Искусственный интеллект

2022/23 учебный год

Инженерный тур

Общая информация

Задача предполагает создание алгоритма, способного ответить на вопросы к видео, заданные на естественном (русском) языке.

Актуальность задачи

Решение подобных задач способствует развитию технологий автоматического анализа визуальных данных. Автоматический анализ визуальных данных в значительной степени сокращает объем анализа по сравнению с ручной обработкой и создает возможности для автоматизации решения таких задач, как: поиск нужного фрагмента видео по запросу (например, на камерах видеонаблюдения), поиск в базе данных различных видео (например, классификация видео с определенными участниками) и так далее.

Требования к команде и компетенциям участников

Максимальное число участников в команде — 2. При этом мы не выделяем ролей и предлагаем участникам быть равноправными партнерами.

В качестве рекомендации участникам предлагалось при объединении выбирать партнеров таким образом, чтобы вместе они могли на высоком уровне закрыть большинство необходимых компетенций.

Оборудование и программное обеспечение

Участникам были предоставлены ноутбуки: оперативная память — 8 Гб, хранилище SSD — 256 Гб, процессор — Intel Core i5.

Предустановленное ПО:

- Браузер Chrome, Версия 108.0.5359.95.
- PyCharm, community edition, Версия 2021.3.2.
- Jupyter Notebook, из дистрибутива Anaconda 3 2020.11.
- Python 3.10.
- GIT 2.34.1.
- MS Office 2016.

Дополнительно для обучения моделей участникам был предоставлен доступ к сервису ML Space, где они получали возможность обучения моделей на 1 GPU Nvidia A100 в течение 20 ч.

Описание задачи

В задаче Video Question Answering модели необходимо проанализировать короткие видеофрагменты и представленные к ним вопросы по содержанию и запечатленным событиям и/или действиям, чтобы сгенерировать наиболее подходящие ответы на русском языке.

Участникам был предоставлен обучающий набор данных, который включал в себя 2700+ коротких видео и соответствующих им вопросов и ответов на русском языке, а также базовое решение от разработчиков с докером и примером загружаемого решения.

Так как участники не имели доступа к проверяющему датасету, в качестве решения на платформу нужно было загрузить обученную модель. Решения на проверяющей платформе запускались в изолированном окружении при помощи Docker. Время и ресурсы во время тестирования ограничены.

Система оценивания

Оценка решений производилась автоматически на основании метрики BLEU, которая позволяет сравнить эталонный и предсказанный текст. При этом BLEU оценивает не только соответствие отдельных слов, но и ;-грамм, содержащихся в тексте. Метрика BLEU была изначально предложена для оценки качества машинного перевода, однако она может применяться в любых задачах, в которых необходимо оценить близость двух текстов (при этом допуская вариативность текстов-кандидатов, что важно в задаче описания видео).

Проверочный набор данных был разделен на 2 части: публичную и приватную. Результаты участников, сформированные на основании публичной части, были доступны в ходе всего соревнования. После завершения приема решений был сформирован итоговый рейтинг на основании приватной части проверочного набора данных.

После формирования итогового рейтинга участникам будут начислены баллы, рассчитанные по формуле:

$$Total = \frac{Result \times (RankParticipants + 1 - RankPlace) \times 100}{MaxResult \times Participants},$$

где $Total$ — итоговый балл за решение задачи;

$Result$ — результат решения задачи;

$RankParticipants$ — общее количество участников в рейтинге;

$RankPlace$ — место участника в рейтинге;

$MaxResult$ — максимальный результат решения задачи среди всех участников;

$Participants$ — общее количество участников.

Участникам этапа, не загрузившим решение в сроки проведения соревнования, было начислено 0 баллов.

Решение задачи

Рассмотрим модель ответа на вопрос по видео на основе архитектуры CLIP PREFIX CAPTION, включающую в себя CLIP и ruGPT2small.

Установка библиотек

Устанавливаем библиотеки, под которыми запускается данное решение.

```
!pip install wandb
!pip install transformers
!pip install git+https://github.com/openai/CLIP.git
!pip install nltk
```

Инициализируем модель

```
import torch
import torch.nn as nn
from torch.nn import functional as nnf
from torch.utils.data import Dataset, DataLoader
from transformers import GPT2Tokenizer, GPT2LMHeadModel, AdamW,
↪  get_linear_schedule_with_warmup
from tqdm import tqdm, trange
import os
import pickle
import sys
import argparse
import json
from typing import Tuple, Optional, Union
from torch.cuda.amp import autocast

import clip

class MLP(nn.Module):
    def __init__(self, sizes: Tuple[int, ...], bias=True, act=nn.Tanh):
        super(MLP, self).__init__()
        layers = []
        for i in range(len(sizes) - 1):
            layers.append(nn.Linear(sizes[i], sizes[i + 1], bias=bias))
            if i < len(sizes) - 2:
                layers.append(act())
        self.model = nn.Sequential(*layers)

    #@autocast()
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.model(x)

def freeze(
    model,
    freeze_emb=False,
    freeze_ln=False,
    freeze_attn=True,
    freeze_ff=True,
    freeze_other=False,
):
```

```
    for name, p in model.named_parameters():
    # freeze all parameters except the layernorm and positional embeddings
        name = name.lower()
        if 'ln' in name or 'norm' in name:
            p.requires_grad = not freeze_ln
        elif 'embeddings' in name:
            p.requires_grad = not freeze_emb
        elif 'mlp' in name:
            p.requires_grad = not freeze_ff
        elif 'attn' in name:
            p.requires_grad = not freeze_attn
        else:
            p.requires_grad = not freeze_other

    return model

class ClipCaptionModel(nn.Module):
    def __init__(self, prefix_length: int, prefix_size: int = 768):
          super(ClipCaptionModel, self).__init__()
          self.prefix_length = prefix_length
          """
          ru gpts fix, help sometimes

          """
          self.gpt =
          ↪  GPT2LMHeadModel.from_pretrained('sberbank-ai/rugpt3small_based_on_gpt2')

          self.gpt_embedding_size = self.gpt.transformer.wte.weight.shape[1]
          self.clip_project = MLP((prefix_size, (self.gpt_embedding_size *
          ↪  prefix_length) // 2,
                                   self.gpt_embedding_size * prefix_length))

    def get_dummy_token(self, batch_size: int, device: torch.device) -> torch.Tensor:
        return torch.zeros(batch_size, self.prefix_length, dtype=torch.int64,
        ↪  device=device)

    #@autocast()
    def forward(self, tokens: torch.Tensor, prefix: torch.Tensor, mask:
    ↪  Optional[torch.Tensor] = None,
                labels: Optional[torch.Tensor] = None):

        embedding_text = self.gpt.transformer.wte(tokens)
        prefix_projections = self.clip_project(prefix).view(-1, self.prefix_length,
        ↪  self.gpt_embedding_size)

        embedding_cat = torch.cat((prefix_projections, embedding_text), dim=1)
        if labels is not None:
            dummy_token = self.get_dummy_token(tokens.shape[0], tokens.device)
            labels = torch.cat((dummy_token, tokens), dim=1)
        out = self.gpt(inputs_embeds=embedding_cat, labels=labels,
        ↪  attention_mask=mask)
        return out

class ClipCaptionPrefix(ClipCaptionModel):

    def parameters(self, recurse: bool = True):
        return self.clip_project.parameters()

    def train(self, mode: bool = True):
        super(ClipCaptionPrefix, self).train(mode)
```

```
        self.gpt.eval()
        return self
```

Инициализируем модель CLIP и токенайзер для текста

```
device = 'cuda:0'
clip_model, preprocess = clip.load("ViT-L/14@336px", device=device, jit=False)
tokenizer = GPT2Tokenizer.from_pretrained('sberbank-ai/rugpt3small_based_on_gpt2')
```

Инициализируем код предобработки данных, обработанные данные сохраняться в `.pkl`, что позволит не выполнять повторно дорогостоящую операцию преобразования.

```
import io
import os
import PIL
import random
import numpy as np
import torch
import torchvision
import transformers
import more_itertools
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import pandas as pd
from torch.utils.data import Dataset
from tqdm import tqdm
from dataclasses import dataclass, field
import torchvision.transforms as T
import torchvision.transforms.functional as TF
import cv2
from PIL import Image
import pickle
from tqdm.contrib import tzip
def image_grid(imgs, rows, cols):
    pils = imgs

    assert len(imgs) == rows*cols

    w, h = imgs[0].size
    grid = Image.new('RGB', size=(cols*w, rows*h))
    grid_w, grid_h = grid.size

    for i, img in enumerate(imgs):
        grid.paste(img, box=(i%cols*w, i//cols*h))
    return grid

def read_video(path, transform=None, frames_num=16, window=30):
    frames = []
    cap = cv2.VideoCapture(path)

    fps = int(cap.get(cv2.CAP_PROP_FPS))

    length = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    N = length//(frames_num)

    current_frame = 0
```

```
    for i in range(length):
        ret, frame = cap.read(current_frame)

        if ret and i==current_frame and len(frames)<frames_num:
            size = 196, 196
            frame = Image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
            frame.thumbnail(size, Image.ANTIALIAS)

            frames.append(frame)
            current_frame += N

    cap.release()
    return frames

out_path = f"Features_train.pkl"
video_path =  'videos'
import pandas as pd
df = pd.read_csv('ru_train_full.csv')

all_embeddings = []
all_captions = []
i = 0
for video_name, question, answer in tzip(df.video_name, df.question, df.answer):
    name = f'{video_path}/{video_name}.mp4'
    text = f'Вопрос: {question} ? Ответ: {answer}.'
    # обратите внимание, что с этой частью стоит поэкспрементировать
    if os.path.exists(name):
        video = read_video(path = name, frames_num=4)
        if len(video)>1:
            image = image_grid(video,2,2)# можно попробовать меньше/больше кадров

            image = preprocess(image).unsqueeze(0).to(device)
            with torch.no_grad():
                prefix = clip_model.encode_image(image).cpu()
            all_embeddings.append(prefix)
            all_captions.append(text)

with open(out_path, 'wb') as f:
    pickle.dump({"clip_embedding": torch.cat(all_embeddings, dim=0), "captions":
    ↪  all_captions}, f)

print('Done')
print("%0d embeddings saved " % len(all_embeddings))
```

Опишем класс датасета, он наследуется из `torch.dataset`.

```
class ClipCocoDataset(Dataset):
    def __init__(self, data_path: str,  prefix_length= 50, gpt2_type =
    ↪  'sberbank-ai/rugpt3small_based_on_gpt2',
                 normalize_prefix=False):
        self.tokenizer = GPT2Tokenizer.from_pretrained(gpt2_type)
        self.prefix_length = prefix_length
        self.normalize_prefix = normalize_prefix
        with open(data_path, 'rb') as f:
            all_data = pickle.load(f)
        print("Data size is %0d" % len(all_data["clip_embedding"]))
        sys.stdout.flush()
        self.prefixes = all_data["clip_embedding"]
        captions_raw = all_data["captions"]
```

```
        self.captions = captions_raw

        self.captions_tokens = []
        self.caption2embedding = []
        max_seq_len = 0
        i=0
        for caption in tqdm(captions_raw):
            self.captions_tokens.append(torch.tensor(self.tokenizer.encode(caption),
            ↪  dtype=torch.int64))
            self.caption2embedding.append(self.prefixes[i])
            i+=1
            max_seq_len = max(max_seq_len, self.captions_tokens[-1].shape[0])

        all_len = torch.tensor([len(self.captions_tokens[i]) for i in
        ↪  range(len(self))]).float()
        self.max_seq_len = min(int(all_len.mean() + all_len.std() * 10),
        ↪  int(all_len.max()))

    def pad_tokens(self, item: int):
        tokens = self.captions_tokens[item]
        padding = self.max_seq_len - tokens.shape[0]
        if padding > 0:
            tokens = torch.cat((tokens, torch.zeros(padding, dtype=torch.int64) - 1))
            self.captions_tokens[item] = tokens
        elif padding < 0:
            tokens = tokens[:self.max_seq_len]
            self.captions_tokens[item] = tokens
        mask = tokens.ge(0)  # mask is zero where we out of sequence
        tokens[~mask] = 0
        mask = mask.float()
        mask = torch.cat((torch.ones(self.prefix_length), mask), dim=0)  # adding
        ↪  prefix mask
        return tokens, mask

    def __len__(self) -> int:
        return len(self.captions_tokens)

    def __getitem__(self, item):
        tokens, mask = self.pad_tokens(item)
        prefix = self.prefixes[item]
        if self.normalize_prefix:
            prefix = prefix.float()
            prefix = prefix / prefix.norm(2, -1)
        return tokens, mask, prefix
```

Опишем функцию обучения

```
import torch
import torch.nn as nn
from torch.nn import functional as nnf
from torch.utils.data import Dataset, DataLoader
from transformers import GPT2Tokenizer, GPT2LMHeadModel, AdamW,
↪  get_linear_schedule_with_warmup
from tqdm import tqdm
import os
import pickle
import sys
import argparse
import json
```

```python
from typing import Tuple, Optional, Union
from torch.cuda.amp import autocast

# Импортируем необходимые библиотеки
import wandb
wandb.login()
wandb.init(project="clip_caption_video", entity="alexwortega")

# Начинаем определять функцию обучения модели
def train(dataset: ClipCocoDataset, model: ClipCaptionModel, args,
          warmup_steps: int = 2000, output_dir: str = ".", output_prefix: str = ""):

    device =torch.device('cuda:0') # Определяем устройство, на котором будем обучать
    ↪  модель

    batch_size = args.bs # Задаем размер пакета
    epochs = args.epochs # Задаем количество эпох

    if not os.path.exists(output_dir):
        os.makedirs(output_dir) # Создаем папку для сохранения модели

    model = freeze(model) # Замораживаем слои модели
    model = model.to(device) # Переносим модель на устройство

    model.train() # Определяем, что модель находится в режиме обучения
    optimizer = AdamW(model.parameters(), lr=args.lr) # Используем оптимизатор AdamW
    ↪  для обучения модели

    train_dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
    ↪  drop_last=True) # Загружаем данные в DataLoader

    scheduler = get_linear_schedule_with_warmup(
        optimizer, num_warmup_steps=warmup_steps, num_training_steps=epochs *
        ↪  len(train_dataloader)
    ) # Создаем расписание скорости обучения с прогревом

    for epoch in range(epochs):
        print(f">>> Training epoch {epoch}")
        sys.stdout.flush()
        progress = tqdm(total=len(train_dataloader), desc=output_prefix) # Отображаем
        ↪  полосу прогресса
        step=0
        for idx, (tokens, mask, prefix) in enumerate(train_dataloader):
            model.zero_grad() # Обнуляем градиенты
            step+=1

            tokens, mask, prefix = tokens.to(device), mask.to(device),
            ↪  prefix.to(device, dtype=torch.float32) # Переносим данные на
            ↪  устройство

            outputs = model(tokens, prefix, mask) # Получаем выход модели

            logits = outputs.logits[:, dataset.prefix_length - 1: -1] # Отбрасываем
            ↪  лишние данные

            loss = nnf.cross_entropy(logits.reshape(-1, logits.shape[-1]),
            ↪  tokens.flatten(), ignore_index=0) # Вычисляем функцию потерь

            loss.backward() # Распространяем градиенты обратно по модели
            optimizer.step() # Обновляем параметры модели
```

```
            scheduler.step()
            optimizer.zero_grad() # Обнуляем градиенты оптимизатора
            progress.set_postfix({"loss": loss.item()}) # Отображаем значение функции
            ↪  потерь в полосе прогресса

            wandb.log({"loss":  loss.item()}) # Логируем значение функции потерь в
            ↪  WandB
            progress.update() # Обновляем полосу прогресса

            if (idx + 1) % 7000 == 0:
                torch.save(
                    model.state_dict(),
                    os.path.join(output_dir, f"{output_prefix}_latest.pt"),
                ) # Сохраняем модель

        progress.close() # Закрываем полосу прогресса
        if epoch % args.save_every ==0:
            torch.save(
                model.state_dict(),
                os.path.join(output_dir, f"{output_prefix}-{epoch:03d}.pt"),
            ) # Сохраняем модель на каждой эпохе, заданной в аргументах

    return model # Возвращаем обученную модель
# Определяем класс для хранения аргументов командной строки
class Args():
    def __init__(self):
        self.backbone = 'sberbank-ai/rugpt3small_based_on_gpt2'
        self.data = 'Features_train.pkl'
        self.out_dir = 'checkpoints'
        self.prefix = 'prefix_small'
        self.epochs = 2
        self.save_every = 1
        self.prefix_length = 50
        self.bs = 1
        self.only_prefix = False
        self.lr = 2e-5

# Определяем функцию main для запуска обучения модели
def main():
    args = Args() # Создаем объект класса Args и задаем значения аргументов

    wandb.config = {
      "learning_rate": args.lr,
      "epochs": args.epochs,
      "batch_size": args.bs
    } # Логируем аргументы в WandB

    prefix_length = args.prefix_length # Задаем длину префикса

    dataset = ClipCocoDataset(args.data, prefix_length) # Создаем датасет

    model = ClipCaptionModel(prefix_length) # Создаем модель

    print("Train both prefix and GPT")
    sys.stdout.flush()

    train(dataset, model, args, output_dir=args.out_dir, output_prefix=args.prefix) #
    ↪  Обучаем модель
```

Материалы для подготовки

В ходе 1-го этапа для участников был проведен онлайн-интенсив, который позволял в короткие сроки погрузиться в тему искусственного интеллекта и развить необходимые базовые навыки для участия в соревнованиях. Отдельно выделялись темы, связанные с компьютерным зрением и обработкой естественного языка. Материалы интенсива были доступны участникам на протяжении всего соревнования по ссылке: https://stepik.org/course/124175.