2023.11.23 - homework 3

November 23, 2023

0.0.1 2023.11.23 - Introduction to Transformers Continued | Homework 3

Proximal Policy Optimization (PPO) is a type of policy gradient method for reinforcement learning that was used in the final training stage of GPT. In this homework, you will implement a highly simplified version of the PPO algorithm to get a foundational understanding of its mechanics. The aim is to outline the whole concept with mostly static components, to give you a birds eye view of how it works.

The endresult is a model that learned that the reward model prefers positive words over negative ones.

We know that there is quite a long path between this excercise and the real world application of PPO. However, since this is an introductory lecture and due to the vast diversity of the audience we will noo go into more depth. In case you are interested in diving deeper into the realm of reinforcement learning and PPO, checkout the following resources: - OpenAI Spinning Up - Lilian Weng Policy Gradient Algorithms - Proximal Policy Optimization (PPO): The Key to LLM Alignment

Base your code on the following skeleton code that we provide.

We tried to explain our simplifications as much as possible, to help you on the one hand understand the high level concept and on the other are able to connect it to real world implementation details.

[ ]:
```
import torch
from torch import nn, optim
import random
import matplotlib.pyplot as plt
from scipy.signal import medfilt
# Dependencies you might need
```

Agent and Policy: In RLHF for LLMs, the agent is the pre-trained language model itself. The policy is represented by the model's forward function. In our case instead of using a full language model we will use a simple Multilayer perceptron (MLP) that in the end should predict positive words more likeyl than negative words, independently of the input.

[ ]:
```
# Define the list of words and their corresponding "positivity" score
word_scores = {"happy": 1, "joyful": 1, "pleasant": 1, "sad": -1, "angry": -1,␣
 ↪"unpleasant": -1}
words = list(word_scores.keys())
positive_words = set({k:v for k,v in word_scores.items() if v == 1}.keys())
```

```
[ ]: class Agent(nn.Module):
        """
        The Agent class represents a simplified version of a policy model used in␣
     ↪Proximal Policy Optimization (PPO) that operates as
        a basic language model. The model is a Multilayer Perceptron (MLP) designed␣
     ↪to predict the likelihood of positive words
        over negative ones, regardless of the input.

        Example:
            # Example instantiation and forward pass
            agent = Agent(num_words)
            input_data = torch.tensor([...])  # Some one-hot encoded input data
            output = agent(input_data)
        """
        # A simple MLP with 2 linear layers and a ReLU activation in between
        def __init__(self, num_words):
            super(Agent, self).__init__()
            """
            TODO: Define the architecture of the MLP: Decide on the number of␣
     ↪layers, units in each layer, and activation functions.
            Hint: You can start as simple as 2 linear layers of low dimension (e.g.␣
     ↪128) with a simple ReLU activation function in between. (nn.Linear, torch.
     ↪relu)
            """
            pass

        def forward(self, x):
            """
            TODO: Implement the forward pass: Define how the input data flows␣
     ↪through the network to produce output.
            The output should be a probability distribution over words. (Hint: you␣
     ↪might want to use torch.softmax for that)
            """
            pass
```

Value Function/Critic/Reward Model The critic is a separate model or mechanism that evaluates the quality of the outputs generated by the agent. In the context of RLHF, this could be a model trained to predict human preferences or judgments regarding the quality or appropriateness of the model's responses.

In our case we just return the predefined positivy score for a given word.

```
[ ]: class Critic:
        """
        A simple static model that simply uses our word_score lookup table to␣
     ↪"judge" a given input.
```

```
    In real scenarios, critics can be complex models trained to evaluate text␣
↪based on various criteria like coherence, grammar,
    and alignment with human preferences.
    """
    def __init__(self):
        super(Critic, self).__init__()

    def forward(self, word):
        return torch.tensor([word_scores.get(word, 0)])

    def __call__(self, input):
        return self.forward(input)
```

Baseline The primary purpose of using a baseline is to reduce the variance of the gradient estimate without biasing it. In policy gradient methods, the goal is to maximize the expected reward. However, the gradient of this expectation can have high variance, leading to inefficient learning. By subtracting a baseline from the reward, the variance of the gradient can be significantly reduced, which typically results in more stable and efficient learning.

[ ]:
```
class Baseline:
    """
    The Baseline class in this exercise represents a mechanism to calculate and␣
↪update a baseline value used
    in the Proximal Policy Optimization (PPO) algorithm. The baseline is␣
↪utilized to reduce the variance in the
    policy gradient estimation, leading to more stable and efficient learning.

    In this exercise, you will implement and use the Baseline class to compute␣
↪an average score.

    Example:
        # Example of updating and getting the baseline
        baseline = Baseline()
        baseline.update(new_score=1.0)
        current_baseline = baseline.get()
    """
    def __init__(self):
        self.total_score = 0.0
        self.count = 0

    def update(self, new_score):
        # TODO: Implement the update method: Update the total score and count␣
↪based on the new score received after each action.
        pass

    def get(self):
```

```
        # TODO: Implement the get method: Calculate and return the current␣
↪average score (baseline).
        pass
```

Training Loop

[ ]:
```
def train(agent, critic, optim, epsilon=0.01, iter=100):
    """
    This function represents the training loop where the Proximal Policy␣
↪Optimization (PPO) algorithm is applied
    to train the Agent. The function takes the agent, critic, and optimizer as␣
↪inputs and performs iterative training.

    The loop involves selecting actions (words), evaluating them, and updating␣
↪the agent's policy based on the calculated loss.


    Parameters:
        agent (Agent): The policy model (MLP) being trained.
        critic (Critic): The model used to evaluate the quality of actions.
        optimizer (torch.optim): The optimizer used for updating the agent.
        epsilon (float): The probability of choosing a random action␣
↪(exploration).
        iter (int): The number of training iterations.

    Returns:
        tuple: A tuple containing the loss and positive word count metrics over␣
↪training iterations.

    Example:
        # Example training call
        agent = Agent(len(words))
        critic = Critic()
        optimizer = optim.Adam(agent.parameters(), lr=3e-4)
        losses, positive_word_counts = train(agent, critic, optimizer)
    """

    # Training Metrics
    losses = []
    positive_word_counts = []
    total_positive_words = 0

    onehot_encorder = torch.eye(len(words))

    for i in range(iter):
        # In our case our "environment" is just the random selection of a word
        random_idx = random.randint(0, len(words) - 1)
```

```
        input = onehot_encorder[random_idx]

        agent_output = agent(input)

        # TODO: Implement the logic for choosing actions based on exploration␣
↪or exploitation.
        # Hint: You might want to decide between choosing a random word_index␣
↪as a suggestions vs using something like torch.agrmax
        # on the agents output. (e.g. if random.random() < epsilon: choose␣
↪random word else choose based on probability distribution
        # Important this section should assign a value to the var "word_index"

        chosen_word = words[word_index] # chosen word (=the suggested action)

        # TODO: calculate the critic_score score

        # TODO: update the baseline using the critic_score

        # TODO: calculate the advantage as the difference between the␣
↪critic_score and the baseline

        # TODO: Calculate loss as the negative log of the probability of the␣
↪chosen_word multiplied by the advantage

        # Update the agent
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Record and compute moving average of the loss
        losses.append(loss.item())

        if chosen_word in positive_words:
            total_positive_words += 1
        positive_word_counts.append(total_positive_words / (i + 1))

        if i % 1 == 0:
            print(f"Iteration {i}, Loss: {loss.item()}, Generated Word:␣
↪{chosen_word}")

    return losses, positive_word_counts
```

Run Exercise 1 Run this cell to evaluate your implementation.

[ ]:
```
"""
Expected Outcome:
```

```
Iteration 0, Loss: 0.0, Generated Word: angry
Iteration 1, Loss: 2.910965919494629, Generated Word: pleasant
...
Note: The losses as well as the generated word will obviously be different.
However you should see that the number of positive words is increasing over␣
 ↪time.

!! Important !! : With low model complexity and random weights initalizaion the␣
 ↪model might get stuck
and produce a constant result with 0 loss. Just rerun this cell.
"""
agent = Agent(len(words))
critic = Critic()

baseline = Baseline()
optimizer = optim.Adam(agent.parameters(), lr=3e-4)

losses, positive_word_counts = train(agent, critic, optimizer)
```

[ ]:
```
"""
Expected Outcome:
You should see 3 figures, the first two should show a declining slop, the third␣
 ↪one should look more like a log function.
"""
filtered_loss = medfilt(losses, kernel_size=5)

# Plot the filtered loss
plt.figure(figsize=(18, 6))
plt.subplot(1, 3, 1)
plt.plot(losses)
plt.title("Raw Loss over Time")
plt.xlabel("Epoch")
plt.ylabel("Filtered Loss")

plt.subplot(1, 3, 2)
plt.plot(filtered_loss)
plt.title("Loss over Time")
plt.xlabel("Epoch")
plt.ylabel("Filtered Loss")

# Plotting the frequency of choosing positive words
plt.subplot(1, 3, 3)
plt.plot(positive_word_counts)
plt.title("Frequency of Choosing Positive Words")
plt.xlabel("Epoch")
plt.ylabel("Frequency")
```

```
plt.tight_layout()
plt.show()
```