project2_part_b

November 28, 2023

[8]:
```
import numpy as np
import torch
import json
import random
from transformers import AutoTokenizer, GPT2Tokenizer, GPT2LMHeadModel
from mingpt.model import GPT
from mingpt.trainer import Trainer
from datasets import load_dataset
from scipy.stats import norm
import os
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from datasets import load_dataset
```

[ ]:
```
data = []
with open('proj2_data.jsonl', 'r') as jsonl_file:
    for line in jsonl_file:
        data.append(json.loads(line))
```

[18]:
```
class CustomThePileDataset(Dataset):
    def __init__(self, dataset, max_length=1024, do_UL2=False):
        self.data = dataset
        self.tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
        self.sentinel = [f'token_added{i}' for i in range(500)]
        self.tokenizer.add_tokens(self.sentinel)
        self.tokenizer.add_tokens(['[NLU]', '[S2S]','[NLG]'])
        self.tokenizer.pad_token_id = self.tokenizer.vocab_size-1
        self.vocab_size = len(self.tokenizer)
        self.denoisers = {
            'Regular (r)': ['[NLU]', self.Regular_denoising],
            'Sequential (s)': ['[S2S]', self.Sequential_denoising],
            'Extreme (x)': ['[NLG]', self.Extreme_denoising]
        }
        self.max_length = max_length
        self.do_UL2 = do_UL2

    def __len__(self):
        return len(self.data)
```

```
  def __getitem__(self, idx):
      text = self.data[idx]['text']

      if self.do_UL2 is True:
          random_choice = np.random.choice(['Regular (r)', 'Sequential (s)',␣
↪'Extreme (x)'], size=1)[0]
          denoiser_id, denoiser = self.denoisers[random_choice]
          text = denoiser_id  + text
          inputs = self.tokenizer.encode(text, truncation=True,␣
↪max_length=self.max_length, return_tensors='pt', add_special_tokens=True,␣
↪padding=True)
          return denoiser(inputs, self.tokenizer)

      elif self.do_UL2 is not True:
          # Tokenize the text
          inputs = self.tokenizer(text, return_tensors="pt", truncation=True,␣
↪max_length=self.max_length, padding='max_length')

          # Split tokens into chunks
          input_ids = inputs.input_ids[:,:-1]
          input_chunks = torch.split(input_ids, self.max_length)

          # Generate targets
          target_chunks = []
          for chunk in input_chunks:
              target_chunk = chunk[:,1:].clone()
              target_chunks.append(target_chunk)
          result_targets = inputs.input_ids[:,1:]

          return input_ids, result_targets

  def Regular_denoising(self, inputs, tokenizer, corruption=0.15):
      spans = [2, 3, 4, 5]
      tokens = None
      skip = 0
      used = 0

      for i in range(1, inputs.shape[1]):
          if skip > 0:
              skip -= 1
              continue

          if np.random.random() < corruption:
              mask = tokenizer.convert_tokens_to_ids(self.sentinel[used])
              used += 1
              span = np.random.choice(spans)
```

```
                if tokens is None:
                    tokens = [torch.tensor([[mask]]), inputs[:, i:i + span]]
                    new_inputs = [inputs[:, :i], torch.tensor([[mask]]),␣
↪inputs[:, i + span:]]
                    skip = span
                else:
                    tokens.extend([torch.tensor([[mask]]), inputs[:, i:i +␣
↪span]])
                    new_inputs.extend([torch.tensor([[mask]]), inputs[:, i +␣
↪span:]])
                    skip = span

        # Adjust inputs to max_length by padding with eos_token_id
        padding_length_inputs = self.max_length - new_inputs.shape[1]
        eos_padding_inputs = torch.full((1, padding_length_inputs), tokenizer.
↪eos_token_id)
        inputs = torch.cat((new_inputs, eos_padding_inputs), dim=1)

        # Calculate the total length for old_toks padding
        padding_length_tokens = self.max_length - sum([x.shape[1] for x in␣
↪tokens])
        eos_padding_tokens = torch.full((1, padding_length_tokens), tokenizer.
↪eos_token_id)

        # Concatenate all tensors in old_toks along with eos_padding_old_toks
        tokens_concatenated = torch.cat(tokens, dim=1)
        tokens = torch.cat((tokens_concatenated, eos_padding_tokens), dim=1)

        return inputs, tokens

    def Sequential_denoising(self, inputs, tokenizer):
        mask = tokenizer.convert_tokens_to_ids(self.sentinel[0])
        tokens = torch.cat((torch.tensor([[mask]]), inputs[:, :]].clone()),␣
↪dim=1)
        inputs[:, -1] = mask

        # Adjust inputs to max_length by padding with eos_token_id
        padding_length_inputs = self.max_length - inputs.shape[1]
        eos_padding_inputs = torch.full((1, padding_length_inputs), tokenizer.
↪eos_token_id)
        inputs = torch.cat((inputs, eos_padding_inputs), dim=1)

        # Calculate the total length for old_toks padding
        padding_length_tokens = self.max_length - sum([x.shape[1] for x in␣
↪tokens])
```

```
        eos_padding_tokens = torch.full((1, padding_length_tokens), tokenizer.
 ↪eos_token_id)

        # Concatenate all tensors in old_toks along with eos_padding_old_toks
        tokens_concatenated = torch.cat(tokens, dim=1)
        tokens = torch.cat((tokens_concatenated, eos_padding_tokens), dim=1)

        return inputs, tokens


    def Extreme_denoising(self, inputs, tokenizer, corruption=0.50):
        return self.Regular_denoising(inputs, tokenizer, corruption)


# Create an instance of the custom dataset
thepile_dataset = CustomThePileDataset(dataset=data)
thepile_dataset_UL2 = CustomThePileDataset(dataset=data, do_UL2=True)
```

[10]:
```
thepile_dataset_UL2[1]
```

[10]:
```
(tensor([[50757,    59,  5458,  …, 50256, 50256, 50256]]),
 tensor([[50257,  7531,  9391,  …, 50256, 50256, 50256]]))
```

[11]:
```
checkpoint_dir = 'thepile'
dir_path = f'./checkpoints/{checkpoint_dir}'

if not os.path.exists(dir_path):
    os.makedirs(dir_path)
    checkpoints = os.listdir(dir_path)
else:
    checkpoints = os.listdir(dir_path)

checkpoints.sort()
```

[12]:
```
model_config = GPT.get_default_config()
model_config.model_type = 'gpt-nano'
model_config.ul2 = True
model_config.vocab_size = thepile_dataset_UL2.vocab_size
model_config.block_size = thepile_dataset_UL2.max_length
model_config.checkpoint = None
model_UL2 = GPT(model_config)
```

```
number of parameters: 2.57M
```

[13]:
```
model_config = GPT.get_default_config()
model_config.model_type = 'gpt-nano'
model_config.ul2 = False
model_config.vocab_size = thepile_dataset.vocab_size
```

```
model_config.block_size = thepile_dataset.max_length
model_config.checkpoint = None
model = GPT(model_config)
```

```
number of parameters: 2.57M
```

[14]:
```
train_config = Trainer.get_default_config()
train_config.learning_rate = 5e-4
train_config.max_iters = 400 + model.iter_num if model_config.checkpoint else␣
 ↪400
train_config.num_workers = 0
train_config.checkpoint_iters = 100
train_config.batch_size = 1
train_config.checkpoint_name = f'{checkpoint_dir}/checkpoint_UL2'
trainer_UL2 = Trainer(train_config, model_UL2, thepile_dataset_UL2)
```

```
running on device cpu
```

[15]:
```
train_config = Trainer.get_default_config()
train_config.learning_rate = 5e-4
train_config.max_iters = 400 + model.iter_num if model_config.checkpoint else␣
 ↪400
train_config.num_workers = 0
train_config.checkpoint_iters = 100
train_config.batch_size = 1
train_config.checkpoint_name = f'{checkpoint_dir}/checkpoint'
trainer = Trainer(train_config, model, thepile_dataset)
```

```
running on device cpu
```

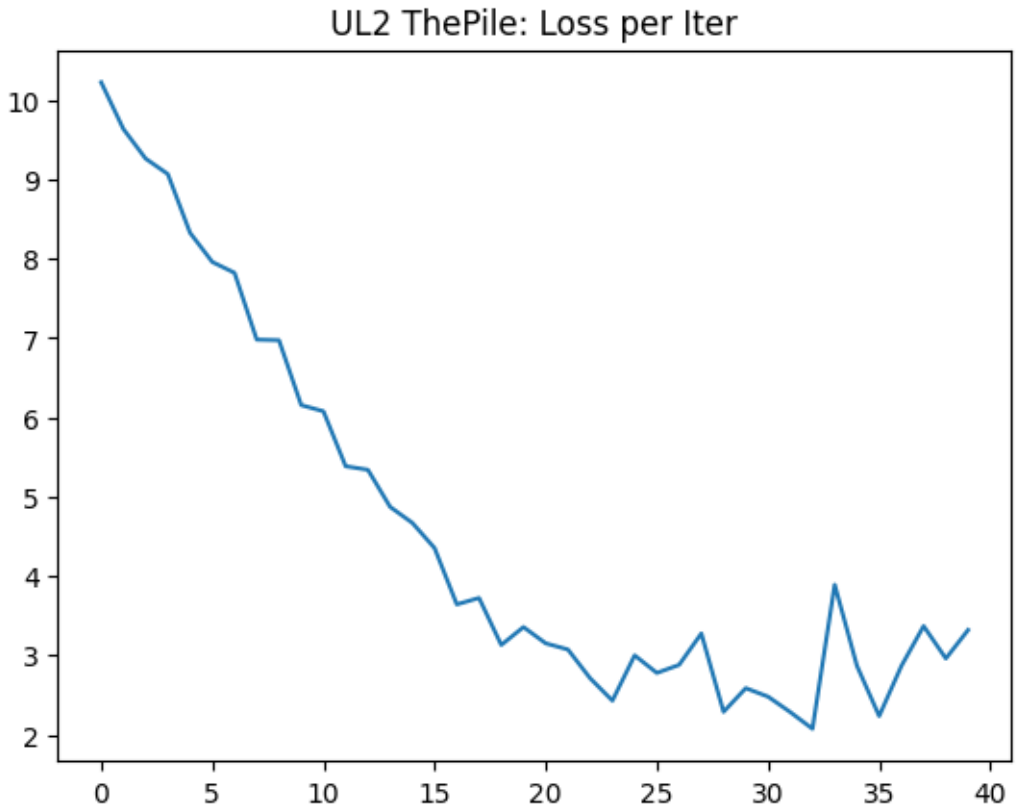
[16]:
```
def batch_end_callback(trainer):
    if trainer_UL2.iter_num % 100 == 0:
        print(f"iter_dt {trainer_UL2.iter_dt * 1000:.2f}ms; iter {trainer_UL2.
 ↪iter_num}: train loss {trainer_UL2.loss.item():.5f}")
trainer_UL2.set_callback('on_batch_end', batch_end_callback)

trainer_UL2.run()

# Plot the loss
losses = trainer_UL2.curr_loss
x = 10
new_losses = np.mean(np.array(losses).reshape(-1, x), axis=1)
plt.plot(np.arange(len(np.mean(np.array(losses).reshape(-1, x), axis=1))), np.
 ↪mean(np.array(losses).reshape(-1, x), axis=1))
plt.title('UL2 ThePile: Loss per Iter')
plt.show()
```

```
iter_dt 0.00ms; iter 0: train loss 10.86568
iter_dt 1259.37ms; iter 100: train loss 6.03233
```

```
iter_dt 1011.66ms; iter 200: train loss 3.72412
iter_dt 902.85ms; iter 300: train loss 0.26408
```



[ ]: