Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Дисциплина: Объектно-ориентированное программирование

Тема: Построение и изучение работы модели FNet

Выполнил студент гр. 3331506/10401 Бураев Н.О.
Преподаватель Ананьевский М.С.

Санкт-Петербург

2024

Оглавление

1. Введение

Актуальность темы

В последние годы наблюдается значительное развитие технологий обработки естественного языка (NLP), что открывает новые возможности для автоматизации задач, связанных с текстом. Одной из ключевых задач в этой области является генерация текста, которая находит применение в создании чат-ботов, автоматическом переводе, создании контента и других областях. С развитием глубокого обучения и архитектуры трансформеров, качество генерации текста значительно улучшилось. Однако высокая вычислительная сложность традиционных трансформеров побуждает исследователей искать более эффективные решения.

Цели и задачи работы

Цель данной курсовой работы – изучить модель FNet для генерации текста, провести её реализацию на языке Python и проанализировать её эффективность по сравнению с традиционными моделями на базе трансформеров. Для достижения этой цели необходимо решить следующие задачи:

1. Изучить теоретические основы модели FNet и её отличия от стандартных трансформеров.
2. Провести выбор и подготовку данных для обучения модели.
3. Реализовать модель FNet с использованием Python и соответствующих библиотек.
4. Обучить модель и провести тестирование генерации текста.
5. Проанализировать результаты и сравнить их с результатами, полученными с помощью других моделей.

Обзор содержания

Первая глава работы посвящена теоретическим основам, где рассматриваются основные подходы к генерации текста и подробно разбирается архитектура трансформеров и модели FNet. Во второй главе описывается методология, включая выбор данных, используемые инструменты и библиотеки, а также алгоритм реализации модели FNet. В третьей главе представлена практическая реализация: от подготовки данных до обучения модели и генерации текста. В четвертой главе анализируются результаты, обсуждаются преимущества и недостатки модели FNet, а также проводится сравнение с другими подходами. В заключении подводятся итоги проделанной работы и обозначаются перспективы дальнейших исследований.

2. Теоретические основы

Модели генерации текста

Генерация текста является одной из ключевых задач в области обработки естественного языка (NLP). Эта задача включает в себя создание осмысленных и грамматически правильных текстов на основе заданных входных данных. В ранних подходах к генерации текста использовались статистические методы, такие как модели N-грамм и методы на основе скрытых марковских моделей (HMM). Однако с развитием глубокого обучения на смену этим методам пришли нейронные сети.

Среди нейронных сетей важную роль играют рекуррентные нейронные сети (RNN) и их модификации, такие как LSTM и GRU, которые способны обрабатывать последовательности данных. Тем не менее, RNN имеют свои ограничения, такие как сложность обучения и проблемы с долгосрочной зависимостью.

Архитектура трансформеров

Значительный прогресс в генерации текста был достигнут с появлением архитектуры трансформеров, представленной в работе "Attention is All You Need" (Vaswani et al., 2017). Трансформеры решают проблемы RNN, используя механизм внимания (attention), который позволяет моделям учитывать все слова в последовательности одновременно, а не по одному.

Трансформеры состоят из энкодера и декодера, каждый из которых содержит несколько слоев внимания и полностью связанных слоев. Механизм самовнимания (self-attention) в энкодере позволяет модели учитывать отношения между всеми словами во входной последовательности, что значительно улучшает качество генерации текста.

Одним из недостатков трансформеров является их высокая вычислительная сложность, особенно для длинных последовательностей, что ограничивает их применение в реальных задачах с большими объемами данных.

Основы модели FNet

Модель FNet была предложена для снижения вычислительной сложности трансформеров. FNet заменяет механизм самовнимания на быстрое преобразование Фурье (FFT), что значительно уменьшает количество вычислений.

Основные идеи FNet включают:

1. **Быстрое преобразование Фурье (FFT)**: Используется вместо механизма самовнимания для уменьшения вычислительной сложности.
2. **Сохранение архитектуры трансформеров**: FNet сохраняет общую структуру трансформеров, включая энкодеры и декодеры, но заменяет механизм внимания на FFT.

Преимущества FNet включают:

- Снижение вычислительной сложности за счет использования FFT.
- Улучшение производительности при работе с длинными последовательностями.

Недостатки FNet:

- Возможная потеря качества генерации текста по сравнению с трансформерами, использующими механизм самовнимания.

Итог

Теоретические основы генерации текста включают множество подходов, от ранних статистических методов до современных нейронных сетей и трансформеров. Модель FNet представляет собой инновационное решение, которое снижает вычислительную сложность трансформеров, сохраняя при этом их основные преимущества. В следующей главе будет рассмотрена методология, используемая для реализации модели FNet на практике.

3. Методология

Выбор и подготовка данных

Для успешной реализации модели FNet необходима тщательная подготовка данных. Этот процесс включает в себя выбор соответствующего набора данных, предобработку текстов, их токенизацию и подготовку к обучению.

1. **Выбор данных**: В рамках данной работы используется набор данных Wikitext-2, который представляет собой коллекцию статей из Википедии. Этот набор данных содержит качественные текстовые данные, что делает его подходящим для задач генерации текста.
2. **Предобработка данных**: Предобработка включает несколько этапов:
    - Приведение текста к нижнему регистру для унификации данных.
    - Удаление нежелательных символов (например, цифр и специальных символов), оставляя только буквы и знаки препинания.
    - Удаление лишних пробелов и строк, которые не содержат полезной информации.
    - Фильтрация предложений, длина которых меньше определенного порога, чтобы исключить слишком короткие и, следовательно, малоинформативные тексты.

    Эта предобработка необходима для того, чтобы данные были однородными и пригодными для обучения модели.

3. **Токенизация данных**: Токенизация заключается в преобразовании текста в последовательности чисел, которые могут быть обработаны моделью. Для этого используется токенизатор, который разбивает текст на отдельные слова или подслова и преобразует их в числовые индексы. В данной работе используется токенизатор библиотеки Hugging Face, который адаптирован для работы с набором данных Wikitext-2.

Описание используемых библиотек и инструментов

Для реализации модели FNet используются следующие основные библиотеки и инструменты:

1. **PyTorch**: PyTorch - это популярная библиотека для глубокого обучения, которая предоставляет удобные и гибкие средства для создания и обучения нейронных сетей. В данной работе PyTorch используется для реализации архитектуры модели FNet, обучения модели и выполнения всех вычислений.
2. **Datasets**: Библиотека datasets предоставляет готовые наборы данных для различных задач обработки естественного языка. В нашем случае

используется набор данных Wikitext-2, который легко загружается и обрабатывается с помощью данной библиотеки.
3. **Transformers**: Библиотека transformers от Hugging Face предоставляет инструменты и модели для обработки естественного языка, включая токенизаторы и предварительно обученные модели. В данной работе используется токенизатор из этой библиотеки для преобразования текстовых данных в числовые последовательности.
4. **Numpy**: Эта библиотека используются для различных численных и научных вычислений, таких как создание и манипулирование массивами, выполнение линейной алгебры и другие операции.
5. **Torch.fft**: Модуль torch.fft используется для реализации быстрого преобразования Фурье (FFT), которое заменяет механизм внимания (attention) в архитектуре модели FNet. Это позволяет значительно сократить вычислительную сложность модели.

Алгоритм реализации модели FNet

Модель FNet представляет собой модифицированную архитектуру трансформеров, где механизм самовнимания заменен на быстрое преобразование Фурье (FFT). Это позволяет значительно уменьшить вычислительную сложность, сохраняя при этом высокое качество генерации текста.

Алгоритм реализации модели FNet включает следующие основные этапы:

1. **Позиционное кодирование**:
    - В отличие от стандартных трансформеров, FNet использует позиционное кодирование, основанное на синусоидах и косинусоидах для представления позиции каждого слова в последовательности.
2. **Энкодер FNet**:
    - Энкодер модели FNet состоит из нескольких слоев, каждый из которых включает в себя быстрое преобразование Фурье для обработки входных данных.
    - FFT применяется к входным данным, чтобы получить их представление в частотной области, что позволяет учитывать отношения между всеми словами в последовательности одновременно.
    - Результаты FFT комбинируются с исходными данными и проходят через плотный слой для дальнейшей обработки и нормализации.
3. **Декодер FNet**:
    - Декодер модели FNet также состоит из нескольких слоев, но дополнительно включает механизмы внимания для обработки выходных последовательностей.

- В отличие от энкодера, декодер использует механизм самовнимания для учета контекста как в целевой, так и в исходной последовательностях.
- После применения внимания и плотных слоев результаты объединяются и нормализуются.

4. **Обучение модели**:
   - Модель обучается на подготовленных и токенизированных данных с использованием функции потерь и оптимизатора.
   - Процесс обучения включает многократное прохождение данных через модель (эпохи), корректировку весов модели на основе вычисленной ошибки, и улучшение способности модели к генерации текстов.
5. **Генерация текста**:
   - После обучения модель способна генерировать тексты на основе входных данных. Процесс генерации заключается в последовательном предсказании слов и их добавлении к текущему тексту до достижения максимальной длины или появления специального токена завершения.

Таким образом, методология реализации модели FNet включает выбор и подготовку данных, использование специализированных библиотек и инструментов, а также разработку алгоритма, который позволяет эффективно обучать и применять модель для задач генерации текста.

4. Практическая реализация

Для успешного выполнения задач по генерации текста с использованием архитектуры FNet необходимо правильно подготовить данные и загрузить модель. В данной главе мы подробно рассмотрим этапы предобработки данных и загрузки модели, а также обеспечим корректное выполнение этих операций с использованием Python и библиотек PyTorch.

Импорт библиотек и подготовка окружения

Для начала работы нам необходимо установить и импортировать все необходимые библиотеки. В данном проекте мы будем использовать библиотеки `datasets` и `torch`.

```
!pip install datasets
!pip install torch[transformers]
```

Импортируем все необходимые модули и объявим переменную `device` для использования GPU, если это возможно.

```
import torch
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

Загрузка и предобработка данных

Для обучения и тестирования модели нам потребуется набор данных. В этом примере мы используем датасет `wikitext`, который доступен через библиотеку `datasets`.

```
from datasets import load_dataset
datasets = load_dataset('wikitext','wikitext-2-raw-v1')
```

Далее нам необходимо предобработать текст.

```
import re


def preprocess_text(sentence):
  text = sentence['text'].lower()
  text = re.sub('[^a-z?!.,]', ' ', text)
  text = re.sub('\s\s+', ' ', text)
  sentence['text'] = text
  return sentence


datasets['train'] = datasets['train'].map(preprocess_text)
datasets['test'] = datasets['test'].map(preprocess_text)
datasets['validation'] = datasets['validation'].map(preprocess_text)
```

```
datasets['train'] = datasets['train'].filter(lambda x: len(x['text']) >
20)
datasets['test'] = datasets['test'].filter(lambda x: len(x['text']) > 20)
datasets['validation'] = datasets['validation'].filter(
  lambda x: len(x['text']) > 20)
```

Теперь, когда датасет загружен и текст предобработан, необходимо выполнить токенизацию текста. Мы будем использовать токенизатор для преобразования текстовых данных в числовые.

```
from torch.utils.data import DataLoader
from transformers import DataCollatorWithPadding
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)


# Tokenizer
def tokenize(sentence):
  sentence = tokenizer(sentence['text'], truncation=True)
  return sentence


tokenized_inputs = datasets['test'].map(tokenize)
tokenized_inputs = tokenized_inputs.remove_columns(['text'])


# DataCollator
batch = 16
data_collator = DataCollatorWithPadding(
  tokenizer=tokenizer, padding=True, return_tensors="pt")
dataloader = DataLoader(
  tokenized_inputs, batch_size=batch, collate_fn=data_collator)
```

Позиционное кодирование

После токенизации данных нам необходимо создать два класса: для позиционного энкодинга и для эмбеддингов.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.fft as fft
import numpy as np
import pandas as pd

class PositionalEncoding(torch.nn.Module):
```

```
  def __init__(self, d_model, max_sequence_length):
    super().__init__()
    self.d_model = d_model
    self.max_sequence_length = max_sequence_length
    self.positional_encoding =
self.create_positional_encoding().to(device)

  def create_positional_encoding(self):

    positional_encoding = np.zeros((self.max_sequence_length,
self.d_model))

    for pos in range(self.max_sequence_length):
      for i in range(0, self.d_model, 2):
        positional_encoding[pos, i] = np.sin(pos / (10000 ** ((2 * i) /
self.d_model)))

        if i + 1 < self.d_model:
          positional_encoding[pos, i + 1] = np.cos(pos / (10000 ** ((2 *
i) / self.d_model)))

    return torch.from_numpy(positional_encoding).float()

  def forward(self, x):
    expanded_tensor = torch.unsqueeze(self.positional_encoding,
0).expand(x.size(0), -1, -1).to(device)

    return x.to(device) + expanded_tensor[:,:x.size(1), :]

class PositionalEmbedding(nn.Module):
  def __init__(self, sequence_length, vocab_size, embed_dim):
    super(PositionalEmbedding, self).__init__()
    self.token_embeddings = nn.Embedding(vocab_size, embed_dim)
    self.position_embeddings =
PositionalEncoding(embed_dim,sequence_length)

  def forward(self, inputs):
    embedded_tokens = self.token_embeddings(inputs).to(device)
    embedded_positions =
self.position_embeddings(embedded_tokens).to(device)
    return embedded_positions.to(device)
```

Построение и обучение модели

Теперь нам нужно реализовать саму архитектуру сети, начиная с энкодера:

```
class FNetEncoder(nn.Module):

  def __init__(self,embed_dim, dense_dim):
    super(FNetEncoder,self).__init__()
    self.embed_dim = embed_dim
```

```
    self.dense_dim = dense_dim
    self.dense_proj =
nn.Sequential(nn.Linear(self.embed_dim,self.dense_dim), nn.ReLU(),
nn.Linear(self.dense_dim,self.embed_dim))

    self.layernorm_1 = nn.LayerNorm(self.embed_dim)
    self.layernorm_2 = nn.LayerNorm(self.embed_dim)

  def forward(self,inputs):

    fft_result = fft.fft2(inputs)
    fft_real = fft_result.real.float()

    proj_input = self.layernorm_1 (inputs + fft_real)
    proj_output = self.dense_proj(proj_input)
    return self.layernorm_2(proj_input +proj_output)
```

Реализуем декодер:

```
class FNetDecoder(nn.Module):

  def __init__(self,embed_dim,dense_dim,num_heads):
    super(FNetDecoder,self).__init__()
    self.embed_dim = embed_dim
    self.dense_dim = dense_dim
    self.num_heads = num_heads

    self.attention_1 =
nn.MultiheadAttention(embed_dim,num_heads,batch_first=True)
    self.attention_2 =
nn.MultiheadAttention(embed_dim,num_heads,batch_first=True)

    self.dense_proj = nn.Sequential(nn.Linear(embed_dim,
dense_dim),nn.ReLU(),nn.Linear(dense_dim, embed_dim))

    self.layernorm_1 = nn.LayerNorm(embed_dim)
    self.layernorm_2 = nn.LayerNorm(embed_dim)
    self.layernorm_3 = nn.LayerNorm(embed_dim)

  def forward(self, inputs, encoder_outputs, mask=None):
    causal_mask =
nn.Transformer.generate_square_subsequent_mask(inputs.size(1)).to(device)

    attention_output_1, _ = self.attention_1(inputs, inputs, inputs,
attn_mask=causal_mask)
    out_1 = self.layernorm_1(inputs + attention_output_1)

    if mask != None:
      attention_output_2, _ = self.attention_2(out_1, encoder_outputs,
encoder_outputs, key_padding_mask =torch.transpose(mask, 0, 1).to(device))
    else:
```

```
      attention_output_2, _ = self.attention_2(out_1, encoder_outputs,
encoder_outputs)
      out_2 = self.layernorm_2(out_1 + attention_output_2)

    proj_output = self.dense_proj(out_2)
    return self.layernorm_3(out_2 + proj_output)
```

Используем ранее написанные нами энкодер и декодер для сборки самой модели:

```
class FNetModel(nn.Module):
  def __init__(self, max_length, vocab_size, embed_dim, latent_dim,
num_heads):
    super(FNetModel, self).__init__()

    self.encoder_inputs = PositionalEmbedding(max_length,vocab_size,
embed_dim)
    self.encoder1 = FNetEncoder(embed_dim, latent_dim)
    self.encoder2 = FNetEncoder(embed_dim, latent_dim)
    self.encoder3 = FNetEncoder(embed_dim, latent_dim)
    self.encoder4 = FNetEncoder(embed_dim, latent_dim)


    self.decoder_inputs = PositionalEmbedding(max_length,vocab_size,
embed_dim)
    self.decoder1 = FNetDecoder(embed_dim, latent_dim, num_heads)
    self.decoder2 = FNetDecoder(embed_dim, latent_dim, num_heads)
    self.decoder3 = FNetDecoder(embed_dim, latent_dim, num_heads)
    self.decoder4 = FNetDecoder(embed_dim, latent_dim, num_heads)


    self.dropout = nn.Dropout(0.5)
    self.dense = nn.Linear(embed_dim, vocab_size)

  def encoder(self,encoder_inputs):
    x_encoder = self.encoder_inputs(encoder_inputs)
    x_encoder = self.encoder1(x_encoder)
    x_encoder = self.encoder2(x_encoder)
    x_encoder = self.encoder3(x_encoder)
    x_encoder = self.encoder4(x_encoder)
    return x_encoder

  def decoder(self,decoder_inputs,encoder_output,att_mask):
    x_decoder = self.decoder_inputs(decoder_inputs)
    x_decoder = self.decoder1(x_decoder, encoder_output,att_mask)
    x_decoder = self.decoder2(x_decoder, encoder_output,att_mask)
    x_decoder = self.decoder3(x_decoder, encoder_output,att_mask)
    x_decoder = self.decoder4(x_decoder, encoder_output,att_mask)
    decoder_outputs = self.dense(x_decoder)

    return decoder_outputs
```

```
  def forward(self, encoder_inputs, decoder_inputs,att_mask = None):
    encoder_output = self.encoder(encoder_inputs)
    decoder_output =
self.decoder(decoder_inputs,encoder_output,att_mask=None)
    return decoder_output
```

Загрузка модели FNet

Следующим шагом является задание констант и создание инстанса модели FNet.

```
MAX_LENGTH = 512
VOCAB_SIZE = len(tokenizer.vocab)
EMBED_DIM = 256
LATENT_DIM = 100
NUM_HEADS = 4

fnet_model = FNetModel(MAX_LENGTH, VOCAB_SIZE, EMBED_DIM, LATENT_DIM,
NUM_HEADS).to(device)
```

Обучение модели

Теперь, когда данные подготовлены и модель загружена, мы можем приступить к обучению модели. Определим оптимизатор и функцию потерь, а затем выполним тренировку модели.

```
optimizer = torch.optim.Adam(fnet_model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index=0)

epochs = 100
for epoch in range(epochs):
  train_loss = 0
  for batch in dataloader:
    encoder_inputs_tensor = batch['input_ids'][:,:-1].to(device)
    decoder_inputs_tensor = batch['input_ids'][:,1:].to(device)

    att_mask = batch['attention_mask'][:,:-1].to(device).to(dtype=bool)
    optimizer.zero_grad()
    outputs = fnet_model(encoder_inputs_tensor,
decoder_inputs_tensor,att_mask)
    decoder_inputs_tensor.masked_fill(batch['attention_mask'][:,1:].ne(1).
to(device), -100).to(device)

    loss = criterion(outputs.view(-1, VOCAB_SIZE),
decoder_inputs_tensor.view(-1))
    train_loss = train_loss + loss.item()
    loss.backward()
    optimizer.step()
```

```
  print (f" epoch: {epoch}, train_loss : {train_loss}")
```

Импорт библиотек и подготовка окружения

Заключительным шагом является написание кода для непосредственного ее использования. Здесь мы прописываем предсказание токенов, токены начала и конца.

```
MAX_LENGTH = 100

def decode_sentence(input_sentence, fnet_model):
  fnet_model.eval()

  with torch.no_grad():
    tokenized_input_sentence =
torch.tensor(tokenizer(preprocess_text(input_sentence)['text'])['input_ids
']).to(device)#
    tokenzied_target_sentence = torch.tensor([101]).to(device) # '[CLS]'
token
    current_text = preprocess_text(input_sentence)['text']
    for i in range(MAX_LENGTH):
      predictions = fnet_model(tokenized_input_sentence[:-
1].unsqueeze(0),tokenzied_target_sentence.unsqueeze(0))
      predicted_index = torch.argmax(predictions[0, -1, :]).item()
      predicted_token = tokenizer.decode(predicted_index)
      if predicted_token == "[SEP]": # Assuming [end] is the end token
        break
      current_text += " "+ predicted_token
      tokenized_target_sentence = torch.cat([tokenzied_target_sentence,
torch.tensor([predicted_index]).to(device)], 0).to(device)
      tokenized_input_sentence =
torch.tensor(tokenizer(current_text)['input_ids']).to(device)
    return current_text
decode_sentence({'text': 'party'}, fnet_model)
```

5. Результаты и вывод

Анализ результатов генерации текста

Модель FNet была обучена на наборе данных Wikitext-2 и протестирована на генерации текста. Один из результатов генерации текста приведен ниже:

Пример генерации:

```
party goalkeeper italian oldham oldham oldham oldham
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol symbol symbol
symbol symbol symbol symbol symbol symbol
```

Анализ данного примера показывает, что модель имеет трудности с генерацией разнообразного и связного текста. Замечается повторение слов и фраз, что указывает на проблемы с обучением и возможное переобучение или недостаточную регуляризацию модели. Важно отметить, что такие результаты могут быть связаны с недостаточно оптимальными гиперпараметрами, недостаточным объемом данных для обучения или архитектурными особенностями модели FNet.

Сравнение с другими моделями

Для объективной оценки результатов генерации текста модели FNet, мы проведем сравнение с первыми моделями GPT и другими моделями той же эпохи, такими как LSTM и GRU, которые были популярны до широкого распространения трансформеров.

GPT (первого поколения):

- **Качество генерации текста**: GPT первой версии уже показала способность генерировать связные и контекстуально адекватные тексты благодаря использованию механизма внимания.

- **Проблемы**: Хотя GPT-1 лучше справляется с генерацией текста, чем FNet, она все же не лишена проблем с повторением и иногда теряет контекст при длинных последовательностях.

LSTM и GRU:

- **Качество генерации текста**: LSTM и GRU хорошо справляются с последовательными данными и способны генерировать более разнообразный текст по сравнению с классическими RNN. Однако, их качество генерации текста все еще уступает современным трансформерным моделям.
- **Проблемы**: Основные проблемы включают в себя забывание контекста на длинных последовательностях и сложности с обучением.

Преимущества и недостатки модели FNet

Преимущества FNet:

1. **Эффективность вычислений**: Замена механизма самовнимания на быстрое преобразование Фурье (FFT) значительно снижает вычислительные затраты, что делает FNet более эффективной по сравнению с традиционными трансформерами.
2. **Простота архитектуры**: Упрощенная архитектура позволяет легче понимать и реализовывать модель, а также потенциально снижает потребности в вычислительных ресурсах.

Недостатки FNet:

1. **Качество генерации текста**: Как показывает пример генерации, качество текста, сгенерированного FNet, значительно уступает современным моделям, что может быть связано с недостаточной способностью модели к учету контекста.
2. **Проблемы с повторением**: Модель склонна к повторению слов и фраз, что указывает на недостатки в обучении и архитектурных особенностях.
3. **Ограниченная гибкость**: Замена механизма самовнимания на FFT может ограничивать способность модели эффективно учитывать сложные зависимости в тексте, что негативно сказывается на качестве генерации.

Заключение

Модель FNet представляет собой интересную альтернативу традиционным трансформерам, предлагая преимущества в вычислительной эффективности и простоте архитектуры. Однако, результаты генерации текста показывают, что FNet на данный момент уступает в качестве более современным моделям,

таким как GPT-1 и другим моделям на основе механизма самовнимания. Для улучшения качества генерации текста требуется дальнейшая работа над архитектурой модели и гиперпараметрами, а также возможно привлечение более сложных методов регуляризации и предобработки данных.

Список литературы

[1] FNet: Mixing Tokens with Fourier Transforms [Электронный ресурс]: n.d. [сайт]. — сайтURL: https://arxiv.org/abs/2105.03824

[2] Adaptive Fourier Neural Operators: Efficient Token Mixers for Transformers [Электронный ресурс]: n.d. [PDF]. — URL: https://arxiv.org/abs/2111.13587

[3] Choose a Transformer: Fourier or Galerkin [Электронный ресурс]: n.d. [сайт]. —
URL: https://proceedings.neurips.cc/paper/2021/hash/d0921d442ee91b896ad95059d13df618-Abstract.html