ADL HW2 Report

B09611048 蘇蓁葳

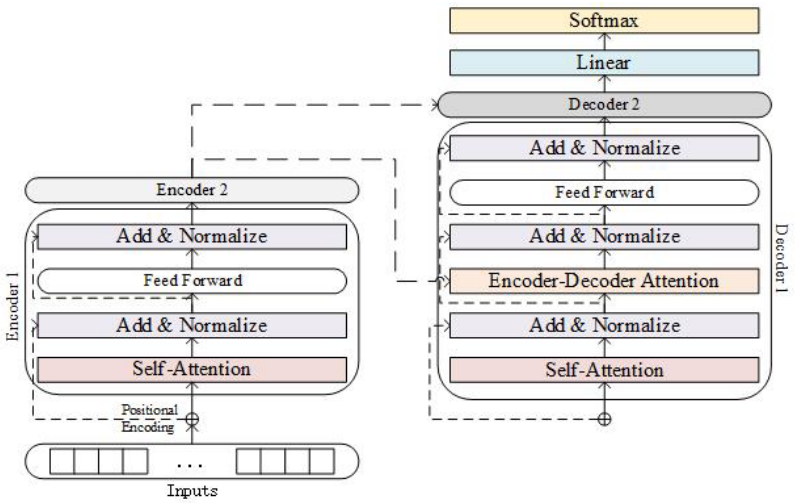
Model description:

I used google/mt5-small as base model to train my title generation model. The below is the config of mt5-small.


```
{
  "_name_or_path": "/content/drive/MyDrive/ADL/HW2/epoch3_1/",
  "architectures": [
    "MT5ForConditionalGeneration"
  ],
  "classifier_dropout": 0.0,
  "d_ff": 1024,
  "d_kv": 64,
  "d_model": 512,
  "decoder_start_token_id": 0,
  "dense_act_fn": "gelu_new",
  "dropout_rate": 0.1,
  "eos_token_id": 1,
  "feed_forward_proj": "gated-gelu",
  "initializer_factor": 1.0,
  "is_encoder_decoder": true,
  "is_gated_act": true,
  "layer_norm_epsilon": 1e-06,
  "model_type": "mt5",
  "num_decoder_layers": 8,
  "num_heads": 6,
  "num_layers": 8,
  "pad_token_id": 0,
  "relative_attention_max_distance": 128,
  "relative_attention_num_buckets": 32,
  "tie_word_embeddings": false,
  "tokenizer_class": "T5Tokenizer",
  "torch_dtype": "float32",
  "transformers_version": "4.34.1",
  "use_cache": true,
  "vocab_size": 250112
}
```


The model has 8 layers, 8 decoded layers and 0.1 dropout rate, more information can be checked in the config. Mt5 is a multilingual version of the original t5, it is pretrained on the mc4 (multilingual Common Crawl corpus) dataset and convers 101 languages.

The original t5 structure can be shown below, it contains 2 encoders and 1 decoder, after decoding the value, we will get a probability list in the end. Then we can use the probability list in different decoding algorithm to produced our result.



Preprocessing:

1. I used T5tokenizer as our tokenizer, it will first split the context into tokens and thus label the context to generate model readable vectors. An example is showed below. (Same as the ones in ALD report 1)
   (1) Split the context:
   The tokenizer will split the context into “readable words”.
   Example: [“ADL 作業好難嗚嗚”] -> [“ADL”, “作”, “業” , “好”, “難”, “嗚”, “嗚”]
   (2) Label the context:
   Since the model can only read “number”, the tokenizer will then convert the words into ids (or vectors) that the training model can read.
   Example: [“ADL”, “作”, “業” , “好”, “難”, “嗚”, “嗚”] -> [1234, 5678, 9876, 5432, 9876, 2345, 2345]
   (3) The AutoTokenizer will generate tokenize_id, attention_mask, offset mapping as well.

2. Prefix: if we did set the prefix for the model, before tokenizing the input, it will change all the inputs into “prefix” + “input”. But since I didn’t use prefix for this training, the input will remain the same.

3. After tokenizing the input, to prevent unscramble or shuffling in the evaluation part (which might cause mismatch for the result position and the output position). I added another column for the “id” to the model inputs generated by the tokenizer. By doing so, I can inference the id later on the evaluation stage and prevent mismatch. But for further investigation, this operation remains no use because the order of the sequence won’t easily change as long as you are not using parallel model predictions.

Hyperparameters:

I trained the epochs separately (3-1-1 for detail), since the optimizer algorithm AdamW will scale the learning rate for each epoch, I output the learning rate when I finish training a model and adjusting it on the next training. The below is the Hyperparameters for each training.

According to the description of the model (Link), if you are going to do finetuning for a single task with the model, there is no advantage for using a tack prefix, so I didn't add any prefix in training.

1st training:

Training parameters	Value/method
Learning rate	5e-5
Batch size	gradient_accumulation_steps (2) * per_device_train_batch_size (1) = 2
Loss Function	Cross Entropy
training epochs	3
optimization algorithm	AdamW
source_prefix	None

2nd training:

Training parameters	Value/method
Learning rate	2.5e-5
Batch size	gradient_accumulation_steps (2) * per_device_train_batch_size (1) = 2
Loss Function	Cross Entropy
training epochs	1
optimization algorithm	AdamW
source_prefix	None

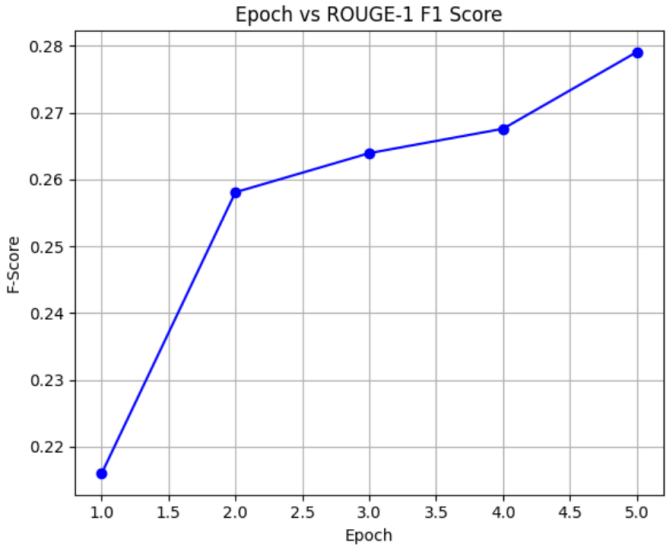
3nd training:

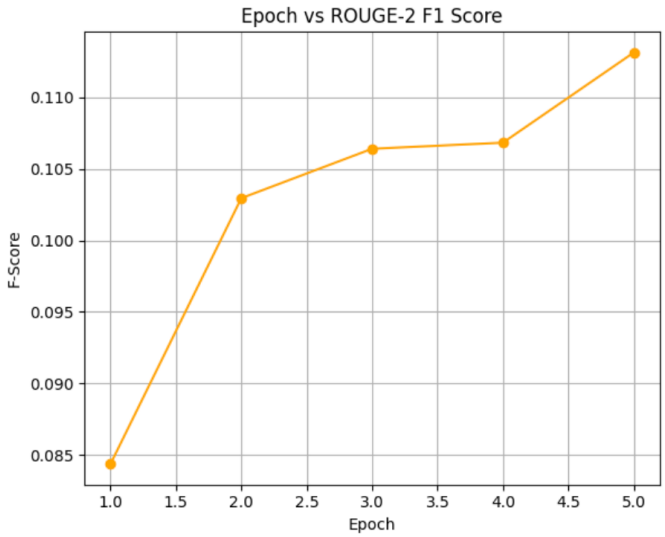
Training parameters	Value/method
Learning rate	2.5e-5
Batch size	gradient_accumulation_steps (2) * per_device_train_batch_size (1) = 2
Loss Function	Cross Entropy
training epochs	1
optimization algorithm	AdamW
source_prefix	None

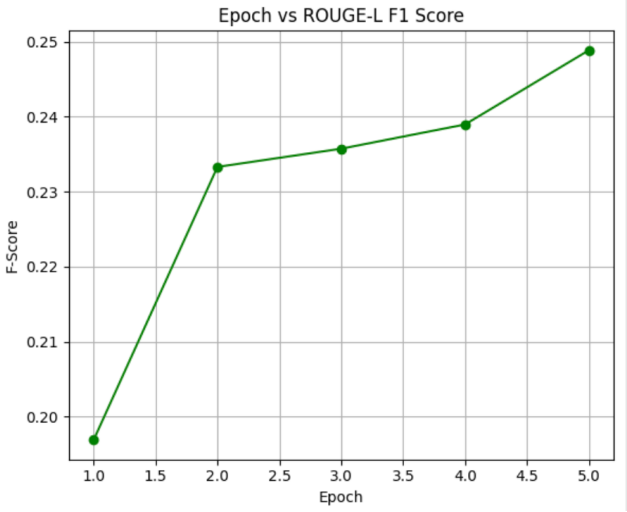
I trained the model for a total 5 epochs.

Learning Curve:

I used the rouge score that are calculated right after the training, and it is slightly difference with the result that is run by eval.py that the TA provide.







Strategies:

1. **Description for each strategies:**

(1) Greedy

Greedy is the simplest text generation. At each step in the generation, it selects the word or token with the highest probability as the next element. It will always pick the most likely next word given the previous sequence of words. The greedy search can generate fluent text, but the generation is lack of diversity and can result in repetitive or predictable text since it only choose the “safest” path at the time point without considering alternative, less probable, but have better relations and the options that is potentially more contextually appropriate the the given inputs.

(2) Beam search

When we use greedy search, we only take care of the most probable path. In this case, it will probably only chooses the safest path. But on the other hand, if you maintain the whole tree to find the most probable path in the end, the space and time complexity will be Exponential.

Beam search is a balance between greedy search and more exhaustive methods, often leading to better results as it can find more optimal sequences that the greedy approach might miss. Beam search will keep track on top n most probable candidates in each stage, instead of one comparing to greedy search. For each sequence in the beam, it looks at the next possible tokens and calculates their probabilities, and again choose the n most probabilities to continue. After all the "beams" reaches the EOS or the max sequence length, it will then choose the sequence with the highest total probability or the highest sum of log probabilities will be select as output.

(3) Top-k Sampling

In top-k sampling, the model first considers only the top 'k' most probable next tokens at each step of the generation process. From this subset, the next token is sampled randomly according to their probabilities. This prevents the model from focusing on the narrow set of high-probability choices while still allowing for the generation of coherent text. Top-k sampling introduces randomness into the text generation process, which helps to increase diversity and reduce the risk of getting stuck in repetitive loops.

(4) Top-p Sampling

Similar to top-k sampling, top-p takes more dynamic approach than top-k. Instead of choosing the top 'k' tokens, it selects the smallest set of tokens whose cumulative probability exceeds the threshold 'p'. The number of tokens in consideration will vary in each stage due to the probability distribution. This method allows for more diversity in generation and still focusing on the more probable parts of the distribution. It is less likely to produce completely irrelevant text.

(5) Temperature

To calculate the probability for each candidates in a single stage, we get the vector of scores for each candidates and put the vector of scores into the softmax function and calculate the probability list with the following function.

$$P(w) = \frac{\exp(s_w)}{\sum_{w' \in V} \exp(s_{w'})}$$

A Temperature is a hyperparameter for the soft max function, it will divide the score vector with a scalar τ. By adding the temperature, we can adjust the probability list into a more uniform or a more spciky one, and thus control the distribution of the word sets.

$$P(w) = \frac{\exp(s_w/\tau)}{\sum_{w' \in V} \exp(s_{w'}/\tau)}$$

2. Performance for each strategy:
   We can simply select different strategies by setting the kwargs in the summarization code, the performance of each setting of the code has been written below.

Generation strategies		Eval Scores (f-score)		
Strategies	Settings	rouge-1	rouge-2	rouge-l
Greedy	X	0.24473	0.09146	0.22081
Beam search	n=3	0.25655	0.10276	0.23012
	n=5	**0.25893**	**0.10444**	**0.23238**
Top-k Sampling	k=5	0.22659	0.07829	0.20271
	k=10	0.21708	0.073115	0.19297
Top-p Sampling	p=0.25	0.24162	0.08866	0.21671
	p=0.5	0.23297	0.08357	0.20838
Temperature (For top-p p=0.25)	$\tau = 5$	0.08138	0.00595	0.06903
	$\tau = 2$	0.17957	0.05351	0.15829

For the three rouge scores, the beam search n=5 has the best performance. And that is the one I choose for the inference, too.