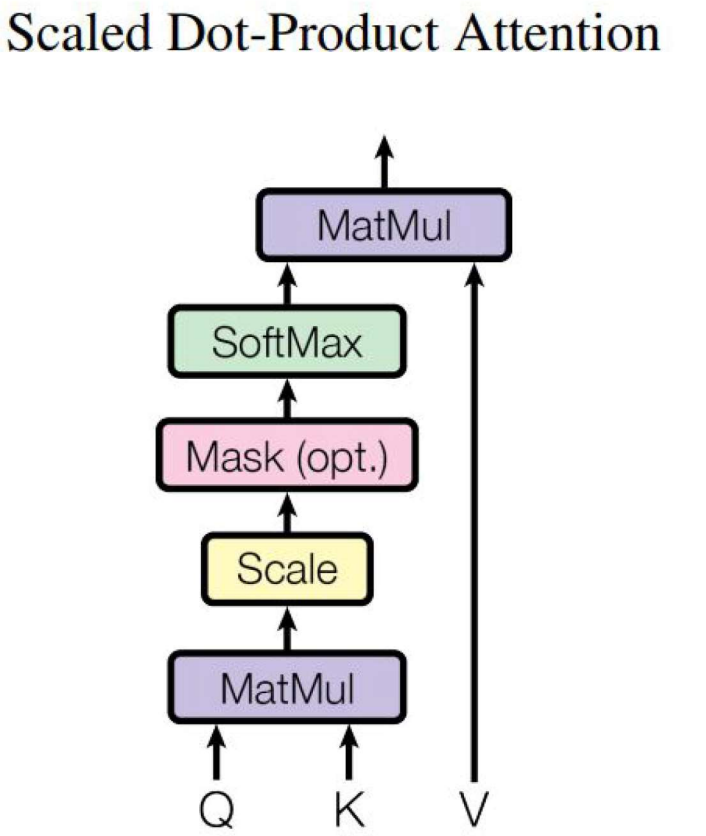
Zbog korištenja skalarnog produkta za uspoređivanje u kombinaciji s softmaxom, javljaju se problemi prilikom exponenciranja. Pošto je rezultat unutarnjeg produkta proizvoljno velik, eksponenciranje tako velikih vrijednosti može uzrokovati numeričke probleme u slučaju pozitivnih vrijednosti, ili u slučaju negativnih može rezultirati u nulu. Kako bi izbjegli ovaj problem s gradijentima, rezultat unutarnjeg produkta $q_i \cdot k_j$ dijelimo s korijenom dimenzije od vektora upita i ključa $\sqrt{d_k}$.

Ovaj proces mehanizma pažnje je bio iz perspektive računanja jednog izlaza u pojedinom trenutku, ali pošto se svi izlazi y_i računaju neovisno jedan o drugom, ovaj proces možemo paralelizirati. Ovo možemo postići grupiranjem ulaznih elemenata u jednu matricu X i množeći ju s matricom ključeva W^K, upita W^Q i vrijednosti W^V.

- $Q = W^Q X$ predstavlja upite za koje želimo izračunati ocjene pažnje dimenzije d_k.
- $K = W^K X$ predstavlja ključeve koje uspoređujemo sa upitima dimenzije d_k
- $V = W^V X$ predstavlja vrijednosti povezane sa svakim ključem dimenzije d_v

Uz gornje matrice možemo izračunati sve usporedbe istovremeno množeći Q i K^T u jednom matričnom množenju, skalirati njihove ocjene i pomnožiti rezultat s V

$$Pažnja(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



Slika 2: Prikz skalarnog unutarnjeg produkta. Slika preuzeta s [1, p. 4]

Primjer 3.1. Primjer matrice pažnje

	Hello	how	are	you
Hello	0.8	0.1	0.05	0.05
how	0.1	0.6	0.2	0.1
are	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

Tablica 1: Primjer matrice pažnje

Primjetimo kako zbroj redaka čini 1.

```
def scaled_dot_product_attention(self, Q, K, V, mask=None):
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    attention = torch.softmax(scores, dim=-1)

    context = torch.matmul(attention, V)

    return context
```

3.2. Mehanizam pažnje s više glava (eng. Multi head attention)

Različite riječi u rečenici mogu imati različite odnose istovremeno. Teško je naučiti samo jedan sloj mehanizma pažnje različite vrste odnosa među svojim ulazima. Transformer ovaj problem rješava s mehanizmom pažnje s više glava[3, p. 7]. Sastoji se od skupa slojeva mehanizama pažnje (glave), koji se nalaze na paralelnim slojevima na istoj dubini.

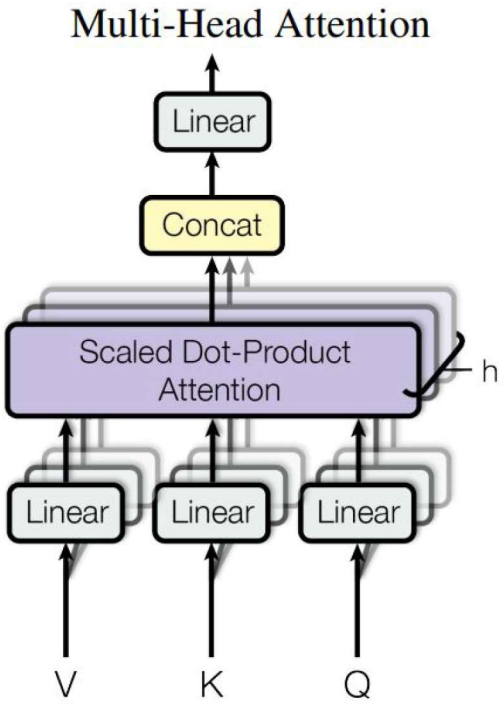
Svaka glava i, u sloju samopažnje ima svoj skup matrica upit W_i^Q, ključ W_i^K i vrijednosti W_i^V. Koriste se kako bi projekcirali ulaz u sloj x_i odvojeno za svaku glavu, gdje ostatak računanja mehanizma pažnje ostaje nepromjenjen. Izlaz mehanizma pažnje s h glava, rezultira s h vektora iste duljine koji se konkateniraju, te linearnom projekcijom realiziramo matricu W^O

$$\text{VišeGlava}(Q, K, V) = \text{Concat}(head_1, ..., head_h)W^O$$

$$\text{gdje je } head_i = Pažnja(QW_i^Q, KW_i^K, VW_i^V)$$

,

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k},$$
$$W_i^K \in \mathbb{R}^{d_{model} \times d_k},$$
$$W_i^V \in \mathbb{R}^{d_{model} \times d_v}\text{i}$$
$$W_i^O \in \mathbb{R}^{hd_v \times d_{model}}$$



Slika 3: Prikaz mehanizma pažnja s više glava. Slika preuzeta s [1, p. 4]

Kako bi implementirali pažnju s više glava potrebno je definirati tri linearna sloja W^Q, W_K i W_V. S d_{model} definiramo veličinu našeg modela, num_{heads} je broj glava, a d_k definiramo kao $d_k = \frac{d_{model}}{num_{heads}}$

split_heads pomoćna funkcija zadužena je za razdvajanje matrica kako bi proveli računanje pažnje za svaku glavu.

combine_heads pomoćna funkcija zadužena za vraćanje rezultata u jednu matricu.

Kada smo pripremili sve potrebne pod slojeve, u **forward** funkciji možemo složiti sve prema slici.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_Q = nn.Linear(d_model, d_model)
        self.W_K = nn.Linear(d_model, d_model)
        self.W_V = nn.Linear(d_model, d_model)

        self.output_linear = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # matmul Q and K and scale
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # apply mask
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # softmax layer
        attention = torch.softmax(scores, dim=-1)

        # matmul attention and V
        context = torch.matmul(attention, V)

        return context

    def split_heads(self, x):
        batch_size, seq_len, _ = x.size()

        return x.view(batch_size,
                      seq_len,
                      self.num_heads,
                      self.d_k).transpose(1, 2)
```

```
def combine_heads(self, x):
    batch_size, _, seq_len, _ = x.size()

    return x.transpose(1, 2).contiguous().view(batch_size, seq_len, -1)

def forward(self, Q, K, V, mask=None):
    # batch_size x seq_len x d_model
    Q = self.W_Q(Q)
    K = self.W_K(K)
    V = self.W_V(V)

    # batch_size x num_heads x seq_len x d_k
    Q = self.split_heads(Q)
    K = self.split_heads(K)
    V = self.split_heads(V)

    attention = self.scaled_dot_product_attention(Q, K, V, mask)

    # batch_size x seq_len x d_model
    attention = self.combine_heads(attention)

    output = self.output_linear(attention)

    return output
```

3.3. Ulazno kodiranje (eng. Input embedding)

Ključan korak u obradi prirodnog jezika. Uključuje pretvaranje tokena u vektore te omogućuje neuronskim mrežama učinkovitu obradu tekstualnih podataka. Tokeni mogu predstavljati različite elemente poput riječi, slova, itd.. Ovaj proces započinje tokenizacijom, gdje se tekst dijeli na tokene i preslikava u cijelobrojne identifikatore (id), koji se zatim pretvaraju u vektore.

3.4. Pozicijsko kodiranje (eng. Positional encoding)

Budući da naš model nema informaciju u kojem redoslijedu se nalaze tokeni u sekvenci, kao rješenje ovog problema, uvodimo pozicijsko kodiranje[1, p. 5] koje dodaje potrebnu informaciju o redoslijedu tokena u nizu. Kako pozicijski koder ima istu dimenziju kao i ulazni koder, d_{model}, zbrajamo ga sa slojem ulaznog kodiranja.

Postoje razne mogućnosti za odabiranje pozicijskoj kodiranja. Popularno rješenje je sa funkcijama **sin** i **cos**.

$$PE(pos, 2i) = sin(pos/1000^{2i/d_{model}})$$
$$PE(pos, 2i+1) = cos(pos/1000^{2i/d_{model}})$$

gdje je *pos* pozicija, a i dimenzija. Izlaz pozicijskog kodiranja je ulaz pažnje s više glava.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length,
                                dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2)
                             .float() * -(math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

3.5. Mreža prosljeđivanja (eng. Feedforward network)

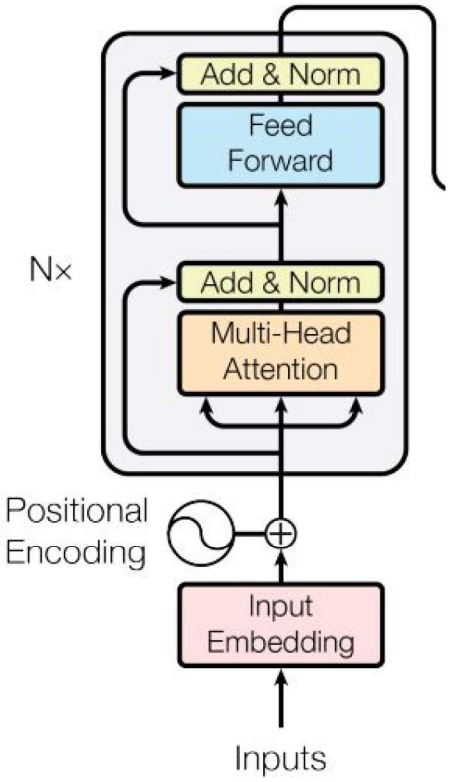
Enkoder i dekoder u sebi sadrže potpuno povezanu mrežu prosljeđivanja. Sastoji se od dva linearna sloja i aktivacijske funkcije **ReLU** između njih. Mreža prosljeđivanja [1, p. 5] je poziciona mreža, svaka pozicija se obrađuje zasebno i na indentičan način.

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

```
class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()

        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.linear1(x) # d_model x d_ff
        x = torch.relu(x)
        x = self.linear2(x) # d_ff x d_model

        return x
```

3.6. Enkoder (eng. Encoder layer)

Enkoder [1, p. 2] se sastoji od N jednakih slojeva, od kojih svaki sloj ima dva podsloja. Prvi je sloj pažnje s više glava, a drugi je pozicijski sloj. Također postoji zaostala veza koja okružuje svaki glavni podsloj. Ove veze prenose neobrađeni ulaz x podsloja u funkciju normalizacije sloja. Izlaz svakog sloja je **SlojNormalizacije(x + PodSloj(x))**, gdje je **PodSloj(x)** funkcija implementirana od strane podsloja (Sloja pažnje s više glava i pozicijskoj sloja).

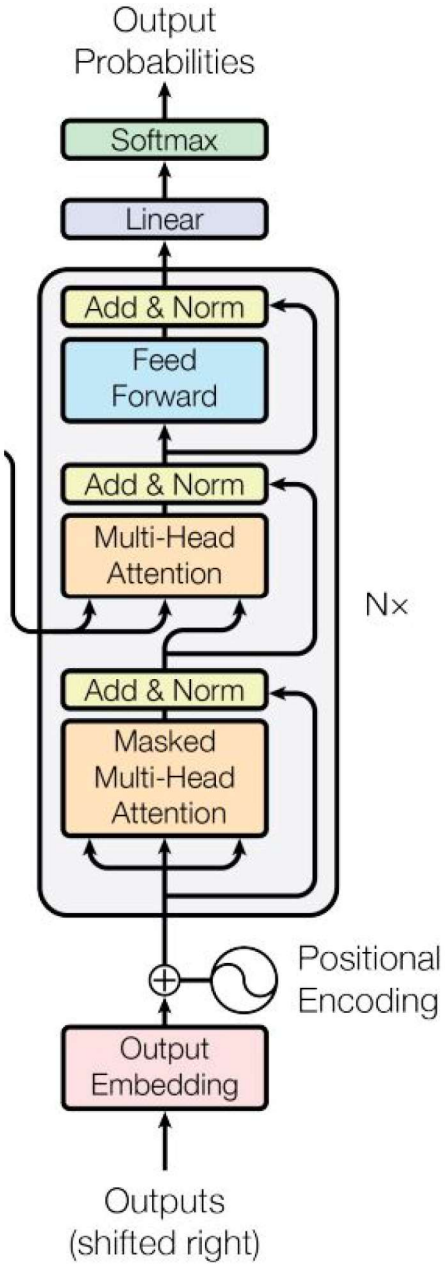


Slika 4: Prikaz enkodera opisanog gore. Slika preuzeta s [1, p. 3]

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()

        self.multi_head_attention = MultiHeadAttention(d_model, num_heads)
        self.position_wise_feed_forward = PositionWiseFeedForward(d_model
                                                                  , d_ff)

        self.layer_norm1 = nn.LayerNorm(d_model)
        self.layer_norm2 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # multi-head attention
        attention = self.multi_head_attention(x, x, x, mask)

        # add and norm
        x = self.layer_norm1(x + attention)
        x = self.dropout1(x)

        # position-wise feed forward
        feed_forward = self.position_wise_feed_forward(x)

        # add and norm
        x = self.layer_norm2(x + feed_forward)
        x = self.dropout2(x)

        return x
```

3.7. Dekoder (eng. Decoder layer)

Dekoder [1, p. 3] se, isto kao enkoder, sastoji od N jednakih dijelova. No, za razliku od enkodera, dekoder ima drugi podsloj koji koristi pažnju s više glava preko izlaza enkodera. Također prvom podsloju dodajemo masku da prilikom učenja ne vidimo buduće dijelove niza koje još nismo predvidjeli. Kao i u enkoderu, postoji zaostala veza podslojeva i normalizacijskog sloja.



Slika 5: Prikaz dekodera opisanog gore. Slika preuzeta s [1, p. 3]

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super().__init__()

        self.multi_head_attention1 = MultiHeadAttention(d_model, num_heads)
        self.multi_head_attention2 = MultiHeadAttention(d_model, num_heads)

        self.position_wise_feed_forward = PositionWiseFeedForward(d_model,
                                                                  d_ff)

        self.layer_norm1 = nn.LayerNorm(d_model)
        self.layer_norm2 = nn.LayerNorm(d_model)
        self.layer_norm3 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

    def forward(self, x, encoder_output, src_mask=None, trg_mask=None):
        # masked multi-head attention
        masked_attention = self.multi_head_attention1(x, x, x, trg_mask)

        # add and norm
        x = self.layer_norm1(x + masked_attention)
        x = self.dropout1(x)

        # multi-head attention
        attention = self.multi_head_attention2(x, encoder_output,
                                               encoder_output, src_mask)

        # add and norm
        x = self.layer_norm2(x + attention)
        x = self.dropout2(x)

        # position-wise feed forward
        feed_forward = self.position_wise_feed_forward(x)

        # add and norm
        x = self.layer_norm3(x + feed_forward)
        x = self.dropout3(x)

        return x
```

3.8. Transformer

Nakon što smo implementirali enkoder, dekoder i sve potrebne podslojeve možemo implementirati transformer. Na ulazu stavljamo ulazni koder čiji izlaz prosljeđujemo pozicijskom enkoderu. Isto radimo i za izlazne podatke kod dekodera. Zatim izlaz enkodera tada prosljeđujemo dekoderu zajedno s izlazom dekodera iz prijašnje iteracije. Enkoder se niže jedan na drugi N puta, isto kao i dekoder sloj. Nakon što se sve izvršilo, izlazne podatke dekodera prosljeđujemo linearnom sloju. Transformer u konstruktoru prima veličine vokabulara ulaznih i izlaznih podataka, dimenziju modela, broj glava, broj slojeva, dimenziju mreže prosljeđivanja i maksimalnu duljinu sekvence.

```
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads,
                 num_layers, d_ff, max_seq_length, dropout):
        super().__init__()

        self.src_embedding = nn.Embedding(src_vocab_size, d_model)
        self.tgt_embedding = nn.Embedding(tgt_vocab_size, d_model)

        self.positional_encoding = PositionalEncoding(d_model
                                                       , max_seq_length)

        self.encoder_layers = nn.ModuleList([
            EncoderLayer(
                d_model,
                num_heads,
                d_ff,
                dropout) for _ in range(num_layers)])
        self.decoder_layers = nn.ModuleList([
            DecoderLayer(
                d_model,
                num_heads,
                d_ff,
                dropout) for _ in range(num_layers)])

        self.linear = nn.Linear(d_model, tgt_vocab_size)

    def generate_mask(self, src, tgt):
        src_mask = (src != 0).unsqueeze(1).unsqueeze(2).to(device)
        tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3).to(device)
        seq_length = tgt.size(1)
        nopeak_mask = (1 - torch.triu(
            torch.ones(1,
                       seq_length,
                       seq_length), diagonal=1)).bool().to(device)
        tgt_mask = tgt_mask & nopeak_mask

        return src_mask, tgt_mask
```

```
def forward(self, src, tgt):
    src_mask, tgt_mask = self.generate_mask(src, tgt)

    src = self.src_embedding(src)
    tgt = self.tgt_embedding(tgt)

    src = self.positional_encoding(src)
    tgt = self.positional_encoding(tgt)

    for layer in self.encoder_layers:
        src = layer(src, src_mask)

    for layer in self.decoder_layers:
        tgt = layer(tgt, src, src_mask, tgt_mask)

    output = self.linear(tgt)

    return output
```

4. Prevoditelj hrvatskog na engleski jezik

U ovoj sekciji ćemo iskoristiti našu implementaciju transformera kako bismo napravili prevoditelj hrvatskog jezika na engleski. Promotrit ćemo potrebne procese kao što su obrada podataka, tokenizacija riječi i treniranje.

4.1. Učitavanje i obrada skupa podataka za treniranje

Prvo što trebamo napraviti je odabrati skup podataka koji u sebi sadrži hrvatske i engleske rečenice. Rečenice ćemo spremiti u dvije liste, jednu listu za hrvatski jezik, a drugu listu za engleski jezik. Za svaki jezik potrebno je definirati vokabular. Vokabular je skup tokena koje će naš model primati na ulazu i predviđati. Vokabular možemo definirati na više načina. U ovom radu ćemo uzeti za jedan token jedan karakter.

START - token koji označava početak niza
END - token koji označava kraj niza
UNK - token koji označava karakter koji se ne nalazi u vokabularu
PAD - token koji označava prazan prostor, odnosno mjesta koja nismo popunili

```
hr_flat_tokens = ['\n', '!', '"', '#', '$', '%', '&', "'",
                  '(', ')', '*', '+', ',', '-', '.', '/',
                  '0', '1', '2', '3', '4', '5', '6',
                  '7', '8', '9', ':', '<', '=', '>',
                  '?', '@', '[', '\\', ']', '^', '_',
                  '`', 'a', 'b', 'c', 'd', 'e', 'f',
                  'g', 'h', 'i', 'j', 'k', 'l', 'm',
                  'n', 'o', 'p', 'q', 'r', 's', 't',
                  'u', 'v', 'w', 'x', 'y', 'z', '{',
                  '|', '}', '~', '¡', '¢', '£', '¤', '¥', '¦', '§']

en_flat_tokens = ['\n', '!', '"', '#', '$', '%', '&', "'",
                  '(', ')', '*', '+', ',', '-', '.', '/',
                  '0', '1', '2', '3', '4', '5', '6',
                  '7', '8', '9', ':', '<', '=', '>',
                  '?', '@', '[', '\\', ']', '^', '_',
                  '`', 'a', 'b', 'c', 'd', 'e', 'f',
                  'g', 'h', 'i', 'j', 'k', 'l', 'm',
                  'n', 'o', 'p', 'q', 'r', 's', 't',
                  'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~']

# Count character frequencies
hr_vocab_counter = collections.Counter(hr_flat_tokens)
en_vocab_counter = collections.Counter(en_flat_tokens)

# Special tokens and vocabulary
special_tokens = ["<PAD>", "<UNK>", "<START>", "<END>"]
hr_vocab = special_tokens + sorted(hr_vocab_counter.keys())
en_vocab = special_tokens + sorted(en_vocab_counter.keys())
```

4.2. Pretvorba rečenica u tokene

Kako bi pretvorili riječi u tokene i obratno koristit ćemo strukturu podataka rječnik, gdje će nam ključevi biti tokeni, a vrijednosti karakter u slučaju pretvorbe tokena u karakter, a obratno u slučaju pretvorbe karaktera u tokene. Također ćemo definirati funkciju **sentance2index** koja će pretvoriti riječi u niz tokena.

```
# Function for sentence to index conversion
def sentence2index(src, sentence_tokens, sentence_length,
                   start_token=False, end_token=False):
    token2index = hr_token2index if src == "hr" else en_token2index

    if start_token:
        sentence_index = [token2index["<START>"]]
    else:
        sentence_index = []

    sentence_index += [token2index[token.lower()]
                       if token.lower() in token2index
                       else token2index["<UNK>"]
                       for token in list(sentence_tokens)]

    if end_token:
        sentence_index.append(token2index["<END>"])

    sentence_index += [token2index["<PAD>"]
                       for _ in
                       range(sentence_length - len(sentence_index))]

    return sentence_index
```

4.3. Izrada skupa podataka za treniranje

Radi lakšeg treniranja podataka u manjim skupinama, izradit ćemo **TranslationDataset** klasu koja nasljeđuje klasu **Dataset** iz **PyTorch** biblioteke. Klasa će primiti neobrađene skupove podataka hrvatskog i engleskog jezika, maksimalnu duljinu karaktera koju možemo generirati i broj podataka koje želimo uzeti za generiranje koda.

```
class TranslationDataset(torch.utils.data.Dataset):
    def __init__(self, hr_dataset, en_dataset, max_seq_length, take):
        hr, en = self.remove_long_sentences(hr_dataset,
                                            en_dataset,
                                            max_seq_length)

        train_data = []
        test_data = []

        for tokens in hr[:take]:
            train_data.append(sentence2index("hr",
                                             tokens,
                                             max_seq_length))

        for tokens in en[:take]:
            test_data.append(sentence2index("en",
                                            tokens,
                                            max_seq_length,
                                            True,
                                            True))

        train_data = torch.tensor(train_data)
        test_data = torch.tensor(test_data)

        self.X = train_data
        self.y = test_data

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

    def remove_long_sentences(self, hr, en, max_length):
        hr_short = []
        en_short = []

        print(hr[0])

        for c, e in zip(hr, en):
            if len(c) < max_length - 2 and len(e) < max_length - 2:
```

```
                hr_short.append(c)
                en_short.append(e)

        return hr_short, en_short
```

Inicijaliziramo naš **TranslationDataset** i podijelimo podatke na skup podataka za treniranje, validaciju i testiranje.

```
translation_dataset = TranslationDataset(hr, en, max_seq_length, 3000)
lengths = [int(0.8 * len(translation_dataset)),
           int(0.1 * len(translation_dataset)),
           len(translation_dataset) -
           int(0.8 * len(translation_dataset)) -
           int(0.1 * len(translation_dataset))]
train_dataset, dev_dataset, test_dataset = random_split(translation_dataset,
                                                        lengths=lengths)

print(f'Veli ina skupa za treniranje: {len(train_dataset)}')
print(f'Veli ina skupa za validaciju: {len(dev_dataset)}')
print(f'Veli ina skupa za testiranje: {len(test_dataset)}')


train_loader = DataLoader(train_dataset,
                          batch_size=batch_size, shuffle=True)
dev_loader = DataLoader(dev_dataset,
                        batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset,
                         batch_size=batch_size, shuffle=False)
```

4.4. Treniranje na jednom podskupu

Prvo ćemo implementirati funkciju za treniranje za prezentacijske svrhe kako bismo vidjeli na manjem broju podataka kako se transformer ponaša.

Prije svega potrebno je inicijalizirati transformer.

```
src_vocab_size = len(hr_vocab)
tgt_vocab_size = len(en_vocab)
d_model = 512
num_heads = 8
num_layers = 6
d_ff = 2048
max_seq_length = 200
dropout = 0.1

batch_size = 8
num_epochs = 200

X_batch, y_batch = next(iter(train_loader))
X_batch = X_batch.to(device)
y_batch = y_batch.to(device)

transformer = Transformer(src_vocab_size,
                          tgt_vocab_size,
                          d_model,
                          num_heads,
                          num_layers,
                          d_ff,
                          max_seq_length,
                          dropout)
transformer = transformer.to(device)
```

Nakon toga definirati ćemo funkciju **train_one_batch** koja se koristi za treniranje transformer modela.

Unutar funkcije se koristi **CrossEntropyLoss** kao funkcija gubitka za usporedbu stvarnih izlaza **(y)** i izlaza modela **(o)**. Optimizacijski algoritam **Adam** se koristi za prilagodbu parametara modela kako bi se minimalizirala funkcija gubitka.

Petlja se izvodi kroz zadani broj epoha **(num_epochs)** i u svakoj se epohi računa izlaz modela na temelju ulaznih podataka **(X_batch)** i stvarnih izlaza **(y_batch)**. Zatim se gubitak izračunava usporedbom izlaza modela i stvarnih izlaza.

```
def train_one_batch(transformer,
                    X_batch,
                    y_batch,
                    num_epochs,
                    tgt_vocab_size,
                    leraning_rate=0.0001):
    criterion = nn.CrossEntropyLoss(ignore_index=0)
    optimizer = torch.optim.Adam(transformer.parameters(),
                                 lr=leraning_rate,
                                 betas=(0.9, 0.98), eps=1e-9)
    progress_bar = tqdm(range(num_epochs), position=0, leave=True)

    transformer.train()

    for epoch in progress_bar:
        optimizer.zero_grad()

        output = transformer(X_batch, y_batch[:, :-1])

        o = output.contiguous().view(-1, tgt_vocab_size)
        y = y_batch[:, 1:].contiguous().view(-1)

        loss = criterion(o, y)
        loss.backward()
        optimizer.step()

        progress_text = f"Epoch {epoch + 1} | Train Loss: {loss.item()}"
        progress_bar.set_description(progress_text)

train_one_batch(transformer, X_batch, y_batch, num_epochs, tgt_vocab_size)
```

4.5. Funkcija prevođenja

Kako bismo mogli koristiti naš model potrebno je implementirati funkciju prevođenja. Ova funkcija prima rečenicu i naš natrenirani model. Funkcija tokenizira našu proslijeđenu rečenicu i kreira tokeniziranu praznu rečenicu za engleski jezik. Tokenizirana rečenica engleskog jezika će imati samo početni START token kako bi se označio početak rečenice i naglasio transformeru da ovdje treba krenuti generirati nove tokene. Zatim postavljamo petlju koja će se izvršiti maksimalno broj puta koliki nam je najveći broj tokena ili dok ne dođe do END tokena. Unutar petlje uzimamo generirani token na i-tom indeksu i pridružujemo ga generiranom nizu.

```
def translate(sentance, model):
    result = ""
    src = torch.tensor([sentence2index("hr",
                                       sentance,
                                       max_seq_length)]).to(device)

    for i in range(max_seq_length):
        tgt = torch.tensor([sentence2index("en",
                                           result,
                                           max_seq_length,
                                           start_token = True)]).to(device)

        output = model(src, tgt)

        next_word_prob = output[0][i]
        next_word_index = torch.argmax(next_word_prob).item()
        next_word = en_index2token[next_word_index]

        if next_word == "<END>":
            break

        result += next_word

    return result

sent = "sada ima toliko novca da mo e kupiti sve, sve to god za eli!"
translate(sent, transformer)
```

Funkciji **translate** smo proslijedili rečenicu *"sada ima toliko novca da može kupiti sve, sve što god zaželi!"*, kao rezultat je vratio *"now he has so much money that he can buy everything, everyth"*. Možemo primijetiti da prijevod nije savršen, no većim brojem epoha, kvalitetnijim skupom podataka i boljim odabirom parametara transformera možemo poboljšati rezultate.

4.6. Učitavanje i spremanje modela

Kako ne bi izgubili naš trenirani model potrebno je implementirati **save_transformer** i **load_transformer** funkcije. Funkcija **save_transformer** sprema sve težine u **.pt** datoteku, a korištene parametre prilikom treniranja u **.json** datoteku. Za obje datoteke dodijelimo jedinstveni indetifikator kojeg ćemo proslijediti **load_transformer** funkciji kako bi učitali spremljeni model.

```
def save_transformer(transformer):
    path = "./models"

    model_id = str(uuid.uuid4())[:8]

    model_args = { "src_vocab_size": len(hr_vocab),
                   "tgt_vocab_size": len(en_vocab),
                   "d_model": d_model,
                   "num_heads": num_heads,
                   "num_layers": num_layers,
                   "d_ff": d_ff,
                   "max_seq_length": max_seq_length,
                   "dropout": dropout }

    json_object = json.dumps(model_args, indent=4)

    torch.save(transformer.state_dict(), path + f'/{model_id}.pt')

    with open(path + f'/{model_id}.json', "w") as outfile:
        outfile.write(json_object)

    return model_id

save_transformer(transformer)

def load_transformer(model_id):
    with open(f"./models/{model_id}.json", "r") as file:
        model_args = json.load(file)

    transformer = Transformer(**model_args)
    transformer.load_state_dict(torch.load(f"./models/{model_id}.pt"))
    transformer = transformer.to(device)
    transformer.eval()

    return transformer
```

4.7. Aplikacija

U ovom radu smo pokazali kako implemenirati svoju transformer arhitekturu te kako je natrenirati da prevodi s hrvatskog na engleski jezik. Kako bi objedinio sve u jednu cjelinu, napravit ćemo jednostavnu aplikaciju u pythonu uz pomoć **streamlit** biblioteke. Potrebno je učitati spremljeni model kojeg smo ranije natrenirali. Prilikom spremanja smo dobili jedinstveni indentifikator modela kojeg trebamo prosljediti kao parametar funkciji **load_transformer**. Nakon toga inicijalizirat ćemo tekstualno polje u kojem će korisnik moći upisati rečenicu koju želi prevesti na hrvatski jezik. Nakon što korisnik pritisne enter pozovemo **translate** funkciju kojoj prosljedimo unesenu rečenicu i učitani model.

Slika 6: Prikaz aplikacije

```
import streamlit as st
from transformer import load_transformer, translate

saved_transformer = load_transformer("2e98b75a") # model name

st.title('Prevoditelj sa hrvatskog na engleski')
st.caption("""Unesi re enicu na hrvatskom jeziku i pritisni enter.
  Re enica  e  biti prevedena na engleski jezik.""")

sentence = st.text_input('Input sentence', '')

if sentence:
    translated = translate(sentence, saved_transformer)
    st.write("**Prijevod re enice:** ", translated)
```

5. Zaključak

U ovom radu obradili smo transformer arhitekturu, pokazali smo što je to mehanizam pažnje te kako ju izračunati uz pomoć skalarnog unutarnjeg produkta. Pokazali smo implementacije pojedinih slojeva koji čine transformer arhitekturu. Pokazali smo kako možemo obraditi podatke, tokenizirati rečenice i trenirati transformer. Implementirani transformer smo natrenirali kako bi prevodio hrvatski na engleski jezik.

Literatura

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., Polosukhin, I. , *Attention is all you need, 2017.*

[2] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam , *Automatic differentiation in PyTorch, 2017.*

[3] Daniel Jurafsky James H. Martin , *Speech and Language Processing, 2023.*