15-780 HW6

Due 4/1

This homework will build upon the same code you used for HW5. We are not releasing separate starter code, but instead you should make a copy of the same notebook and Python file that you used for that assignment and write your solutions there.

Adam

Implement the Adam solver to train the language model you developed in HW5. Recall that the Adam updates in the notation we used in class are given by

$$u := \beta u + (1-\beta)\nabla f(\theta) \tag{1}$$
$$v := \gamma v + (1-\gamma)\nabla f(\theta)^2 \tag{2}$$
$$\hat{u} = u/(1-\beta^t) \tag{3}$$
$$\hat{v} = v/(1-\gamma^t) \tag{4}$$
$$\theta := \theta - \texttt{lr} \cdot \hat{u}/(\sqrt{\hat{v}} + \epsilon) \tag{5}$$

where t denotes the iteration number. You should use the following class template based upon the SGD optimizer `SGD` in the provided code (file `code_15780.py`) to develop your implementation:

```
class Adam:
    def __init__(self, params, lr=1e-3, beta=0.9, gamma=0.999, eps=1e-8):
        ...

    def step(self):
        ...

    def zero_grad(self):
        ...
```

Note that you will need to maintain the u and v terms internal to the class `Adam` (initialize both u and v to zero vectors)[1]. Train your network using Adam versus the original SGD optimizer and compare the relative loss after one epoch. Using the default learning rate of `1e-3` for Adam, the trained Transformer model should reach a lower evaluation loss than the one trained with SGD. Finally, it is also fine to use the `torch.optim.Adam` optimizer while debugging your implementation (but obviously not in your final solution).

Implement KV Caching for self-attenion

As discussed in class, implement the KV cache for the self-attention layer. You are free to do this in multiple ways (the manner we discussed in class was only one possibility), but your modified `SelfAttention` layer should have the following form:

[1] For reference, you can also check out the note from `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`.

```
class SelfAttention(Module):
    def __init__(self, d, num_heads, max_seq_len=None):
        ...

    def clear_cache(self):
        ...

    def forward(self, X, mask = None, use_kv_cache=False):
        ...
```

When `use_kv_cache` is set to `True`, the attention computation should take into account the key and value states in the kv cache. Also, the number of kv vectors in kv cache should not exceed `max_seq_len`. Remember to add additional `use_kv_cache` flags for the other layers that require it (i.e., `TransformerBlockPreNorm` and `LanguageModel`), so that these can be passed down through the forward pass of your network.

Efficient sampling

Finally, using the KV cache, implement an efficient sampler for your language model. A "naive" sampler that does not use the KV cache could look something like this:

```
def sample_transformer_naive(model, tokens, max_context, num_tokens, temperature=0.6):
    for i in range(num_tokens):
        probs = torch.softmax(model(tokens[None,-max_context:])[0,-1] / temperature,-1)
        tokens = torch.cat([tokens, torch.multinomial(probs, 1)])
    return tokens
```

In other words, this function passes the *entire* previous `max_context` tokens into the model each time. Implement an efficient version of the function by leveraging the KV cache, which would only pass a single new token to the function at each iteration. The implementation using KV cache should be faster in sampling than using the naive approach `sample_transformer_naive`.