Final Project Report: Building GPT-2

May 13, 2025

Contents

1 Basic information

The proposal should have the following information at the top:

- **Title:** A Framework for Building and Fine-Tuning GPT-2 Models
- **Team members:**
  - Songlin Songlin (`zhousl24@tsinghua.edu.cn`)
    Contribution: Doing theoratical research and writing project proposal, implementing the structure of GPT-2 within the process of Stanford NLU Final Project Guidence, which is the first part of our project.
    Zhongwei Sun(`sun-zw23@mails.tsinghua.edu.cn`)
    Shibo Dai(`dsb24@mails.tsinghua.edu.cn`)
- **Custom or Default Project:** Default Project—Building GPT-2

2 Project description

- **Main Goals:**
  - Use the codebase that is adapted from the final project of Stanford CS224n with pre-trained of GPT-2 and implement some important components of the GPT-2 model to better understand its architecture. See this link.
  - Fine-tune the pretrained model on multiple downstream NLP tasks: 1) Sentiment Analysis, 2) Paraphrase Identification, and 3) Sonnet Generation.
  - Train a mini-GPT model from scratch without loading any pretrained transformer weights (word embeddings are allowed).
- **Downstream NLP Tasks:**
  - Sentiment Analysis on IMDB reviews (binary classification).
  - Paraphrase Detection on MRPC (sentence-pair classification).
  - Sonnet Generation given a short Shakespearean prompt (autoregressive generation).
- **Data Usage:**
  - *Pre-training:* None. The pre-trained data is provided within *NLU-Course-Project-GPT-and-Downstream-Tasks*
  - All we need to do is complete the implements of the missing block of algorithm such as Multi-headed Attention, Adam Optimization and position-wise Feed-Frward Network.
  - *Fine-tuning and Prediction for Sentiment Analysis:*
    - Stanford Sentiment Treebank (SST) (8,544 train / 1,101 dev / 1,101 test).
    - CFIMDB dataset (1,701 train / 245 dev / 488 test)
  - *Fine-tuning and Prediction for Paraphrase Detection:*
    - Quora dataset (141,506 train / 20,215 dev / 40,431 test)
  - Test sets and methods are also provided in the e final project of Stanford CS224n.
- **Methods:**
  - Use the GPT-2 structure provided within *NLU-Course-Project-GPT-and-Downstream-Tasks*: GPT-2 small (12 layers, 12 heads, 768-dim).
  - Training with Adam, linear warmup (10 % steps) + linear decay.
  - Fine-tuning: add classification heads for IMDB/MRPC; use beam search for sonnet generation.

- **Baselines:**
  - For SA:
    - Last Linear Layer for SST: Dev Accuracy: 0.462
    - Full Model for SST: Dev Accuracy: 0.513
    - Last Linear Layer for CFIMDB: Dev Accuracy: 0.861
    - Full Model for CFIMDB: Dev Accuracy: 0.976
  - For PD:
    - **IMDB Sentiment Analysis:**
    - BERT-base fine-tuned (Devlin et al., 2019): 94.5% accuracy.
    - GPT-2 zero-shot (Radford et al., 2019): 70.3% accuracy.
    - **MRPC Paraphrase Detection:**
    - BERT-base fine-tuned (Devlin et al., 2019): 88.9% accuracy, F1 = 89.2%.
    - GPT-2 zero-shot (Radford et al., 2019): 65.1% F1.
  - For Sonnet: Published BERT-base fine-tuned results (GLUE leaderboard).
- **Evaluation Metrics:**
  - See Baselines for SA.
  - *Accuracy* and *F1* for IMDB and MRPC.
  - *CHRF score* for sonnet generation.

3 Songlin Zhou: Implementing GPT-2

3.1 Folder and File Roles

`__pycache__` A local snapshot of the *official* HuggingFace GPT-2 weights; it is loaded on demand by `base_gpt.py` to avoid an on-line download.

`Downstream-tasks/` Example fine–tuning code for text classification, summarisation, *etc.* — not required in my assignment.

`models/` (central folder)

- `__pycache__` Compiled bytecode and cached tensors; reuse of the pre-trained weights.
- `base_gpt.py` Generic "*foundation*" class (`GPTPreTrainedModel`) : stores the `config`, performs weight initialisation and keeps track of global `dtype`. Every concrete GPT variant inherits from this class.
- `gpt2.py` Instantiates the Transformer stack defined in `base_gpt.py`, wiring the attention blocks, feed–forward layers, and the final LM head.

`modules/` `attention.py` Full implementation of *Causal Self-Attention* (multi–head, padding and causal masking, output projection).

`gpt2_layer.py` Combines `attention.py` with *Pre-LayerNorm* and the feed-forward MLP to form one Transformer block (`Dropout` $\rightarrow$ `Dense` $\rightarrow$ `Residual` per layer).

`test/` Minimal unit tests for GPT-2 model forward-pass correctness and for the `Adam` optimiser.

`config.py` Default hyper-parameters (hidden size, #layers, dropout probability, learning rate, ...).

`optimizer.py` A clean implementation of the `Adam` algorithm.

`README.md` Step-by-step running instructions.

3.2 Attention Module (`attention.py`)

`def __init__(self, config):` Initializes the three linear projection layers for **query**, **key** and **value**, each of shape $[H] \rightarrow [H]$, where $H = \text{hidden\_size}$. Imports all relevant hyperparameters from `config` (e.g. `num_attention_heads`, `hidden_size`, `attention_dropout`). Defines a `Dropout` layer that is applied to the *normalized* attention scores following the original Transformer paper. Although somewhat unusual, we empirically observe that this yields better generalisation.

`def transformer(self, x, linear_layer):` Given an input feature tensor

$$x \in \mathbb{R}^{B \times T \times H},$$

where B is batch size, T is sequence length and H is hidden dimension, applies a single linear projection (`linear_layer`) to obtain $\text{proj} \in \mathbb{R}^{B \times T \times H}$, then reshapes into multiple heads:

$$\text{proj} = \texttt{rearrange}(\text{proj}, \text{` } b\ t\ (h\ d) \rightarrow b\ t\ h\ d',\ h = \text{num\_attention\_heads}),$$

which yields a tensor of shape $[B, T, h, d]$ and enables parallel computation over h attention heads of size $d = H/h$ for speed and efficiency.

`def attention(self, key, query, value, attention_mask):` Computes masked, scaled dot–product self–attention. Given

$$Q, K \in \mathbb{R}^{B \times h \times T \times d}, \quad V \in \mathbb{R}^{B \times h \times T \times d},$$

it first forms the unnormalized scores

$$S = \frac{Q\,K^\top}{\sqrt{d_k}} \quad \Big[S \in \mathbb{R}^{B \times h \times T \times T}\Big],$$

then applies the mask:

$$S \leftarrow S + \text{attention\_mask},$$

and finally normalises with a dropout to keep robustness:

$$A = \text{softmax}(S,\ \text{dim} = -1), \quad A = \text{Dropout}(A).$$

The output context is

$$\text{context} = A\,V \in \mathbb{R}^{B\times h\times T\times d},$$

which is reshaped back to $[B, T, H]$ before returning.

`def forward(self, hidden_states, attention_mask):` In the feed-forward neural network we apply masking of future information to prevent the model from being influenced by future tokens during training, which would degrade prediction performance. Each `forward` call outputs an attention value that becomes a subcomponent in `gpt2_layer.py`.

3.3 Layer Module (`gpt2_layer.py`)

`def __init__(self, config):` Initializes all sub–modules for one Transformer block:

- Multi–head self–attention (`CausalSelfAttention`) parameters
- Two linear layers for the position-wise feed–forward network
- Two `LayerNorm` instances for Pre–LN
- Dropout probabilities from `config`

`def add(self, residual, sublayer_out, dense_layer, dropout):`

- Projects `sublayer_out` back to the hidden dimension via `dense_layer`.
- Applies `dropout` for regularisation.
- Adds the original `residual` tensor (residual connection).

`def forward(self, hidden_states, attention_mask):` Implements one full block in the Pre–LayerNorm style:

1. **Multi–Head Self–Attention**
   - (1-a) *Pre-normalize*: apply `LayerNorm` to `hidden_states`.
   - (1-b) *Attention*: compute self–attention with `attention_mask`.
   - (1-c) *Dropout + Residual*: use `add(...)` to project, drop out, and add back the input.
2. **Position-wise Feed–Forward**
   - (2-a) *Pre-normalize*: apply `LayerNorm` to the attention output.
   - (2-b) *MLP*: apply `Linear` $\rightarrow$ `GELU` $\rightarrow$ `Linear`.
   - (2-c) *Dropout + Residual*: use `add(...)` again to project, drop out, and add back.

3.4 GPT2 Module Summary (gpt2.py)

`def __init__(self, config):` Initializes embeddings, Transformer blocks, layer-norm and weights.

`def embed(self, input_ids):` Token + position lookup + dropout.

`def encode(self, hidden, mask):` Applies each layer with the extended attention mask.

`def forward(self, input_ids, attention_mask):` Embed → encode → final norm → select last token.

```
x    = self.embed(input_ids)
x    = self.encode(x, attention_mask)
x    = self.ln_f(x)
last = x[torch.arange(B), attention_mask.sum(1)-1]
return {'last_hidden_state': x, 'last_token': last}
```

`def hidden_state_to_token(self, h):` Weight–tie projection to vocabulary logits.

```
return h @ self.wte.weight.T
```

`@classmethod def from_pretrained(cls, ...):` Load HuggingFace weights into this model.

```
hf    = HFModel.from_pretrained(model_name)
model = cls(our_cfg)
model.load_state_dict(filter_weights(hf.state_dict()))
return model
```

3.5 Optimizer Module (optimizer.py)

`def __init__(self, params, lr, betas, eps, weight_decay, correct_bias):` Validates hyper-parameters, stores defaults, and calls the base `Optimizer` constructor.

`def step(self, closure=None):` Performs one optimization step of AdamW:

(1) **(Optional)** call `closure()` to recompute loss.

(2) Loop over each parameter group and each parameter `p`:

(3) **State Initialization** (first time only):

```
if len(state)==0:
    state["step"]       = 0
    state["exp_avg"]    = torch.zeros_like(p.data)
    state["exp_avg_sq"] = torch.zeros_like(p.data)
```

(4) **Moment Updates**:

```
state["step"] += 1
exp_avg.mul_(beta1).add_(grad, alpha=1-beta1)
exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1-beta2)
```

(5) **Bias-Corrected Step Size**:

```
if correct_bias:
    bc1 = 1 - beta1**t
    bc2 = 1 - beta2**t
    step_size = lr * math.sqrt(bc2) / bc1
else:
    step_size = lr
```

(6) **Parameter Update**:

```
denom = exp_avg_sq.sqrt().add_(eps)
p.data.addcdiv_(exp_avg, denom, value=-step_size)
```

(7) **Decoupled Weight Decay**:

```
if weight_decay != 0:
    p.data.add_(p.data, alpha=-lr * weight_decay)
```

Returns the (possibly recomputed) loss.