Week 6

Fine-Tuning

An introduction

Parameter-Efficient Fine-Tuning

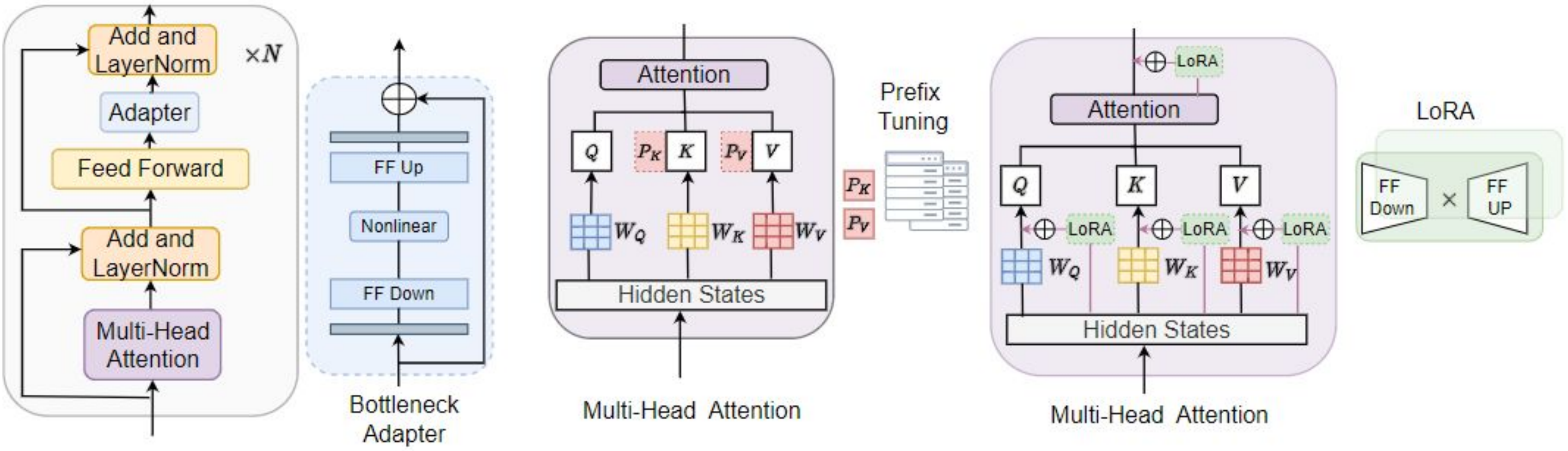
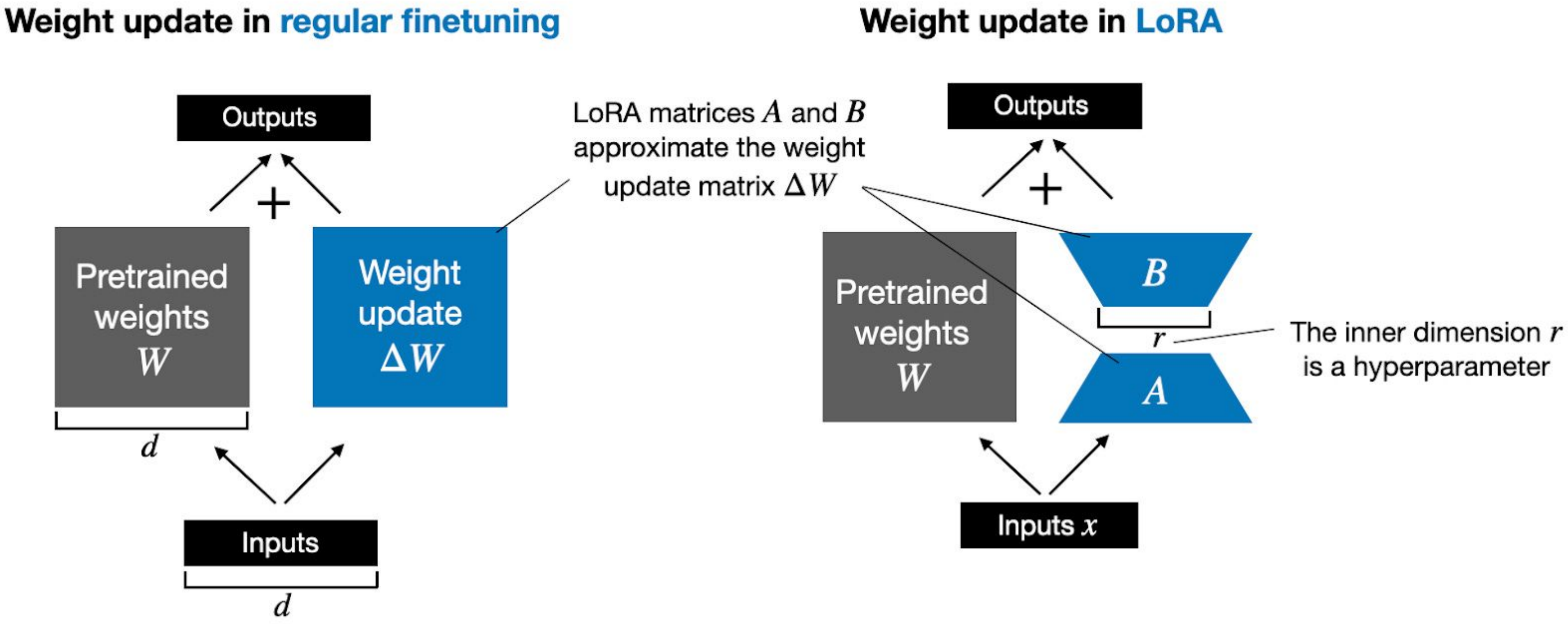


Fig. 2. Transformer architecture along with Adapter, Prefix Tuning, and LoRA.

Overview



Weight update in regular finetuning
Weight update in LoRA
Outputs
Pretrained weights W
Weight update ΔW
d
Inputs
d
LoRA matrices A and B approximate the weight update matrix ΔW
Outputs
Pretrained weights W
B
r
A
The inner dimension r is a hyperparameter
Inputs x

Base

```python
#
#
#
class Base(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.proj = torch.nn.Linear(10, 5)

  def forward(self, x):
    return self.proj(x)


#
#
#
base = Base()
print('\nFull:', sum(p.numel() for p in base.parameters()))
print('\n', base, '\n')

```

```
Full: 55

 Base(
  (proj): Linear(in_features=10, out_features=5, bias=True)
)
```

Adapter

```python
#
#
#
class Lora(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.proj = torch.nn.Linear(10, 5)
    self.lora = torch.nn.Sequential(
      torch.nn.Linear(10, 2, bias=False),
      torch.nn.ReLU(),
      torch.nn.Dropout(0.1),
      torch.nn.Linear(2, 5, bias=False)
    )

  def forward(self, x):
    x = self.proj(x)
    return x + self.lora(x)


#
#
#
lora = Lora()
print('==' * 50 + '\n')
print('Proj:', sum(p.numel() for p in lora.proj.parameters()))
print('Lora:', sum(p.numel() for p in lora.lora.parameters()))
print('Full:', sum(p.numel() for p in lora.parameters()))
print('\n', lora, '\n')
```

```
Proj: 55
Lora: 30
Full: 85

 Lora(
  (proj): Linear(in_features=10, out_features=5, bias=True)
  (lora): Sequential(
    (0): Linear(in_features=10, out_features=2, bias=False)
    (1): ReLU()
    (2): Dropout(p=0.1, inplace=False)
    (3): Linear(in_features=2, out_features=5, bias=False)
  )
)
```

FOUNDERS AND CODERS

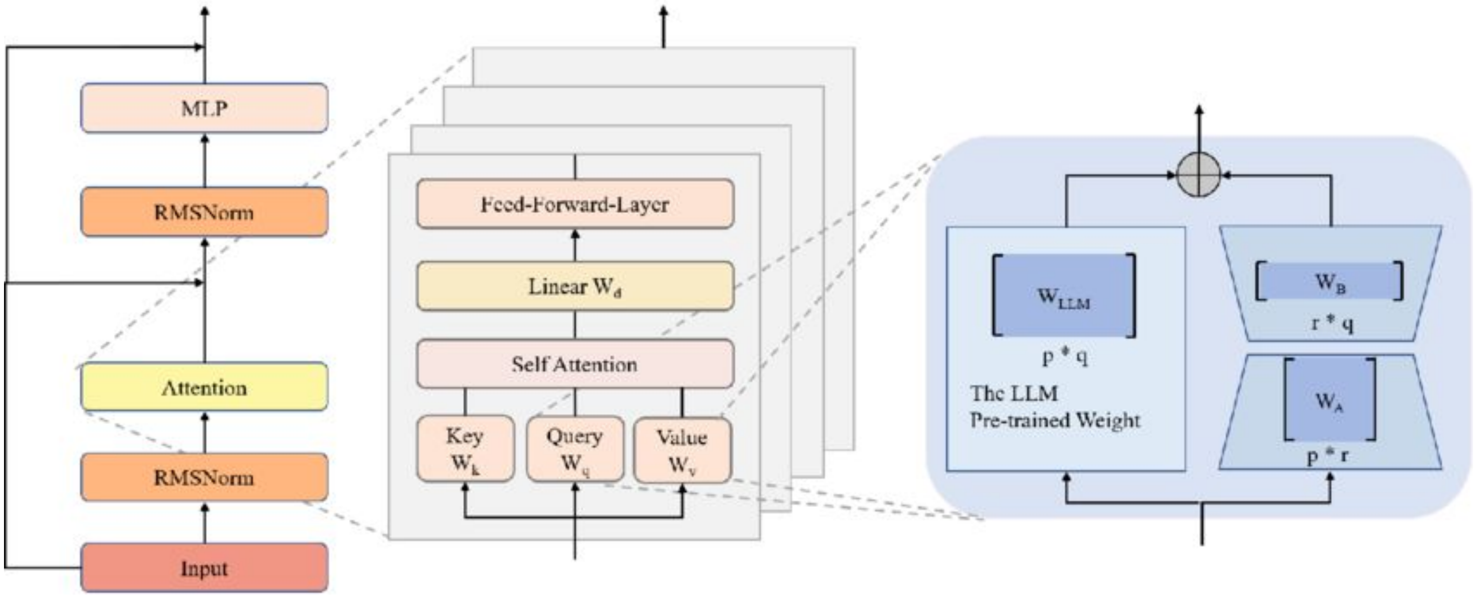
LoRA

Machine Learning Institute

```python
#
#
#
conf = peft.LoraConfig(
  r=2,
  lora_alpha=4,
  lora_dropout=0.1,
  target_modules=['proj']
)


#
#
#
boom = peft.get_peft_model(base, conf)
print('==' * 50 + '\n')
print('Full:', sum(p.numel() for p in boom.parameters()))
print('\n', boom, '\n')

```

```
Full: 85

 PeftModel(
  (base_model): LoraModel(
    (model): Base(
      (proj): lora.Linear(
        (base_layer): Linear(in_features=10, out_features=5, bias=True)
        (lora_dropout): ModuleDict(
          (default): Dropout(p=0.1, inplace=False)
        )
        (lora_A): ModuleDict(
          (default): Linear(in_features=10, out_features=2, bias=False)
        )
        (lora_B): ModuleDict(
          (default): Linear(in_features=2, out_features=5, bias=False)
        )
        (lora_embedding_A): ParameterDict()
        (lora_embedding_B): ParameterDict()
        (lora_magnitude_vector): ModuleDict()
      )
    )
```

LoRA



MLP
RMSNorm
Attention
RMSNorm
Input
Feed-Forward-Layer
Linear W_d
Self Attention
Key W_k
Query W_q
Value W_v
W_{LLM}
p * q
The LLM Pre-trained Weight
W_B
r * q
W_A
p * r

Prefix Tuning

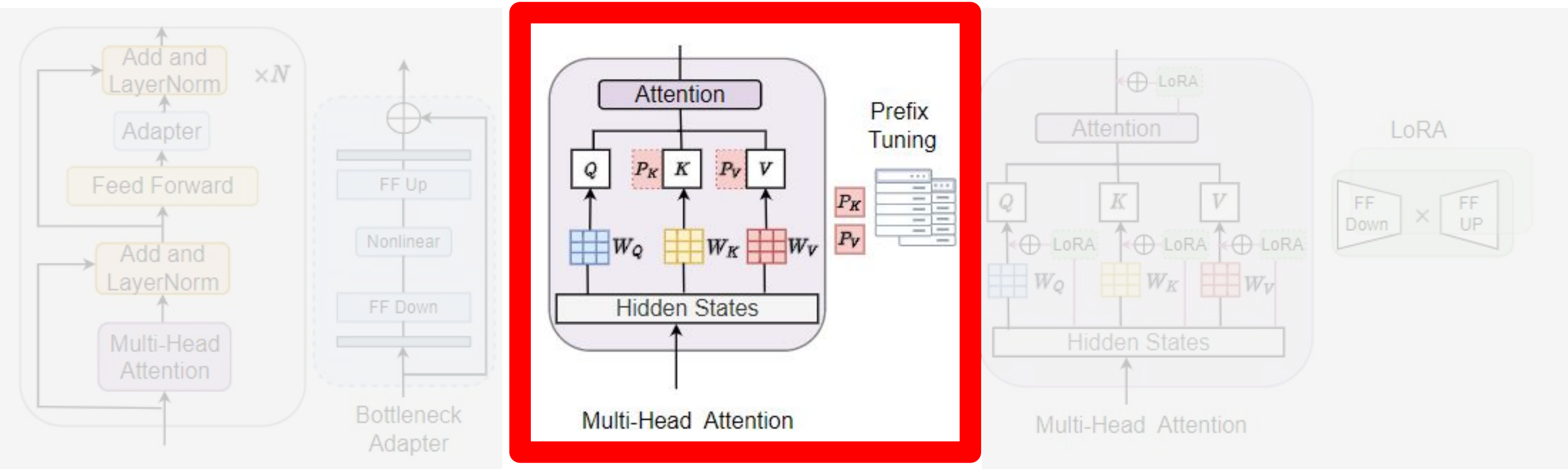


Fig. 2. Transformer architecture along with Adapter, Prefix Tuning, and LoRA.

LLaMA

```python
#
#
#
import transformers


#
#
#
tokenizer = transformers.AutoTokenizer.from_pretrained('meta-llama/Llama-3.2-1B-Instruct')
model = transformers.AutoModelForCausalLM.from_pretrained('meta-llama/Llama-3.2-1B-Instruct')


#
#
#
msgs = [{'role': 'user', 'content': 'What is the meaning of life, the universe, and everything?'}]
ids = tokenizer.apply_chat_template(msgs, tokenize=True)
print('=' * 100)
print(ids)


#
#
#
raw = tokenizer.decode(ids, skip_special_tokens=False)
print('=' * 100)
print(raw)

```

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 2048)
    (layers): ModuleList(
      (0-15): 16 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=512, bias=False)
          (v_proj): Linear(in_features=2048, out_features=512, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (down_proj): Linear(in_features=8192, out_features=2048, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (lm_head): Linear(in_features=2048, out_features=128256, bias=False)
)
```

FOUNDERS AND CODERS

GELU

Machine Learning Institute

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 2048)
    (layers): ModuleList(
      (0-15): 16 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=512, bias=False)
          (v_proj): Linear(in_features=2048, out_features=512, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (down_proj): Linear(in_features=8192, out_features=2048, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (lm_head): Linear(in_features=2048, out_features=128256, bias=False)
)
```


GLU Variants Improve Transformer

Noam Shazeer
Google
noam@google.com

February 14, 2020

Abstract

Gated Linear Units [Dauphin et al., 2016] consist of the component-wise product of two linear projections, one of which is first passed through a sigmoid function. Variations on GLU are possible, using different nonlinear (or even linear) functions in place of sigmoid. We test these variants in the feed-forward sublayers of the Transformer [Vaswani et al., 2017] sequence-to-sequence model, and find that some of them yield quality improvements over the typically-used ReLU or GELU activations.

1 Introduction

The Transformer [Vaswani et al., 2017] sequence-to-sequence model alternates between multi-head attention, and what it calls "position-wise feed-forward networks" (FFN). The FFN takes a vector x (the hidden representation at a particular position in the sequence) and passes it through two learned linear transformations, (represented by the matrices W_1 and W_2 and bias vectors b_1 and b_2). A rectified-linear (ReLU) [Glorot et al., 2011] activation function applied between the two linear transformations.

$$\mathrm{FFN}(x, W_1, W_2, b_1, b_2) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{1}$$

Following the T5 codebase [Raffel et al., 2019] [1], we use a version with no bias:

$$\mathrm{FFN}_{\mathrm{ReLU}}(x, W_1, W_2) = \max(xW_1, 0)W_2 \tag{2}$$

Subsequent work has proposed replacing the ReLU with other nonlinear activation functions such as Gaussian Error Linear Units, $\mathrm{GELU}(x) = x\Phi(x)$ [Hendrycks and Gimpel, 2016], and $\mathrm{Swish}_\beta(x) = x\sigma(\beta x)$ [Ramachandran et al., 2017].

$$\begin{aligned} \mathrm{FFN}_{\mathrm{GELU}}(x, W_1, W_2) &= \mathrm{GELU}(xW_1)W_2 \\ \mathrm{FFN}_{\mathrm{Swish}}(x, W_1, W_2) &= \mathrm{Swish}_1(xW_1)W_2 \end{aligned} \tag{3}$$

2 Gated Linear Units (GLU) and Variants

[Dauphin et al., 2016] introduced Gated Linear Units (GLU), a neural network layer defined as the component-wise product of two linear transformations of the input, one of which is sigmoid-activated. They also suggest omitting the activation, which they call a "bilinear" layer and attribute to [Mnih and Hinton, 2007].

$$\begin{aligned} \mathrm{GLU}(x, W, V, b, c) &= \sigma(xW + b) \otimes (xV + c) \\ \mathrm{Bilinear}(x, W, V, b, c) &= (xW + b) \otimes (xV + c) \end{aligned} \tag{4}$$

We can also define GLU variants using other activation functions:

[1] Also in the interest of ML fairness.

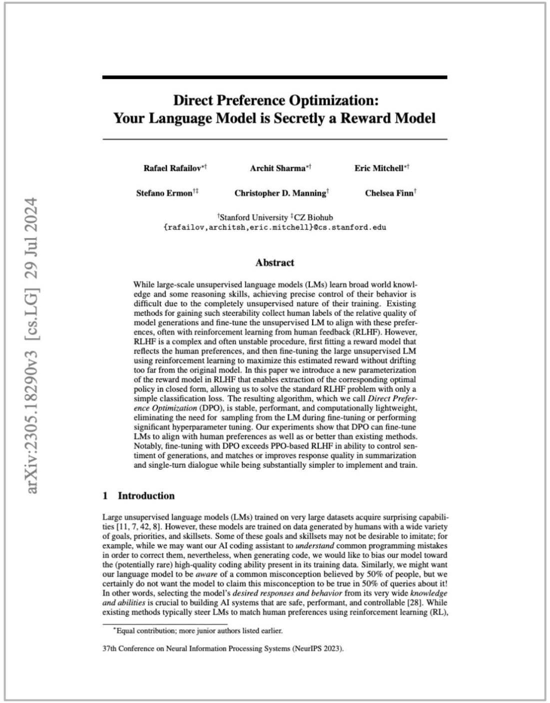
1

GELU

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

DPO



**Direct Preference Optimization:
Your Language Model is Secretly a Reward Model**

Rafael Rafailov*† Archit Sharma*† Eric Mitchell*†

Stefano Ermon†‡ Christopher D. Manning† Chelsea Finn†

†Stanford University ‡CZ Biohub
{rafailov,architsh,eric.mitchell}@cs.stanford.edu

arXiv:2305.18290v3 [cs.LG] 29 Jul 2024

Abstract

While large-scale unsupervised language models (LMs) learn broad world knowledge and some reasoning skills, achieving precise control of their behavior is difficult due to the completely unsupervised nature of their training. Existing methods for gaining such steerability collect human labels of the relative quality of model generations and fine-tune the unsupervised LM to align with these preferences, often with reinforcement learning from human feedback (RLHF). However, RLHF is a complex and often unstable procedure, first fitting a reward model that reflects the human preferences, and then fine-tuning the large unsupervised LM using reinforcement learning to maximize this estimated reward without drifting too far from the original model. In this paper we introduce a new parameterization of the reward model in RLHF that enables extraction of the corresponding optimal policy in closed form, allowing us to solve the standard RLHF problem with only a simple classification loss. The resulting algorithm, which we call *Direct Preference Optimization* (DPO), is stable, performant, and computationally lightweight, eliminating the need for sampling from the LM during fine-tuning or performing significant hyperparameter tuning. Our experiments show that DPO can fine-tune LMs to align with human preferences as well as or better than existing methods. Notably, fine-tuning with DPO exceeds PPO-based RLHF in ability to control sentiment of generations, and matches or improves response quality in summarization and single-turn dialogue while being substantially simpler to implement and train.

1 Introduction

Large unsupervised language models (LMs) trained on very large datasets acquire surprising capabilities [11, 7, 42, 8]. However, these models are trained on data generated by humans with a wide variety of goals, priorities, and skillsets. Some of these goals and skillsets may not be desirable to imitate; for example, while we may want our AI coding assistant to *understand* common programming mistakes in order to correct them, nevertheless, when generating code, we would like to bias our model toward the (potentially rare) high-quality coding ability present in its training data. Similarly, we might want our language model to be *aware* of a common misconception believed by 50% of people, but we certainly do not want the model to claim this misconception to be true in 50% of queries about it! In other words, selecting the model's *desired responses and behavior* from its very wide *knowledge and abilities* is crucial to building AI systems that are safe, performant, and controllable [28]. While existing methods typically steer LMs to match human preferences using reinforcement learning (RL),

*Equal contribution; more junior authors listed earlier.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

Rafailov et al. 2023

$$\mathcal{L}_{\mathrm{DPO}}(\pi_\theta; \pi_{\mathrm{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\mathrm{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\mathrm{ref}}(y_l \mid x)} \right) \right]$$

FOUNDERS AND CODERS

Code

Machine Learning Institute

```python
import torch
import transformers


#
#
#
tkz = transformers.AutoTokenizer.from_pretrained('gpt2')
plc = transformers.AutoModelForCausalLM.from_pretrained('gpt2')
ref = transformers.AutoModelForCausalLM.from_pretrained('gpt2')


#
#
#
ref.eval()
for p in ref.parameters(): p.requires_grad_(False)


#
#
#
optm = torch.optim.Adam(plc.parameters(), lr=1e-5)
beta = 0.1


#
#
#
qry = 'Explain the water-cycle so a 10-year-old can understand.'
pos = 'Think of Earths water as a superhero that never stops travelling.'
neg = 'The water cycle is evaporation, condensation and precipitation. That is all.'


#
#
#
def tokenise(qry, res):
  qry_ids  = tkz(qry, return_tensors='pt', add_special_tokens=False).input_ids
  res_ids  = tkz(res, return_tensors='pt', add_special_tokens=False).input_ids
  acc_ids  = torch.cat([qry_ids, res_ids], dim=-1)
  atn_msk  = torch.ones_like(acc_ids)
  lbl_ids  = acc_ids.clone()
  lbl_ids[:, :qry_ids.size(-1)] = -100
  return acc_ids, atn_msk, lbl_ids


#
#
#
def sum_logp(model, ids, msk, lbl):
  out  = model(input_ids=ids, attention_mask=msk)
  log  = out.logits.log_softmax(-1)[:, :-1]
  tgt  = lbl[:, 1:].masked_fill(lbl[:, 1:] == -100, 0).unsqueeze(-1)
  tok  = log.gather(2, tgt).squeeze(-1)
  msk  = lbl[:, 1:] != -100
  return (tok * msk).sum(-1)


#
#
#
ids_pos, atn_msk_pos, lbl_pos = tokenise(qry, pos)
ids_neg, atn_msk_neg, lbl_neg = tokenise(qry, neg)


#
#
#
with torch.no_grad():
  logp_ref_pos = sum_logp(ref, ids_pos, atn_msk_pos, lbl_pos)
  logp_ref_neg = sum_logp(ref, ids_neg, atn_msk_neg, lbl_neg)


#
#
#
for step in range(2):
  logp_plc_pos = sum_logp(plc, ids_pos, atn_msk_pos, lbl_pos)
  logp_plc_neg = sum_logp(plc, ids_neg, atn_msk_neg, lbl_neg)
  delta_pos = logp_plc_pos - logp_ref_pos
  delta_neg = logp_plc_neg - logp_ref_neg

  mrgs = delta_pos - delta_neg
  loss = -torch.log(torch.sigmoid(beta * mrgs))

  optm.zero_grad()
  loss.backward()
  optm.step()
  print(f'{loss.item():.4f}')
```

Task

- new knowledge injection

- Go multimodal
  - image to HTML/CSS
  - previous weeks tasks

- up to you :)

Bonus

What if I take my new data and ask the model to "summarise" it or repeat it in its own words.

Then the output would be from the same probability distribution i.e. the model itself. This way I could apply a KL divergence between a frozen model and the training model?

Thought for 38 seconds

Your idea is **innovative and theoretically sound**, especially when original training data is unavailable. By generating synthetic data via the model's own paraphrasing/summarization and using KL divergence between a frozen (original) model and the training model, you create an implicit regularization mechanism. Here's a breakdown of how this works, its benefits, and caveats:

Good luck!

Recurrent Rebels

- Loredana
- Stanley
- Dimitar
- Daniel

Gradient Gigglers

- Josh
- Andrea
- Kenton
- David

Perceptron Party

- Yurii
- Guillaume
- Neville
- Maxi

Overfitting Overlords

- Amy
- Milo
- Kori
- Evelyn

Backprop Bunch

- Ardrit
- Coline
- James
- Liam

Dropout Disco

- Aygun
- Dimitris
- Artemis
- Ollie

Activation Aces

- Nnamdi
- Pyry
- Gaurav
- Filippo