Fine tuning pre-trained transformers

In case of any discrepancies or references, check out: AutoCompose.ipynb
After loading the datasets, and converting them into the forms suitable to be into the models, we now move towards the final steps:
Imports:

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel, GPT2Config,
AdamW, get_linear_schedule_with_warmup
```

Setting the device:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

(you would like to train your model on GPU to reduce the training time drastically)

Steps before training loop

1) Pretrained model:

```
# Load model configuration
config = GPT2Config.from_pretrained("gpt2")
```

(as we did before)

2) Creating a model instance

```
# Create model instance and set embedding length
model = GPT2LMHeadModel.from_pretrained("gpt2", config=config)
model.resize_token_embeddings(len(tokenizer))
```

3) Shifting the model to run on GPU

```
# Running the model on GPU
model = model.to(device)
```

4) Setting up the parameters

```
epochs = 4
warmup_steps = 1e2
sample_every = 100
```

epochs: The number of complete passes through the training dataset. Increasing epochs allows the model to learn more but risks overfitting if too high.
warmup_steps: A small number of initial training steps where the learning rate gradually increases, helping the model stabilize before using the full learning rate.

`sample_every`: Determines how frequently (in terms of training steps) you generate a sample or evaluate the model to monitor its performance.

5) Scheduling optimizer:

```
# Using AdamW optimizer with default parameters
optimizer = AdamW(model.parameters(), lr=5e-4, eps=1e-8)

# Toatl training steps is the number of data points times the
number of epochs
total_training_steps = len(train_dataloader)*epochs

# Setting a variable learning rate using scheduler
scheduler = get_linear_schedule_with_warmup(optimizer,

num_warmup_steps=warmup_steps,

num_training_steps=total_training_steps)
```

Optimizer setup (`AdamW`):

1) Uses the AdamW optimizer, which combines adaptive learning rates with weight decay for better generalization.
2) Parameters:
    a) `lr=5e-4`: Initial learning rate.
    b) `eps=1e-8`: Small value to prevent division by zero in calculations.

Total training steps:

- Computes the total number of steps for training as `len(train_dataloader) * epochs`, where:
    c) `len(train_dataloader)` is the number of batches in one epoch.
    d) `epochs` is the number of complete passes through the dataset.

Learning rate scheduler:

- Uses `get_linear_schedule_with_warmup` to adjust the learning rate during training:
    e) `num_warmup_steps`: Number of initial steps for gradual learning rate increase.
    f) `num_training_steps`: Total training steps for linear decay after warm-up.

6) This function is typically used to log or display training durations in a clear and readable format.

```
def format_time(elapsed):

  return str(datetime.timedelta(seconds=int(round(elapsed))))
```

Training loop

Initialization

```
total_t0 = time.time()
training_stats = []
model = model.to(device)
```

Explanation:

- Records the total start time for tracking training duration.
- Prepares a list (`training_stats`) to store metrics like loss and time for each epoch.
- Moves the model to the device (CPU/GPU) for computation.

Epoch Start

```
for epoch_i in range(epochs):

  print(f'Beginning epoch {epoch_i+1} of {epochs}')



  t0 = time.time()

  total_train_loss = 0

  model.train()
```

Explanation:

- Begins a new epoch.
- Logs the epoch number, resets the epoch start time, and cumulative training loss.
- Puts the model in training mode.

Processing Training Batches

```
for step, batch in enumerate(train_dataloader):

    b_input_ids = batch[0].to(device)

    b_labels = batch[0].to(device)

    b_masks = batch[1].to(device)
```

```
    model.zero_grad()


    outputs = model(b_input_ids,
                    labels=b_labels,
                    attention_mask=b_masks)


    loss = outputs[0]


    batch_loss = loss.item()
    total_train_loss += batch_loss
```

Explanation:

- Loops through each batch in the training data.
- Sends input IDs, labels, and attention masks to the device.
- Resets gradients from the previous step.
- Performs a forward pass to compute the loss for the batch.
- Tracks the loss for the current batch and accumulates it.

Sampling and Monitoring

```
# Sampling every x steps
    if step != 0 and step % sample_every == 0:


      elapsed = format_time(time.time()-t0)
      print(f'Batch {step} of {len(train_dataloader)}. Loss:
{batch_loss}. Time: {elapsed}')
      model.eval()
```

```
      sample_outputs = model.generate(

bos_token_id=random.randint(1,30000),
                                      do_sample=True,
                                      top_k=50,
                                      max_length = 200,
                                      top_p=0.95,
                                      num_return_sequences=1
                                   )
      for i, sample_output in enumerate(sample_outputs):
        print(f'Example ouput: {tokenizer.decode(sample_output,
skip_special_tokens=True)}')
      print()
      model.train()
```

Explanation:

- At intervals defined by `sample_every`, logs the elapsed time and current loss.
- Temporarily switches to evaluation mode to generate text samples.
- Uses `model.generate` to produce a sample sequence, decodes it, and displays the output.
- Resumes training mode afterward.

Backpropagation and Optimization

```
    loss.backward()

    optimizer.step()

    scheduler.step()
```

Explanation:

- Computes gradients via backpropagation (`loss.backward`).
- Updates the model's weights using the optimizer.
- Adjusts the learning rate using the scheduler.

Epoch Completion

```
  avg_train_loss = total_train_loss / len(train_dataloader)

  training_time = format_time(time.time()-t0)

  print(f'Average Training Loss: {avg_train_loss}. Epoch time:
{training_time}')

  print()
```

Explanation:

- Computes the average loss over all batches in the epoch.
- Calculates and logs the time taken for the epoch.

Validation

```
  t0 = time.time()
  model.eval()

  total_eval_loss = 0
  nb_eval_steps = 0

  for batch in val_dataloader:
    b_input_ids = batch[0].to(device)
    b_labels = batch[0].to(device)
    b_masks = batch[1].to(device)

    with torch.no_grad():

        outputs  = model(b_input_ids,
                         attention_mask = b_masks,
                         labels=b_labels)
```

```
        loss = outputs[0]

    batch_loss = loss.item()
    total_eval_loss += batch_loss

  avg_val_loss = total_eval_loss / len(val_dataloader)
  val_time = format_time(time.time() - t0)
  print(f'Validation loss: {avg_val_loss}. Validation Time:
{val_time}')
  print()
```

Explanation:

- Switches to evaluation mode.
- Iterates over the validation data to compute the validation loss.
- Uses `torch.no_grad()` to disable gradient calculations, improving efficiency during validation.
- Logs the average validation loss and validation time.

Logging Statistics

```
  training_stats.append(

      {

          'epoch': epoch_i + 1,

          'Training Loss': avg_train_loss,

          'Valid. Loss': avg_val_loss,

          'Training Time': training_time,

          'Validation Time': val_time

      }

  )

  print("------------------------------")
```

Explanation:

- Appends the metrics (training loss, validation loss, training/validation time) for the epoch to `training_stats`.

Total Training Time

```
print(f'Total training took {format_time(time.time()-total_t0)}')
```

Explanation:

- Logs the total time taken for the entire training process.

After training:

Generating poems/content:

```
model.eval()

prompt = "<|BOS|>..." #feel free to experiment with it

generated = torch.tensor(tokenizer.encode(prompt)).unsqueeze(0)
generated = generated.to(device)

sample_outputs = model.generate(
            generated,
            do_sample=True,
            top_k=50,
            max_length=300,
            top_p=0.95,
            num_return_sequences=3,
            temperature=1.0,
            repetition_penalty=1.2,
            no_repeat_ngram_size=2,
            early_stopping=True
      )


for i, sample_output in enumerate(sample_outputs):
  print("{}: {}\n\n".format(i, tokenizer.decode(sample_output,
skip_special_tokens=True)))
```

Parameter Explanations:

1. **`do_sample`**: Enables random sampling instead of deterministic greedy decoding, encouraging more diverse outputs.
2. **`top_k`**: Limits word selection to the top `k` most probable options, balancing diversity and coherence.
3. **`max_length`**: Defines the maximum number of tokens in the generated output.
4. **`top_p`**: Implements nucleus sampling, selecting tokens from the smallest set whose cumulative probability exceeds `p`.
5. **`num_return_sequences`**: Specifies the number of different sequences to generate per input.
6. **`temperature`**: Adjusts randomness in sampling; values >1 increase diversity, and values <1 make outputs more deterministic.
7. **`repetition_penalty`**: Discourages repetitive n-grams in generated text by penalizing them.
8. **`no_repeat_ngram_size`**: Prevents the repetition of n-grams of the specified size, ensuring variety in output.
9. **`early_stopping`**: Halts generation if an end-of-sequence token is produced, avoiding unnecessary extension of output.

This setup ensures diverse and controlled text generation while mitigating redundancy.

Saving the model:

```
import torch
import os

# Specify the directory where the model will be saved
output_dir = "/content/drive/My Drive/……."

# Check if model uses DataParallel and save the correct model
model_to_save = model.module if hasattr(model, 'module') else model

# Save the trained model, configuration, and tokenizer
model_to_save.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)

print(f'Model and tokenizer saved to {output_dir}')
```

Explanation:

1. **`output_dir`**:
   Specifies the path where the model, tokenizer, and training stats will be saved.

2. **`model_to_save`**:
   If you're using multiple GPUs with `DataParallel`, the model is wrapped in `model.module`. This checks if the model is wrapped and extracts the underlying model; otherwise, it saves the model directly.

3. **`save_pretrained(output_dir)`**:
   Saves the model and tokenizer in the specified directory. This function saves all necessary files for the model to be reloaded later using `from_pretrained()`.

4. **`print(f'Model and tokenizer saved to {output_dir}')`**:
   Prints a message confirming the save location for the model and tokenizer.

Some of the poems generated by my model:

The night was dark and the wind was slow
As I lay there singing
The refrain of a bird
Haunting the dead.

I thought my song was lovely,
And I sang in a language
Which the gods spoke
In languages
Of things dead and things alive.

———————————————

A lark, singing in the hollow of night,
Is setting fire to a pane, and burning it
With its flaming breath.

Heaven is a night when the air is fair, a day when
Hissing is not, but is still
As he gathers shade from the dark.

———————————————

I want to love you, dear,
Your voice is sweet and your eyes are kind, and I want
To hear your voice in my dreams.

I dreamed a great world, but I fear
The dark and shadowy corners of it;
You are a voice that I can never hear, except
Through dreams or from the deep dark.

———————————————

The sun has already set, and I am pacing home
from a long-term relationship.
I am not yet ready to say goodbye,
or seek another lover. I want to know
if I’m still here. The world seems so far away.

Between us, there were a thousand sparkling stars...
And circling us now, a hundred little stars.

But to-day I can only see the bright limb of your wrist
From my vision. Only a feeble wish
Had gone along—
That all the stars were pointing to the same place
In that farthest corner of our sky.

The night was starry, and so were her eyes!
And now, a few minutes ago,
They shone with the splendour of love,

In the garden of the south,
In a little garden near
The village;
But now she is gone, and the moon
Deems to fade away, and the air
Is full of darkness, and cold
Till the shadows and the mist
Come drifting back.
But my heart is still
And I feel it.

(My dataset wasn’t that large so it lacks consistency sometimes)