COMS 6998-015
PRAC DEEP LEARNING SYS
Instructor: Parijat Dube and Chen Wang

Homework 3

Due: Nov. 4, 2024

Problem 1 - *SSD, ONNX model, ORT Inferencing* 25 points

In this problem we will be inferencing SSD ONNX model using ONNX Runtime Server. You will follow the github repo and ONNX tutorials (links provided below). You will start with a pretrained Pytorch SSD model and retrain it for your target categories. Then you will convert this Pytorch model to ONNX and deploy it on ONNX runtime server for inferencing.

1. Download pretrained pytorch MobilenetV1 SSD and test it locally using Pascal VOC 2007 dataset. Select any two related categories from Google Open Images dataset and finetune the pretrained SSD model. Examples include, Aircraft and Aeroplane, Handgun and Shotgun. You can use `open_images_downloader.py` script provided at the github to download the data. For finetuning you can use the same parameters as in the tutorial below. Compute the accuracy of the test data for these categories before and after finetuning. (4)

2. Export the Pytorch model to ONNX using `torch.onnx.export()` function and save it. When you export the model the function will execute the model, recording a trace of what operators are used to compute the outputs. Because export runs the model, we need to provide an input tensor `x`. The values in this can be random as long as it is the type and size. Use a dummy random tensor. (4)

3. Load the saved model using `onnx.load` and verify the model's structure using `onnx.checker.check_model`. (3)

4. Next run the model with ONNX Runtime (ORT). You first need to create an inference session for the model and then evaluate the model using the `run()` API. Show the output of your run. (4)

5. Does the output of PyTorch (from `torch.out`) and ONNX Runtime match? What precision you used to match? (2)

6. Test the inferencing set-up using 1 image from each of the two selected categories. For this you will need to load the images, preprocess them, and then do inference using the `run()` API from ORT. (4)

7. Parse the response message from the ORT and annotate the two images. Show inferencing output (bounding boxes with labels) for the two images. (4)

For parts 1 and 2 refer to the steps in the Github repo in the references. For part 3-6 you can refer to the Pytorch tutorial in the reference and the associated notebook. For part 7 you can refer to Pytorch tutorial for creating an inference request to ORT and to ONNX tutorial on how to preprocess an image. For part 8 you need to adopt the code at ONNX tutorial on parsing the ORT response.

References

- Github repo. Shot MultiBox Detector Implementation in Pytorch.
  Available at https://github.com/qfgaohao/pytorch-ssd
- Pytorch tutorial. Exporting a model from Pytorch to Onnx and running it using Onne runtime.
  Available at https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html
- ONNX tutorial. Inferencing SSD ONNX model using ONNX Runtime Server.
  Available at https://github.com/onnx/tutorials/blob/master/tutorials/OnnxRuntimeServerSSDModel.ipynb
- Google. Open Images Dataset V5 + Extensions.
  Available at https://storage.googleapis.com/openimages/web/index.html
- The PASCAL Visual Object Classes Challenge 2007.
  Available at http://host.robots.ox.ac.uk/pascal/VOC/voc2007/

Homework 3

Problem 2 - *Transfer learning: Shallow learning vs Finetuning, Pytorch* 25 points

In this problem, we will train a convolutional neural network for image classification using transfer learning. Transfer learning involves training a base network from scratch on a very large dataset (e.g., Imagenet1K with 1.2M images and 1K categories) and then using this base network either as a feature extractor or as an initialization network for a target task. Thus two major transfer learning scenarios are as follows:

- *Finetuning the base model*: Instead of random initialization, we initialize the network with a pre-trained network, like the one that is trained on the Imagenet dataset. The rest of the training looks as usual however the learning rate schedule for transfer learning may be different.
- *Base model as fixed feature extractor:* Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

1. For fine-tuning you will select a target dataset from the Visual-Decathlon challenge. Their website (link below) has several datasets which you can download. Select any one of the visual decathlon datasets and make it your target dataset for transfer learning. **Important : Do not select Imagenet1K as the target dataset.**
   (a) *Finetuning:* You will first load a pre-trained model (ResNet50) and change the final fully connected layer output to the number of classes in the target dataset. (2)
   (b) First finetune by setting the same value of hyperparameters (learning rate=0.001, momentum=0.9) for all the layers. Keep batch size of 64 and train for 150-200 epochs or until model converges well. You will use a multi-step learning rate schedule and decay by a factor of 0.1 ($\gamma = 0.1$ in the link below). You can choose steps at which you want to decay the learning rate but do 3 drops during the training. So the first drop will bring down the learning rate to 0.0001, second to 0.00001, third to 0.000001. For example, if training for 150 epochs, first drop can happen at epoch 40, second at epoch 80 and third at epoch 120. (6)
   (c) Next keeping all the hyperparameters same as before (including multi-step learning rate schedule), change the learning rate to 0.01 and 0.1 uniformly for all the layers. This means keep all the layers at same learning rate. So you will be doing two experiments, one keeping learning rate of all layers at 0.01 and one with 0.1. Again finetune the model, using the same mutli-step learning rate schedule as in Part (b), and report the final accuracy. How does the accuracy with the three learning rates compare? Which learning rate gives you the best accuracy on the target dataset? (6)
2. When using a pretrained model as feature extractor, all the layers of the network are frozen except the final layer. Thus except the last layer, none of the inner layers' gradients are updated during backward pass with the target dataset. Since gradients do not need to be computed for most of the network, this is faster than finetuning.
   (a) Now train only the last layer for 0.1, 0.01, and 0.001 while keeping all the other hyperparameters and settings same as earlier for finetuning. Which learning rate gives you the best accuracy on the target dataset? (6)
   (b) For your target dataset find the best final accuracy (across all the learning rates) from the two transfer learning approaches. Which approach and learning rate is the winner? Provide a plausible explanation to support your observation. (5)

Homework 3

For this problem the following resources will be helpful.

References

- Pytorch blog. Transfer Learning for Computer Vision Tutorial by S. Chilamkurthy
  Available at https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
- Notes on Transfer Learning. CS231n Convolutional Neural Networks for Visual Recognition.
  Available at https://cs231n.github.io/transfer-learning/
- Visual Domain Decathlon

Problem 3: Prompt Engineering for Math Problem Solving 25 points

Objective: To explore and optimize the effectiveness of different prompting techniques in guiding a Large Language Model (LLM) to solve a specific math word problem from the GSM8K dataset.

Tasks:

1. (3 points) **Dataset Setup:** Utilize the Hugging Face `datasets` library to load the GSM8K dataset. Randomly select one question from the test set for your experiments.
2. (5 points) **Model Selection:** Choose a suitable LLM from the Hugging Face Model Hub or OpenAI's API, considering factors like performance and computational resources.
3. (8 points) **Prompt Engineering:** Implement the following prompting functions for your chosen problem:
   - `generate_solution(prompt, problem)`: A generic function that takes a prompt and a problem as input and returns the model's generated solution.
   - `one_shot_prompting_numeric(problem_to_solve)`: Implements one-shot prompting with a focus on numerical answers.
   - `two_shot_prompting_numeric(problem_to_solve)`: Implements two-shot prompting, also focused on numerical answers.
   - `two_shot_cot_prompting(problem_to_solve)`: Implements two-shot Chain-of-Thought (CoT) prompting, encouraging step-by-step reasoning. Refer to the work of Wei et al. [2022] for more details on CoT prompting.
4. (3 points) **Prompt Refinement:** Experiment with variations of your prompts to improve accuracy for the chosen problem:
   - Try different phrasings, instructions, or examples in your prompts.
   - Explore adding specific instructions for common math operations or problem-solving strategies.
   - Experiment with prompts that encourage the model to double-check its work or consider alternative approaches.
5. (3 points) **Evaluation:** Test your implemented functions and refined prompts on your chosen problem. Compare the performance of different prompting techniques and analyze their effectiveness in producing the correct answer.

6. (3 points) **Summary:** In your notebook, include a brief summary (no more than 200 words) of your prompt engineering experience. Discuss the most effective prompting strategies you discovered, challenges you encountered, and any insights gained from the experiment.

Hints:

- Consider using the `transformers` library for model loading and inference if using Hugging Face models.
- Pay attention to how different wordings or structures in your prompts affect the model's reasoning process.
- Experiment with prompts that break down complex problems into smaller, manageable steps.
- Consider prompts that encourage the model to explain its reasoning, which may lead to more accurate results.

Submission:

- Submit your Jupyter notebook or Python script containing:
  - All implemented functions and experiments
  - The randomly chosen question and its correct answer
  - Results and outputs from your various prompting techniques
  - Your 200-word summary of the prompt engineering experience
- Ensure your code is well-commented and includes instructions for running the experiments

Question 4: Implement ReAct Agent with Multiple Tools 25 points

Implement a ReAct (Reasoning and Acting) agent as described by Yao et al. [2022], incorporating three main tools: search, compare, and analyze. This agent should be able to handle complex queries by reasoning about which tool to use and when.

1. (4 points) Implement the search tool using the SerpAPI integration from previous questions. Ensure it can be easily used by the ReAct agent.
   - Proper integration with SerpAPI
   - Formatting the search results for use by the ReAct agent
2. (5 points) Create a custom comparison tool using LangChain's `Tool` class. The tool should accept multiple items and a category as input and return a comparison result.
   - Implementing the comparison logic
   - Creating an appropriate prompt template for the comparison
   - Proper error handling for invalid inputs
3. (5 points) Implement an analysis tool that can summarize and extract key information from search results or comparisons. This tool should use the OpenAI model to generate insightful analyses.
   - Implementing the analysis logic
   - Creating an appropriate prompt template for the analysis

- Ensuring the analysis output is concise and relevant

4. (6 points) Integrate these tools with a ReAct agent using LangChain. Your implementation should:
   - Use LangChain's `initialize_agent` function with the `AgentType.ZERO_SHOT_REACT_DESCRIPTION` agent type
   - Include all three tools (search, compare, analyze) as available actions for the agent
   - Implement proper error handling and fallback strategies
   - Ensure smooth transitions between tools in the agent's reasoning process
5. (5 points) Implement a simple Streamlit user interface for your ReAct agent. Your implementation should include:
   - A text input field for users to enter their queries
   - A button to submit the query and trigger the ReAct agent
   - A display area for showing the final results
   - (Optional) A section to display the step-by-step reasoning process of the ReAct agent

Submission Requirements

Please submit the following items as part of your solution:

1. Your complete code implementation for the ReAct agent and its tools.
2. A sample question that you used to test your tool (make it complex enough to demonstrate the use of multiple tools).
3. The final answer provided by your ReAct agent for the sample question.
4. The complete history traces of the ReAct agent for your sample question, showing its thought process, actions, and observations. Your traces should follow a format similar to this example:

```
Thought: I need to find information about top smartphones first
Action: Search[top smartphones 2023]
Observation: [Search results about top smartphones]
Thought: Now I should compare the top two options
Action: Compare[iPhone 14 Pro, Samsung Galaxy S23 Ultra, smartphones]
Observation: [Comparison result]
Thought: I should analyze this comparison for the user
Action: Analyze[comparison result]
Observation: [Analysis of the comparison]
Final Answer: [Your agent's final response to the user's query]
```

Ensure that your submission clearly demonstrates the agent's ability to reason about which tool to use and how to interpret the results from each tool. Your history traces should show a logical flow of thoughts, actions, and observations, culminating in a final answer that addresses the initial query.

Note: Ensure that your ReAct agent can seamlessly switch between these tools based on the task at hand. The agent should be able to reason about which tool to use next and how to interpret the results from each tool.

References

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Shunyu Yao, Jian Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/pdf/2210.03629.