# CS480/680, Spring 2023, Assignment 3

## Designer: Chengjie Huang; Instructor: Hongyang Zhang

Released: June 26; Due: July 16, noon

Notes:

- Please save a copy of this notebook to avoid losing your changes.
- Debug your code and ensure that it can run.
- Save the output of each cell. Failure to do so may result in your coding questions not being graded.
- To accelerate the training time, you can use Google Colab and choose 'Runtime' -> 'Change runtime type' -> 'Hardware accelerator' and set 'Hardware accelerator' to 'GPU'. This assignment may be more time-consuming, so you may need to plan ahead and leave enough time for model training.

In [ ]:
```
# In addition to numpy, pytorch, and other standard libraries, you will need the following for this assignment
!pip install transformers datasets
```

## Question 1: Large Language Model (40 pts)

Large pre-trained language models such as GPT can be useful for many natural language tasks other than text generation. In this question, we will take a look at one of such tasks: question-answering (QA).

In QA task, the model is given some **context** text and a **question** related to the context. The model is tasked to generate the correct answer based on the context and question. For example, a context could be "Joe enjoys pizza but prefers pasta over anything else", and given a question "What's Joe's favorite food", the model should output "pasta".

In this question, we will extend and fine-tune a pre-trained large language model (GPT2) to perform question-answering task.

### 1.1 SQuAD Dataset (5 pts)

A popular dataset for question-answering task is the Stanford Question-answering Dataset (SQuAD) (Rajpurkar, Pranav, et al. "Squad: 100,000+ questions for machine comprehension of text." arXiv preprint arXiv:1606.05250 (2016).). The code below will automatically download and load the dataset. The training and validation split can be accessed with `squad_dataset['train']` and `squad_dataset['validation']` respectively.

First familiarize yourself with the format of the SQuAD dataset.
In the following cells, print the size of each split as well as one example from each split in the following format:

```
Train/validation split: 10000 samples
Sample id: 56de57394396321400ea2830
Context: Joe enjoys pizza but prefers pasta over anything else
Question: What's Joe's favorite food
Answer: pasta
```

In [ ]:
```
from datasets import load_dataset
squad_dataset = load_dataset("squad")

################################################################################
# IMPLEMENT ME!
raise NotImplementedError
################################################################################
```

### 1.2 Extending GPT2 for question-answering task (25 pts)

In this part, we will extend the GPT2 model to produce answers from the context based on the questions. To make use of the pre-trained GPT2 model, we will treat it as a feature extractor to compute token-wise feature vectors and add additional MLP layers to process the features for the QA task. These additional task-specific layers are sometimes called **"adapters"**.

Use the skeleton code below and implement the following three core components:

1. **[5 pts] Add additional MLP layer(s) to predict the location of the answer within the input.**
   You can formulate this problem however you see fit. Below are two possible options:

- Classify the start and end locations of the answer. For example, given a input text of length 5 after tokenization, suppose the answer starts at token `2` and ends at token `4`, the model should predict `[0 0 1 0 0]` and `[0 0 0 0 1]` as the start and end location respectively.
- Directly regress the start and end locations, or the start location + length of the answer.

2. **[10 pts] Use the additional layer(s) to compute the location of the answer**
3. **[10 pts] Implement task-specific loss for question-answering**
   Depending on the approach you choose to implement (classification vs. regression), you need to use the appropriate loss.

Unlike image data, text input can have varying length, which makes batch training and loss computation more challenging. For simplicity, you can assume the batch size is 1 in this question. I.e., the `question`, `context` and `answer` belong to a single sample in the dataset.

In [ ]:
```
import torch
import torch.nn as nn
from transformers import GPT2TokenizerFast, GPT2Model

class GPT2QuestionAnswering(nn.Module):
    def __init__(self):
        super().__init__()
        self.tokenizer = GPT2TokenizerFast.from_pretrained('gpt2')
        self.gpt2 = GPT2Model.from_pretrained('gpt2')

        ############################################################################################
        # IMPLEMENT ME!
        # Add additional layers for predicting the location of the answer
        raise NotImplementedError
        ############################################################################################

    def forward(self, question, context, answer=None):
        inputs = self.tokenizer(question, context, return_tensors='pt', return_offsets_mapping=True)
        input_ids = inputs.input_ids[:,:self.gpt2.config.n_positions]
        attention_mask = inputs.attention_mask[:,:self.gpt2.config.n_positions]
        if torch.cuda.is_available:
            input_ids = input_ids.cuda()
            attention_mask = attention_mask.cuda()
        features = self.gpt2(input_ids=input_ids, attention_mask=attention_mask)['last_hidden_state']

        ############################################################################################
        # IMPLEMENT ME!
        # Using the additional layers to compute location of the answer based on the hidden state features
        raise NotImplementedError
        ############################################################################################

        if self.training:
            ########################################################################################
            # IMPLEMENT ME!
            # In training mode, we want to return the loss based on the ground truth answer
            # You may change the
            return self.loss(...)
            ########################################################################################
        else:
            ########################################################################################
            # IMPLEMENT ME!
            # In inference mode, we want to return the answer string based on the predicted start and end indices
            answer_start_index = None
            answer_end_index = None
            ########################################################################################
            return self.tokenizer.decode(inputs.input_ids[0, answer_start_index : answer_end_index + 1]).strip()


    ################################################################################################
    # IMPLEMENT ME!
    def loss(self, ...):
        # Compute the loss based on the answers
        # Tip1: If you choose the classification approach, then you need the start and end locations
        #       of the answer within the **tokenized** input as your classification target.
        #       You may need to use inputs.offset_mapping from the input tokenization.
        #       See https://huggingface.co/docs/transformers/main_classes/tokenizer#transformers.PreTrainedTokenizerFast.__call__.r
        # Tip2: If you choose the regression approach, then you can use answer['answer_start']
        #       and length of the answer as your regression target.
        total_loss = None
        return total_loss
    ################################################################################################
```

We can evaluate the pre-trained model's performance on the validation split. Since the model has not been adapted to the question-answering task yet, and additional untrained layers have been added, we expect the model to perform poorly.

In question-answering task, we use two evaluation metrics:

- **Exact match**: the percentage of predictions that match the ground truth answer exactly
- **F1 score**: the average overlap (in terms of tokens) between the prediction and ground truth answer

Higher values are better for both metrics. For reference, humans can achieve 77.0% exact match and 86.8% F1 score, while SOTA method achieves 90% exact match and over 95% F1 score.

In [ ]:
```
def evaluate(model, dataset, metric):
    from tqdm.autonotebook import tqdm
    model = model.eval()
    preds = []
    for idx, data in enumerate(tqdm(dataset['validation'])):
        preds.append(dict(id=data['id'], prediction_text=model(data['question'], data['context'], data['answers'])))
    references = [dict(answers=data['answers'], id=data['id']) for data in dataset['validation']]
    return metric.compute(predictions=preds, references=references)
```

In [ ]:
```
from datasets import load_metric

squad_metric = load_metric('squad')
model = GPT2QuestionAnswering()
if torch.cuda.is_available:
    model = model.cuda()

evaluate(model, squad_dataset, squad_metric)
```

## 1.3 Fine-tuning GPT2 on Squad (10 pts)

To adapt a pre-trained model to a specific downstream task (in this case the question-answering task), a common technique is to **"fine-tune"** the model. Fine-tuning simply means training the model using task-specific data, typically with shorter epochs and smaller learning rates. Certain part of the model (e.g., pre-trained layers or early layers) can also be frozen, meaning the weights are not updated during training.

In this part, we will fine-tune the model on SQuAD dataset.
In the cell below, implement the training loop and record the loss values in a list to be plotted later.

**Note:** due to the size of the model and dataset, it is not required to train for too many iterations since this question will not be graded purely based on the performance of your model.

Optional Objectives

- **Freezing GPT2 Layers:** Depending on the additional layers you added to the network in the previous part, you may choose to freeze the pre-trained GPT2 layers during fine-tuning. In PyTorch, this can be achieved by setting `requires_grad=False` for the layers of interest. You are encouraged to try both and note your observations.
- **Gradient Accumulation:** The model from previous question is not suitable for batch training, which could increase the stochasiticy of the training process and make convergence slower. One way to circumvent this problem is via gradient accumulation, wherein the gradient is accumulated for multiple iterations before the weights are updated, which increases the effective batch size. In PyTorch, this can be implemented by accumulating the loss and performing `zero_grad()` and `step()` every few iterations instead of every iteration.

In [ ]:
```
from tqdm.autonotebook import tqdm
losses = []

################################################################################
# IMPLEMENT ME!
num_epochs = ...
optimizer = ...
################################################################################

model = model.train()
for e in range(num_epochs):
    # Shuffle dataset in each epoch
    indices = ...

    # tqdm gives you a nice little progress bar
    pbar = tqdm(indices)
    for i, idx in pbar:
        # Obtain data from training split
        data = squad_dataset['train'][idx]

        ################################################################################
        # IMPLEMENT ME!
        loss = ...
        ################################################################################

        # Record loss values
```

```
        losses.append(loss.item())
        pbar.set_postfix(loss=loss.item())
```

Plot the loss values over the iterations.

In [ ]:
```
import matplotlib.pyplot as plt
plt.figure(dpi=90)
plt.plot(losses)
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```

Evaluate the model performance after fine-tuning. You should see a much higher score compared to the previous results.

**Note:** this question will not be graded based on the performance of your model. However, together with the code and the loss plot in the previous parts, the performance will be used to judge if the fine-tuning is implemented properly.

In [ ]:
```
evaluate(model, squad_dataset, squad_metric)
```

Try the model using your own text and question. Can it give you the correct answer?

In [ ]:
```
model('Your question', 'Your context')
```

# Question 2: GAN (60 pts)

In this question, you will be designing and training a GAN for image generation.

Training a GAN for image generation can be computationally demanding. Luckily, MNIST dataset provides 28x28 images of handwritten digits, allowing a GAN to be trained more quickly. Below are some examples from the digits dataset:

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from torchvision.datasets import MNIST

mnist = MNIST(root='.', download=True)
images = np.stack([data[0] for data in mnist]).astype(np.float32)
images = images / 128 - 1    # normalize between -1 and 1
plt.figure(dpi=90)
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(images[np.random.choice(len(images))])
    plt.axis('off')
plt.show()
```

**Note:** You are allowed to use other **publicly available** image datasets for this question. If you choose to do so, please include the link to the dataset you use, and replace the cell above to visualize examples from your dataset.

## 2.1 Generator and Discriminator (20 pts)

In this part, you need to implement a generator and discriminator model using the skeleton code below. Recall that

- The **generator** takes a randomly sampled noise $z$ as input and outputs an image with the same size as the dataset
- The **discriminator** takes an image as input and performs a binary classification

In this case, both the generator and discriminator should be convolutional neural networks (CNNs). You may not use a pretrained network, but other design decisions such as the depth and width of the network are up to you.

Depending on the resources available to you, you may choose to implement a small network.

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        ################################################################################
        # IMPLEMENT ME!
        raise NotImplementedError
        ################################################################################
```

```
    def forward(self, x):
        ##########################################################################################
        # IMPLEMENT ME!
        raise NotImplementedError
        ##########################################################################################

class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        ##########################################################################################
        # IMPLEMENT ME!
        raise NotImplementedError
        ##########################################################################################

    def forward(self, x):
        ##########################################################################################
        # IMPLEMENT ME!
        raise NotImplementedError
        ##########################################################################################

gen = Generator()
disc = Discriminator()
if torch.cuda.is_available:
    gen = gen.cuda()
    disc = disc.cuda()
```

## 2.2 Generate image samples from generator (10 pts)

During the training and inference, the generator needs to generate batch of images from random noise. Implement the generation function below.

**Note:** This function will later be used in training, therefore you need to be careful and avoid cutting off the gradient accidentally

In [4]:
```
def generate_samples(model, num_samples):
    ##########################################################################################
    # IMPLEMENT ME!
    # The shape of the returned samples should be [num_samples, H, W]
    raise NotImplementedError
    ##########################################################################################
```

Without any training, the samples generated by the generator does not resemble any digit in the dataset.

In [ ]:
```
gen.eval()
samples = generate_samples(gen, 25).detach().cpu().numpy()
plt.figure(dpi=90)
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.show()
```

## 2.3 GAN training algorithm (25 pts)

In this part, you will implement the GAN training algorithm, which involves alternating the training of discriminator and generator.

**Note:** You may start with the standard GAN loss (Goodfellow, Ian, et al. "Generative adversarial networks." Communications of the ACM 63.11 (2020): 139-144.). If you have extra time and resources, you can explore other GAN formulations that either improves convergence or alleviates the mode-collapse problem. For instance, LSGAN (Mao, Xudong, et al. "Least squares generative adversarial networks." Proceedings of the IEEE international conference on computer vision. 2017.) and Wasserstein GAN (Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." International conference on machine learning. PMLR, 2017.) are both great options.

In [ ]:
```
def train_gan(gen, disc, images, num_epochs, batch_size):
    from tqdm.autonotebook import tqdm
    from torch.utils.data import DataLoader

    losses_gen = []
    losses_disc = []

    gen.train()
    disc.train()
    loader = DataLoader(images, batch_size=batch_size, shuffle=True)

    ##########################################################################################
    # IMPLEMENT ME!

    # 1. [5 pts] Build optimizer for each model and choose an appropriate learning rate
```

```
    #    You may specify different optimizers or learning rates for generator and discrminator
    #    to balance the training loss and avoid one model overpowering the other.
    #    Ideally we would like to reach an equilibrium between the generator and discriminator.
    optimizer_gen = ...
    optimizer_disc = ...

    pbar = tqdm(range(num_epochs))
    for e in pbar:
        for i, data_real in enumerate(loader):
            if torch.cuda.is_available:
                data_real = data_real.cuda()

            # 2. Update discriminator
            # 2.1. Unfreeze discriminator
            disc.train()
            disc.requires_grad_(True)

            # 2.2. [5 pts] Construct inputs and training labels for discriminator
            #      The discriminator training should use both real and fake samples
            #      Tip: since we do not want to update the generator, the fake samples
            #           need to be detached from the computation graph. You can use
            #           detach().clone() for this operation.
            inputs_disc = ...
            labels_disc = ...
            if torch.cuda.is_available:
                inputs_disc = inputs_disc.cuda()
                labels_disc = labels_disc.cuda()

            # 2.4. [5 pts] Discrminator training
            #      This should include loss computation and weight updates
            #      You can choose implement standard GAN, LSGAN, or Wasserstein gan loss
            loss_disc = ...
            losses_disc.append(loss_disc.item())

            # 3. Update generator
            # 3.1. Freeze discriminator
            disc.eval()
            disc.requires_grad_(False)

            # 3.2. [5 pts] Construct input and training labels for the generator
            #      The generator training only uses fake samples, since in this step
            #      only the generator will be updated.
            #      The training labels should be different from the discriminator
            #      labels since we want to optimize the generator in the opposite
            #      gradient direction in order to "fool" the discriminator.
            inputs_gen = ...
            labels_gen = ...
            if torch.cuda.is_available:
                inputs_gen = inputs_gen.cuda()
                labels_gen = labels_gen.cuda()

            # 3.4. [5 pts] Generator training
            #      This should include loss computation and weight updates
            #      You can choose implement standard GAN, LSGAN, or Wasserstein gan loss
            loss_gen = ...
            losses_gen.append(loss_gen.item())
    ##########################################################################################

            pbar.set_postfix(loss_gen=losses_gen[-1], loss_disc=losses_disc[-1])
    return losses_gen, losses_disc
```

You may change the number of epochs and batch size based on the time and resources available to you.

In [ ]:
```
num_epochs = ...
batch_size = ...
losses_gen, losses_disc = train_gan(gen, disc, images, num_epochs=num_epochs, batch_size=batch_size)
```

## 2.4 Observations and Analysis (5 pts)

**[2.5 pts]** Use the code below to plot the losses for the generator and discriminator. Do you see any problem with the training process? Do you think your GAN has converged? Why and why not?

**Answer:**

In [ ]:
```
plt.figure(dpi=90)
plt.plot(losses_disc, label='Dicriminator')
plt.plot(losses_gen, label='Generator')
plt.xlabel('Iterations')
```

```
plt.ylabel('Loss')
plt.legend()
plt.show()
```

**[2.5 pts]** Use the code below to visualize some images generated by your GAN. Do you see any problem with the samples generated by your GAN? Do you think your GAN has mode collapse problem? Why and why not?

**Answer:**

In [ ]:
```
gen.eval()
samples = generate_samples(gen, 25).detach().cpu().numpy()
plt.figure(dpi=90)
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.show()
```