

テーマと目的と期待できる効果

- ・ テーマは現在すでにあるllmを参考してlanguage modelを自作してトレーニングしてchatbotにする
- ・ 今回含まれているもの：半分自作language model 半分自作事前学習hugging faceを使ってのfin tuening ,強化学習など
- ・ 今的目的：language modelに関連する知識と技術を身につける
- ・ これから的目的：個人のpcで動かすchart-gptなどのものを作りたい
- ・ 今期待できる効果：自分で構築したImでchatbotを作る
- ・ これから期待できる効果：個人用のchart-gptみたいなものの開発

結果

- 失敗した 入力は疑問の場合は大体nineを答える 他の場合は大体……で答える

```
[220] input=sample(sentence1,1)
```

▶ input

```
↳ ['this is it ! this is it ! quiet quiet quiet !']
```

```
[222] res=ge.generate(input,10,0.5)
```

▶ res[0][len(input[0]):]

```
' nine nine nine nine nine nine nine nine nine nine'
```

```
[247] input=sample(sentence1,1)
```

```
[248] input
```

```
['it s rough . i m sorry did she make it ?']
```

▶ res=ge.generate(input,20,0.8)

```
[233] input=sample(sentence1,1)
```

▶ input

```
['wrap it around that rock twice .']
```

▶ res=ge.generate(input,10,0)

▶ res[0][len(input[0]):]

```
↳ '.....'
```

まとめ、感想

- まとめ：言語モデルを作る順番:まずはtokenizerを決定それからどうノルムとembeddingを計算するかを決定 embeddingとattentionの計算を書く ffnを決定 blockを決定 モデルを構築(layerの数、出力語彙数などを決定) generatorを書く
- 事前学習(mask lm)など fine-tuning(lora) 強化学習(reward モデルを設置してppoで更新する)
- 感想:hugging face はとても使いやすい language modelの構築やトレーニングが難しい

これからもう詳しく説明する

- 数学の部分
- プログラムの部分
- 具体的なやり方
- どこを変えた
- 事前学習 (pre-training)をどうやった
- Fin-tuningをどうやった
- 強化学習

データについて

- 事前学習データはpytorchにあるWikiText2データセット これを少し処理する
- Fin-tuneingデータはpytorchのチュートリアルにある
Cornell Movie-Dialogsを使う チュートリアルでデータを処理する関数なども書いてあるのでそのまま使う
- 強化学習のデータはfin-tuningのデータとそれを入力して生成したデータを一つにすることで入力データとする（あと強化学習の部分で説明する）

データの様子

['Senjō no Valkyria 3 : <unk> Chronicles (Japanese : 戦場のヴァルキュリア3 , lit . Valkyria of the Battlefield 3) , commonly referred to as Valkyria Chronicles III outside Japan , is a tactical role @-@ playing video game developed by Sega and Media.Vision for the PlayStation Portable . Released in January 2011 in Japan , it is the third game in the Valkyria series . <unk> the same fusion of tactical and real @-@ time gameplay as its predecessors , the story runs parallel to the first game and follows the " Nameless " , a penal military unit serving the nation of Gallia during the Second European War who perform secret black operations and are pitted against the Imperial unit " <unk> Raven " . ' ,

"The game began development in 2010 , carrying over a large portion of the work done on Valkyria Chronicles II . While it retained the standard features of the series , it also underwent multiple adjustments , such as making the game more <unk> for series newcomers . Character designer <unk> Honjou and composer Hitoshi Sakimoto both returned from previous entries , along with Valkyria Chronicles II director Takeshi Ozawa . A large team of writers handled the script . The game 's opening theme was sung by May 'n . " ,

"It met with positive sales in Japan , and was praised by both Japanese and western critics . After release , it received downloadable content , along with an expanded edition in November of that year . It was also adapted into manga and an original video animation series . Due to low sales of Valkyria Chronicles II , Valkyria Chronicles III was not localized , but a fan translation compatible with the game 's expanded edition was released in 2014 . Media.Vision would return to the franchise with the development of Valkyria : Azure Revolution for the PlayStation 4 . " ,

"The game divided into five classes : Cavalry , Archery , Infantry , Siege and Mounted Cavalry . Each class has

```
[['they do to !', 'they do not !'],
['she okay ?', 'i hope so .'],
['wow', 'let s go .'],
['i m kidding . you know how sometimes you just become this persona ? and you don t know how to quit ?',
 'no'],
['no', 'okay you re gonna need to learn how to lie .'],
['i figured you d get to the good stuff eventually .', 'what good stuff ?'],
['what good stuff ?', 'the real you .'],
['the real you .', 'like my fear of wearing pastels ?'],
['do you listen to this crap ?', 'what crap ?'],
['what crap ?',
 'me . this endless . . blonde babble . i m like boring myself .']]
```

上は事前学習
データ下はfin-tuningのデータ
のペア

tokenizer

```
1 class Tokenizer(nn.Module):
2     def __init__(self, model_path: str):
3         # reload tokenizer
4         assert os.path.isfile(model_path), model_path
5         self.sp_model = SentencePieceProcessor(model_file=model_path)
6         self.n_words: int = self.sp_model.vocab_size()
7         self.bos_id: int = self.sp_model.bos_id()
8         self.eos_id: int = self.sp_model.eos_id()
9         self.pad_id: int = self.sp_model.pad_id()
10        #logger.info(f"#words: {self.n_words} - BOS ID: {self.bos_id} - EOS ID: {self.eos_id}")
11        assert self.sp_model.vocab_size() == self.sp_model.get_piece_size()
12
13    def encode(self, s: str, bos: bool, eos: bool) -> List[int]:
14        assert type(s) is str
15        t = self.sp_model.encode(s)
16        if bos:
17            t = [self.bos_id] + t
18        if eos:
19            t = t + [self.eos_id]
20    return t
21    def decode(self, t: List[int]) -> str:
22        return self.sp_model.decode(t)
```

Tokenizerはllamaのgithubのものをそのまま使うことにした

Tokenizerの設置について

- Llamaのgithubではsentence pieceを使うと書いてたのでそれを使う
- Sentence pieceは事前で訓練する必要がある 訓練データよく使う例とするbotchan.txtにした 語彙数を3000にした pad_id=3にした pad_id設置しない場合は-1になる この後のtorch.nn.nn.embeddingがエラーになる(入力がzero 以下)

```
1 spm.SentencePieceTrainer.Train(  
2     input='botchan.txt',  
3     model_prefix='m',  
4     vocab_size=3000,  
5     pad_id=3)  
6 sp = spm.SentencePieceProcessor()  
7 sp.load('m.model')  
8 |  
9 tokenizer=Tokenizer('m.model')
```

Pad_idを3以下の場合は
sentence pieceがエラー
する

データの処理(事前学習データ)

- 事前学習データの処理については 一つ一つのデータで以下の処理(このデータは一段落を一つ) : その中の単語と符号の数を合わせて90以上800以下のものを選ぶ トークン化してpaddingする
- Padding : トークン化したものの中で長さが一番長いものを選んで事前設置した最大の長さと比べて少ない方を選んで30を足す これをpaddingの長さにする

```
9 tokenizer=Tokenizer('m.model')
10 train_iter = WikiText2(split='train')
11 sentence=[]
12 for item in train_iter :
13     if item.strip() == '' : pass
14
15     elif len(item.strip()) <=90 : pass
16     elif len(item.strip()) >=800 : pass
17     else : sentence.append(item.strip())
18
```

```
[15] 1 token=[tokenizer.encode(x, bos=True, eos=True) for x in sentence]
2 max_batch_size: int = 32
3 max_seq_len: int = 2048
4 min_prompt_size = min([len(t) for t in token])
5 max_prompt_size = max([len(t) for t in token])
6 total_len = min(max_seq_len, 30 + max_prompt_size)
7
8 bsz=len(token)
9
10 tokens=torch.full((bsz, total_len), tokenizer.pad_id)
11 for k, t in enumerate(token):
12     tokens[k, :len(t)] = torch.tensor(t)
13
```

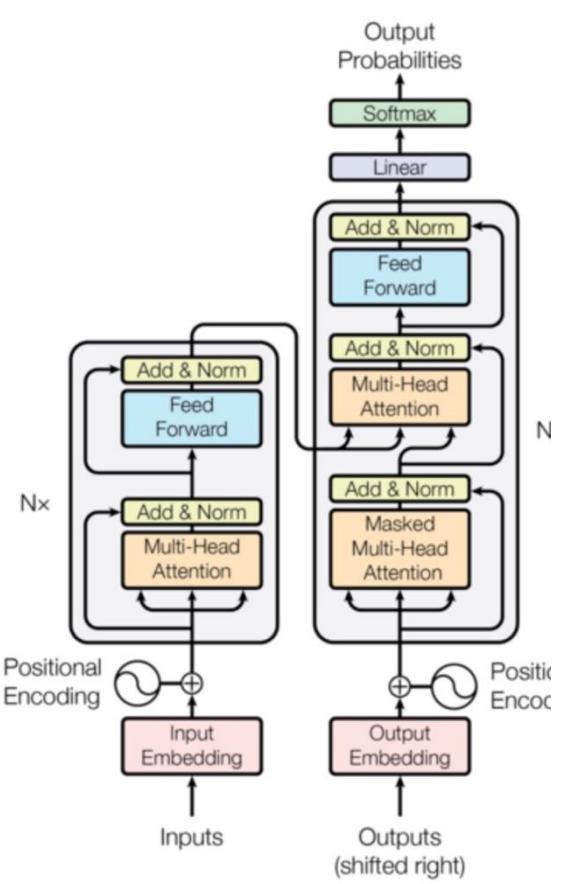
データの処理(fine-tuningデータ)

- Fin-tuningのデータの処理はpytorchのchatbotチュートリアルに書いてある その処理をそのままました
- Fine-tuning用データを先に話したものとその返事二つに分ける その中の単語と符号の数を100以下のものを選んでトークン化する
- あとの処理は前と同様

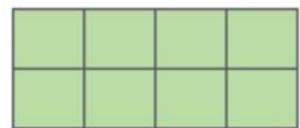
```
▶ 1 lines = open('formatted_movie_lines.txt', encoding='utf-8').\
2     read().strip().split('\n')
3     # Split every line into pairs and normalize
4 pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
```

```
▶ 1 con1=[ pairs[i][0] for i in range(len(pairs))]
2 con2=[ pairs[i][1] for i in range(len(pairs))]
3 sentence1=[]
4 sentence2=[]
5 for i in range(len(con1)) :
6     if len(con1[i]) <=100 and len(con2[i])  <=100:
7         sentence1.append(con1[i])
8         sentence2.append(con2[i])
9     else : pass
10 ts1=[tokenizer.encode(x, bos=True, eos=True) for x in sentence1]
11 ts2=[tokenizer.encode(x, bos=True, eos=True) for x in sentence2]
12 bsz=len(sentence1)
13 tokens1=torch.full((bsz, 100),tokenizer.pad_id)
14 for k, t in enumerate(ts1):
15     tokens1[k, : len(t)] = torch.tensor(t)
```

全体的に

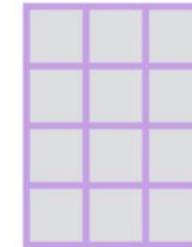


X

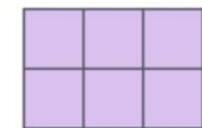


\times

W^Q

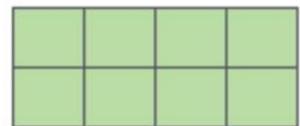


Q



Transformer の中の線形層の
ネットでの図をそのまま利用

X

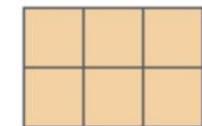


\times

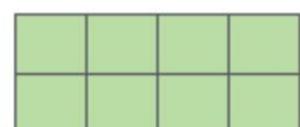
W^K



K



X

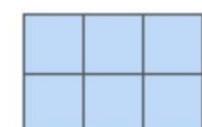


\times

W^V



V



Llamaが使った Rotary position embeddings の数学

論文の内容を引用した

$$f_q(\mathbf{x}_m, m) = (\mathbf{W}_q \mathbf{x}_m) e^{im\theta}$$

$$f_k(\mathbf{x}_n, n) = (\mathbf{W}_k \mathbf{x}_n) e^{in\theta}$$

$$g(\mathbf{x}_m, \mathbf{x}_n, m - n) = \operatorname{Re}[(\mathbf{W}_q \mathbf{x}_m)(\mathbf{W}_k \mathbf{x}_n)^* e^{i(n - m)}]$$

$$f_{\{q,k\}}(\mathbf{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W_{\{q,k\}}^{(11)} & W_{\{q,k\}}^{(12)} \\ W_{\{q,k\}}^{(21)} & W_{\{q,k\}}^{(22)} \end{pmatrix} \begin{pmatrix} \mathbf{x}_m^{(1)} \\ \mathbf{x}_m^{(2)} \end{pmatrix}$$

$$f_{\{q,k\}}(\mathbf{x}_m, m) = \mathbf{R}_{\Theta, m}^d \mathbf{W}_{\{q,k\}} \mathbf{x}_m$$

$$\mathbf{R}_{\Theta, m}^d = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

$$\mathbf{R}_{\Theta, m}^d \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

主な考え方は
rotation matrix
をかけて
位置を変えさせ
る こうするこ
とで相対位置に
変わること

Attention(プログラム)

```
class Attention(nn.Module):
    def __init__(self, args: ModelArgs):
        super().__init__()
        self.n_heads=args.n_heads
        self.head_dim = args.dim // args.n_heads
        self.wq=nn.Linear(args.dim, self.head_dim*args.n_heads)
        self.wk=nn.Linear(args.dim, self.head_dim*args.n_heads)
        self.wv=nn.Linear(args.dim, self.head_dim*args.n_heads)
        self.wo=nn.Linear(self.head_dim*args.n_heads,args.dim)

        self.cache_k = torch.zeros(
            (args.max_batch_size, args.max_seq_len, args.n_heads, self.head_dim)
        ).cuda()
        self.cache_v = torch.zeros(
            (args.max_batch_size, args.max_seq_len, args.n_heads, self.head_dim)
        ).cuda()
```

基本の設置

```
def forward(self, x: torch.Tensor, start_pos: int, freqs_cis: torch.Tensor, mask: Optional[torch.Tensor]):  
    bsz, seqlen, _ = x.shape  
    xq=self.wq(x).view(bsz, -1, self.n_heads, self.head_dim)  
    xk=self.wk(x).view(bsz, -1, self.n_heads, self.head_dim)  
    xv=self.wv(x).view(bsz, -1, self.n_heads, self.head_dim)  
    xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)  
    self.cache_k[:bsz, start_pos : start_pos + seqlen].data = xk  
    self.cache_v[:bsz, start_pos : start_pos + seqlen].data = xv  
    keys = self.cache_k[:bsz, : start_pos + seqlen]  
    values = self.cache_v[:bsz, : start_pos + seqlen]  
    xq = xq.transpose(1, 2)  
    keys = keys.transpose(1, 2)  
    values = values.transpose(1, 2)  
    print(xq.is_cuda)  
    print(keys.is_cuda)  
    print(values.is_cuda)  
    scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
```

Cacheは.dataを付けないと後のfin-tuningにエラーができるgradの計算がうまくいかないのが原因で
しかしこのcacheは元々その計算に参加しないと設置されたはず
今はまだこのところを改善する方法は思い出せない
Apply_rotary_embはrotation計算する関数 次のスライドで話す

```
1 def precompute_freqs_cis(dim: int, end: int, theta: float = 10000.0):
2     freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[:, (dim // 2)].float() / dim))
3     t = torch.arange(end, device=freqs.device) # type: ignore
4     freqs = torch.outer(t, freqs).float() # type: ignore
5     freqs_cis = torch.polar(torch.ones_like(freqs), freqs) # complex64
6     return freqs_cis
7
8
9 def apply_rotary_emb(
10    xq: torch.Tensor,
11    xk: torch.Tensor,
12    freqs_cis: torch.Tensor,
13 ) -> Tuple[torch.Tensor, torch.Tensor]:
14     xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2))
15     xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))
16     #print(freqs_cis.size())
17     #print(xq_.size())
18     #print(xq_.shape[1:])
19     freqs_cis = freqs_cis.view(xq_.shape[1:])
20     xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(3)
21     xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(3)
22     return xq_out.type_as(xq), xk_out.type_as(xk)
```

Precompute_freqs_cisはrotation matrixを生成させる関数

Apply_rotary_embは計算する関数

これはgithubに載ってるコードを少しだけ変えて使った

```
if mask is not None:  
    scores = scores + mask # (bs, n_local_heads, slen, cache_len + slen)  
scores = F.softmax(scores.float(), dim=-1).type_as(xq)  
output = torch.matmul(scores, values) # (bs, n_local_heads, slen, head_dim)  
output = output.transpose(  
    1, 2  
).contiguous().view(bsz, seqlen, -1)  
  
return self.wo(output)
```

attention最後の計算

norm

- Llamaはrms normを使う

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2}. \quad \bar{a}_i = \frac{a_i - \mu}{\sigma} g_i, \quad y_i = f(\bar{a}_i + b_i),$$

```
▶ 1 class RMSNorm(torch.nn.Module):
 2     def __init__(self, dim: int, eps: float = 1e-6):
 3         super().__init__()
 4         self.eps = eps
 5         self.weight = nn.Parameter(torch.ones(dim))
 6
 7     def _norm(self, x):
 8         return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
 9
10    def forward(self, x):
11        output = self._norm(x.float()).type_as(x)
12        return output * self.weight
```

ffn

```
▶ 1 class FeedForward(nn.Module):
2     def __init__(self,
3                  dim: int,
4                  hidden_dim: int,
5                  multiple_of: int,):
6
7         super().__init__()
8         hidden_dim = int(2 * hidden_dim / 3)
9         hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)
10
11        self.w1=nn.Linear(dim,hidden_dim)
12        self.w2=nn.Linear(hidden_dim,dim)
13        self.w3=nn.Linear(dim,hidden_dim)
14
15    def forward(self, x):
16        return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

ここで*で計算するのはkronecker積を得るため
こうすると最後はbatch_,seq_len,dimの出力を得られる

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

Ffnの部
分はそ
のま
使った

block

```
1 class TransformerBlock(nn.Module):
2     def __init__(self, layer_id: int, args: ModelArgs):
3         super().__init__()
4         self.n_heads = args.n_heads
5         self.dim = args.dim
6         self.head_dim = args.dim // args.n_heads
7         self.attention = Attention(args)
8         self.feed_forward = FeedForward(dim=args.dim, hidden_dim=4 * args.dim, multiple_of=args.multiple_of)
9         self.layer_id = layer_id
10        self.attention_norm = RMSNorm(args.dim, eps=args.norm_eps)
11        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)
12
13
14    def forward(self, x: torch.Tensor, start_pos: int, freqs_cis: torch.Tensor, mask: Optional[torch.Tensor]):
15        h = x + self.attention.forward(self.attention_norm(x), start_pos, freqs_cis, mask)
16        out = h + self.feed_forward.forward(self.ffn_norm(h))
17        return out
```

Llamaのgithubに従いこのblockにした

構造はまずadd norm層: normしたxをattentionに渡して本来の入力データを加える その後norm化して Feed forward(前定義したffn)に渡す

Model(設直)

```
1 class Transformer(nn.Module):
2     def __init__(self, params: ModelArgs):
3         super().__init__()
4         self.params = params
5         self.vocab_size = params.vocab_size
6         self.n_layers = params.n_layers
7         self.tok_embeddings=nn.Embedding( params.vocab_size, params.dim)
8         self.layers = torch.nn.ModuleList()
9         for layer_id in range(params.n_layers):
10             self.layers.append(TransformerBlock(layer_id, params))
11         self.norm = RMSNorm(params.dim, eps=params.norm_eps)
12         self.output=nn.Linear(params.dim,params.vocab_size)
13         self.freqs_cis = precompute_freqs_cis(self.params.dim, self.params.max_seq_len * 2)
```

モデルの設定 layerを構築 今回はlayerを3にした(githubでは8)
llamaのembeddingはbertのように複雑ではないこのtokだけでいい
Vocab_sizeは3000(前のtokenizerに設置した語彙数)
Modelのdimは5 1 2
最後出力層を定義する 512から 3000(モデルのdimから語彙数に変える)

言語モデルの原理
は確率分布の近似
出力は語彙数の確
率を作る

Model(forwad)

```
def forward(self, tokens: torch.Tensor, start_pos: int):
    _bsz, seqlen = tokens.shape
    h = self.tok_embeddings(tokens)
    self.freqs_cis = self.freqs_cis.to(h.device)
    freqs_cis = self.freqs_cis[start_pos : start_pos + seqlen]

    mask = None
    if seqlen > 1:
        mask = torch.full((1, 1, seqlen, seqlen), float("-inf"), device=tokens.device)
        mask = torch.triu(mask, diagonal=start_pos + 1).type_as(h)

    for layer in self.layers:
        h = layer(h, start_pos, freqs_cis, mask)
    h = self.norm(h)
    output = self.output(h)
    return output.float()
```

Freqs_cisはattention計算と関連する(回転行列作るため)
embeddingしてlayerに渡して最後は出力

generator(前処理)

```
1 class LLaMA:
2     def __init__(self, model: Transformer, tokenizer: Tokenizer):
3         self.model = model
4         self.tokenizer = tokenizer
5     @torch.no_grad()
6     def generate(
7         self,
8         prompts: List[str],
9         max_gen_len: int,
10        temperature: float = 0.8,
11        top_p: float = 0.95,
12    ) -> List[str]:
13        bsz = len(prompts)
14        params = self.model.params
15        assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)
16        prompt_tokens = [self.tokenizer.encode(x, bos=True, eos=False) for x in prompts]
17        min_prompt_size = min([len(t) for t in prompt_tokens])
18        max_prompt_size = max([len(t) for t in prompt_tokens])
19        total_len = min(params.max_seq_len, max_gen_len + max_prompt_size)
20        tokens = torch.full((bsz, total_len), self.tokenizer.pad_id).cuda().long()
21        for k, t in enumerate(prompt_tokens):
22            tokens[k, :len(t)] = torch.tensor(t).long()
23        input_text_mask = tokens != self.tokenizer.pad_id
```

入力したテキストをトークンしてpaddingする 前のデータの処理と同じ

Generator(計算と生成)

```
start_pos = min_prompt_size
prev_pos = 0
for cur_pos in range(start_pos, total_len):
    logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)[:, -1, :]
    if temperature > 0:
        probs = torch.softmax(logits / temperature, dim=-1)
        next_token = sample_top_p(probs, top_p)
    else:
        next_token = torch.argmax(logits, dim=-1)
    next_token = next_token.reshape(-1)
    # only replace token if prompt has already been generated
    next_token = torch.where(
        input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
    )
    tokens[:, cur_pos] = next_token
    prev_pos = cur_pos

def sample_top_p(probs, p):
    probs_sort, probs_idx = torch.sort(probs, dim=-1, descending=True)
    probs_sum = torch.cumsum(probs_sort, dim=-1)
    mask = probs_sum - probs_sort > p
    probs_sort[mask] = 0.0
    probs_sort.div_(probs_sort.sum(dim=-1, keepdim=True))
    next_token = torch.multinomial(probs_sort, num_samples=1)
    next_token = torch.gather(probs_idx, -1, next_token)
```

まずモデルで確率分布計算する出力はbatch,seq_len,vocab_size最後は最後の1行(seq_len)を選ぶ出力はbatch,vocab_sizeになる

Temperatureは0以上の場合はtop_pで計算

一回目は最小の0からprompt_sizeまで
2回目からは前の位置から一つ後まで

事前学習モデル中身

▶ model.to(device)

```
⌚ Transformer(
    (tok_embeddings): Embedding(3000, 512)
    (layers): ModuleList(
        (0-2): 3 x TransformerBlock(
            (attention): Attention(
                (wq): Linear(in_features=512, out_features=512, bias=True)
                (wk): Linear(in_features=512, out_features=512, bias=True)
                (wv): Linear(in_features=512, out_features=512, bias=True)
                (wo): Linear(in_features=512, out_features=512, bias=True)
            )
            (feed_forward): FeedForward(
                (w1): Linear(in_features=512, out_features=1536, bias=True)
                (w2): Linear(in_features=1536, out_features=512, bias=True)
                (w3): Linear(in_features=512, out_features=1536, bias=True)
            )
            (attention_norm): RMSNorm()
            (ffn_norm): RMSNorm()
        )
    )
    (norm): RMSNorm()
    (output): Linear(in_features=512, out_features=3000, bias=True)
)
```

事前学習(pre-training)

- 事前学習はbertのやり方を参考した 自教師学習のmask lmを使った 入力データの一部をランダムを選んで(15%の確率で)mask tokenにしその中の一部のトークンをランダムで変える他のはそのままにする

```
▶ 1 labels=torch.full((bsz, total_len), tokenizer.pad_id)
  2 rand = torch.rand(tokens.shape)
  3 mask_arr = rand < 0.15
```

```
▶ 1 for k,t in enumerate(labels):
  2     if random()>=0.1:
  3         labels[k][torch.where(mask_arr[k] == 1)] =tokens[k][torch.where(mask_arr[k] == 1)]
  4     else:
  5         labels[k][torch.where(mask_arr[k] == 1)] =
  6         torch.tensor([randint(0,2999) for i in range(len(labels[k][torch.where(mask_arr[k] == 1)]))])
```

事前学習loop

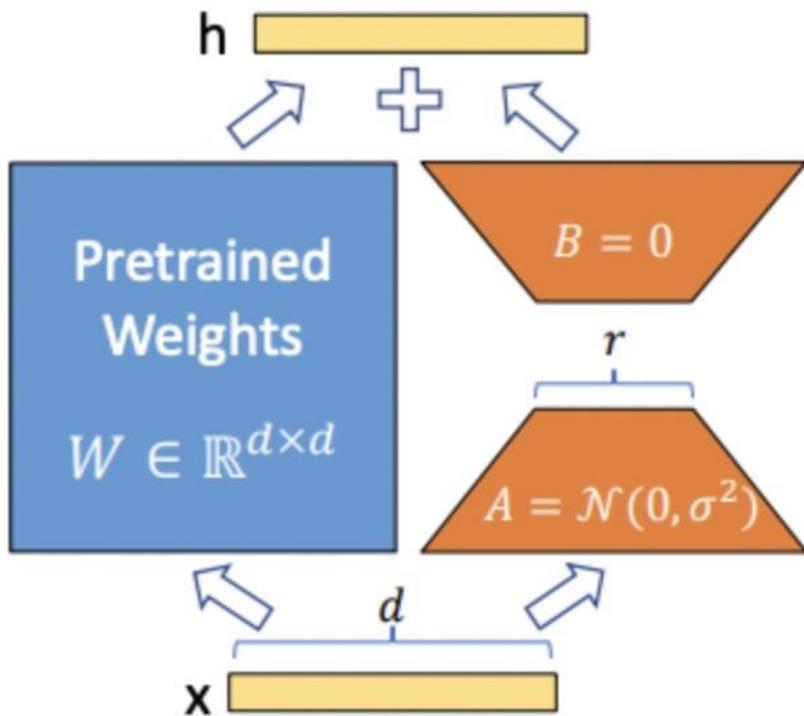
```
[ ] 1 dataset = torch.utils.data.TensorDataset(tokens1[320:380], labels1[320:380])
2 Loader = torch.utils.data.DataLoader(dataset,
3                                         batch_size=5,
4                                         shuffle=True,
5                                         )
```

```
1 model.train()
2
3 for i, data in enumerate(Loader, 0):
4     print(i)
5     inputs, labels = data[0].to('cuda:0'), data[1].to('cuda:0')
6
7     outputs = model(inputs, 0)
8     loss = criterion(outputs.transpose(1, 2), labels)
9     optimizer.zero_grad()
10    loss.backward(retain_graph=True)
11    optimizer.step()
12
13    del inputs, labels, outputs, loss
14    torch.cuda.empty_cache()
15
16
17 print('Finished Training')
```

Epochがない原因はcolabのgpuの計算容量

これだけで使い切るから
実際の訓練ではランタイムを再起動してバッチを変えたり一つのデータセットを複数回で訓練する

Fine-tuningモデル中身



```
(wq): Linear(  
    in_features=512, out_features=512, bias=True  
(lora_dropout): ModuleDict(  
    (default): Dropout(p=0.05, inplace=False)  
)  
(lora_A): ModuleDict(  
    (default): Linear(in_features=512, out_features=8, bias=False)  
)  
(lora_B): ModuleDict(  
    (default): Linear(in_features=8, out_features=512, bias=False)  
)  
(lora_embedding_A): ParameterDict()  
(lora_embedding_B): ParameterDict()  
)
```

Loraを入れた一部

Fine-tuning

- Fine-tuing用データを先に話したものとその返事二つに分ける
その中の単語と符号の数を 100 以下のものを選んでトークン化する

- Fin-tuningはloraをする（
事前学習したモデルのパラメータでloraを入れる

```
[ ] 1 model=torch.load('model_pre.p
```

```
[ ] 1 model.to('cuda:0')
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.AdamW(model
```

```
[ ] 1 for param in model.parameters():
2     param.requires_grad = False # freeze the model - train adapters later
3     if param.ndim == 1:
4         # cast the small parameters (e.g. layernorm) to fp32 for stability
5         param.data = param.data.to(torch.float32)
```

```
[137] 1 from peft import LoraConfig, get_peft_model
      2 config = LoraConfig(
      3     r=8,
      4     lora_alpha=16,
      5     target_modules=["wq", "wk", "wv", "wo"],
      6     lora_dropout=0.05,
      7     bias="none",
      8     #task_type="CAUSAL_LM"
      9
     10 model = get_peft_model(model, config)
```

Fine-tuning

- Hugging face のライブラリーで事前学習モデルのattentionをloraを入れる 入力データは先に話した内容 labelはその返事にしてトレーニング(input maskなどはしなかった)

```
[137] 1 from peft import LoraConfig, get_peft_model
```

```
▶ 1 config = LoraConfig(  
2     r=8,  
3     lora_alpha=16,  
4     target_modules=["wq", "wk", "wv", "wo"],  
5     lora_dropout=0.05,  
6     bias="none",  
7     #task_type="CAUSAL_LM"  
8 )  
9  
10 model = get_peft_model(model, config)
```

```
▶ 1 dataset = torch.utils.data.TensorDataset(tokens1[0:1280], tokens2[0:1280])  
2 Loader = torch.utils.data.DataLoader(dataset,  
3                                         batch_size=32,  
4                                         shuffle=True,  
5                                         )
```

Fine-tuningのパラメータについて



```
1 def print_trainable_parameters(model):
2     """
3     Prints the number of trainable parameters in the model.
4     """
5     trainable_params = 0
6     all_param = 0
7     for _, param in model.named_parameters():
8         all_param += param.numel()
9         if param.requires_grad:
10             trainable_params += param.numel()
11     print(
12         f"trainable params: {trainable_params} || all params: {all_param} ||"
13         f"trainable%: {100 * trainable_params / all_param}"
14     )
```

Hugging faceの
チュートリアル
で書いた関数を
使ってどのぐら
いのパラメータ
をトーレイング
するのかを見る

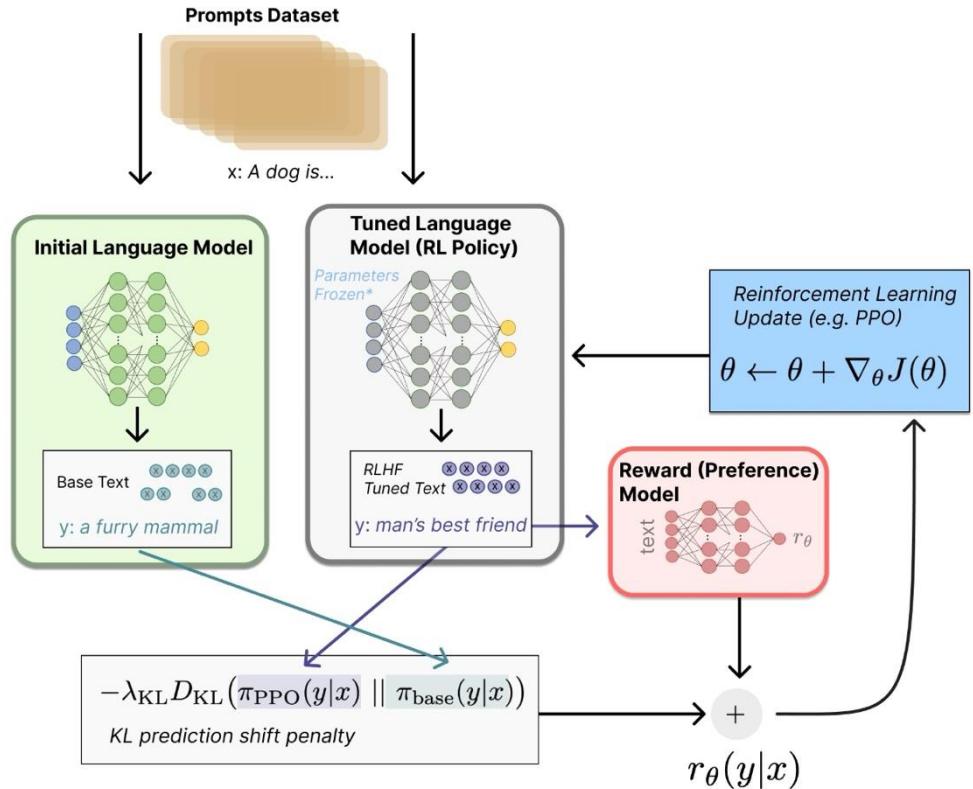
```
[142] 1 print_trainable_parameters(model)
```

```
trainable params: 98304 || all params: 13417400 || trainable%: 0.7326605750741574
```

Fine-tuning loop

```
[ ] 1 for i, data in enumerate(Loader, 0):
2     print(i)
3     inputs, labels = data[0].to('cuda:0'), data[1].to('cuda:0')
4
5     outputs = model(inputs, 0)
6     loss = criterion(outputs.transpose(1, 2), labels)
7     optimizer.zero_grad()
8     loss.backward(retain_graph=True)
9     optimizer.step()
10
11    del inputs, labels, outputs, loss
12    torch.cuda.empty_cache()
```

強化学習 reinforcement learning 構造



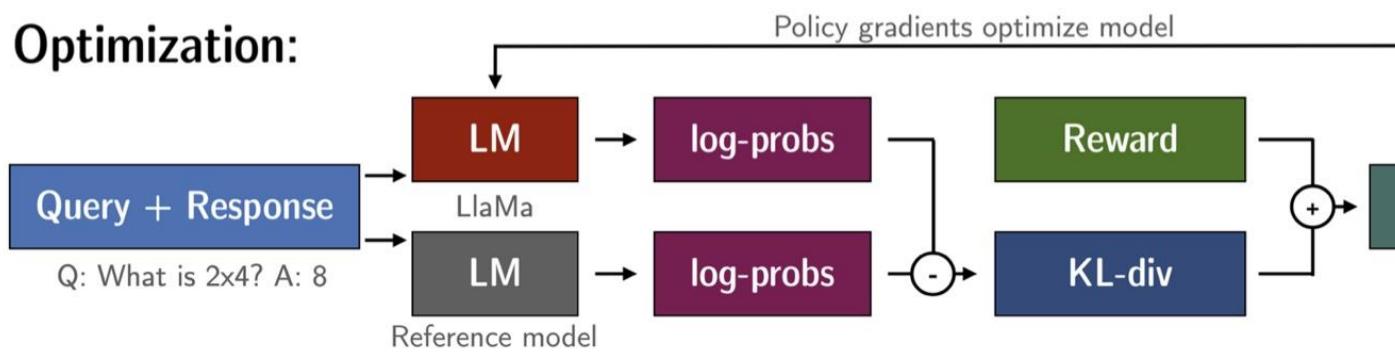
Rollout:



Evaluation:



Optimization:



Hugging faceが書いたトレーニング構造 これを主な参考とする
 最適化はppo (ppoは複雑なので今回は説明をskipする)
 Actor-critic構造を使う actor critic reward 三つのモデルを定義する
 Ppoのkl-divはここでペナルティとして使う

Helper function

```
[ ] def g_token(res):
    seq_token=[tokenizer.encode(x, bos=True, eos=True) for x in res]
    max_batch_size: int = 32
    max_seq_len: int = 2048
    min_prompt_size = min([len(t) for t in seq_token])
    max_prompt_size = max([len(t) for t in seq_token])
    total_len = min(max_seq_len, 20 + max_prompt_size)

    bsz=len(seq_token)

    seq_tokens=torch.full((bsz, total_len),tokenizer.pad_id)
    for k, t in enumerate(seq_token):
        seq_tokens[k, : len(t)] = torch.tensor(t)
    return seq_tokens

[ ] def Advantages(reward,values):
    lastgaelam=0
    advantages = torch.zeros_like(reward).to(device)
    for t in reversed(range(len(reward))):
        if t == len(reward) - 1:
            nextvalues = values[t]
        else:
            nextvalues = values[t + 1]
        delta = reward[t] + gamma * nextvalues - values[t]
        advantages[t] = lastgaelam = delta + gamma * gae_lambda * lastgaelam
```

Tokenを生成する関数とppoに必要なadvantages関数を定義した

強化学習(reinforcement learning)

- ppoを使う
- この言語モデルのトレーニングにおいて stateは入力データ action は生成したデータ [state action]を作つてモデルに渡す
- Actorモデルはfine-tuningしたモデル
- Criticモデルはdeep copy(actor) それから線形層に渡す
- Reward modelはモデル推論の出力を入つてrewardを返す関数にする

```
class Critic(nn.Module):
    def __init__(self,model):
        super().__init__()
        self.critic=model
        self.value_head = nn.Sequential(
            nn.Linear(3000, 1))

    def forward(self,x,mask = None,start_pos=None):
        critic_embeds = self.critic(x,start_pos)
        out=critic_embeds[:, -1, :].clone()
        values = self.value_head(out)
        return values
```

```
class RewardModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.net=nn.Sequential(nn.Linear(vocab_size,vocab_size//2 , bias = False),
                              nn.Linear(vocab_size//2,1 , bias = False))

    def forward(self,x):
        return self.net(x)
```

Criticの出力はスカラーにする rewardはベクトルにする

Reward modelのトレーニング

```
1 model=Transformer(ModelArgs)
2 model.to('cuda:0')
3 ge=LLaMA(model,tokenizer)
4 rewards=RewardModel()
5 rewards.to('cuda:0')
```

```
[ ] 1 st = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
2
[ ] 1 reward_loss= nn.CrossEntropyLoss()
2 reward_opt=optim.Adam(rewards.parameters(), lr=0.001)
```

```
1 model.eval()
2 rewards.train()
3 episodes=5
4 epoch=10
5
6 for _ in range(episodes):
7     sample=[randint(0, 28444) for i in range(32)]
8     input=[sentence1[i] for i in sample]
9     label=[sentence2[i] for i in sample]
10    res=ge.generate(input,20,0.5)
11    output=[res[i][len(input[i]):] for i in range(32)]
12    ebd1=st.encode(output, convert_to_tensor=True)
13    ebd2=st.encode(label, convert_to_tensor=True)
14    score=torch.diag(util.pytorch_cos_sim(ebd1, ebd2)).data*torch.tensor([10]).to(device)
15    seq=g_token(res)
16    a_logi=model(seq.to(device),0)
17    for j in range(epoch):
18        reward=rewards(a_logi)
19        re_loss=reward_loss(reward.squeeze(-1),score.long())
20
21        reward_opt.zero_grad()
22        re_loss.backward(retain_graph=True)
23        reward_opt.step()
```

Sentence transformerでトークン化して生成した返事データと元の返事類似度を算出する(調べたところsentenceの間はよくcos類似度を使う) 類似度をかける10にする(rewardを0から10以内設置したい)

元々ここは人間で採点してrewardを設置すべきが
今回は類似度で入れ替える 主な原因は設置方法がわからない

トレーニング前の設置

```
[ ] 1 actor=model
2 actor.to('cuda:0')
3 actor_optimizer = optim.Adam(actor.parameters(), lr=0.001)
4
5
6 critic=Critic(copy.deepcopy(actor))
7 critic.to('cuda:0')
8 critic_optimizer=optim.Adam(critic.parameters(), lr=0.001)
9
10 rewards=RewardModel()
11 rewards.to('cuda:0')
12 rewards.load_state_dict(torch.load('reward_model_weight.pth'))
13 ge=LLaMA(actor,tokenizer)
```

```
[ ] 1 for param in model.parameters():
2     param.requires_grad = False # freeze the model - train adapters later
3     if param.ndim == 1:
4         # cast the small parameters (e.g. layernorm) to fp32 for stability
5         param.data = param.data.to(torch.float32)
```

```
[ ] 1 print_trainable_parameters(model)
```

trainable params: 28096 || all params: 13445496 || trainable%: 0.20896216844659357

```
[ ] 1 config = LoraConfig(
2     r=8,
3     lora_alpha=16,
4     target_modules=["output"],
5     lora_dropout=0.05,
6     bias="none",
7     #task_type="CAUSAL_LM"
8 )
```

```
[ ] 1 eps_clip=0.2
2 value_clip=0.4
3 beta_s=0.01
4 gamma=0.98
5 gae_lambda=0.94
```

強化学習パラメーターを設置する
モデルのパラメーターをfreeze

モデルの最後の出力層にloraを入れる

強化学習トレーニング

```
1 actor.train()
2 critic.train()
3 rewards.eval()
4 for _ in range(episodes):
5     prompts=sample(sentence1, 32)
6     res=ge.generate(prompts,20,0.5)
7     state_action=g_token(res).to(device)
8
9     for j in range(ecpoch):
10         seq_tokes=state_action
11         action_logi=actor(seq_tokes,0)
12         reward=rewards(action_logi)
13         value=critic(seq_tokes,start_pos=0)
14         advantages=Advantages(reward,value)
15         old_action_probs=action_logi.softmax(dim=-1)[0]
16         old_log_probs=(action_logi.softmax(-1).gather(-1,seq_tokes[...,None].to(device)).squeeze(-1).log())[0]
17         old_values=value[0]
18         for i in range(len(prompts)):
```

毎回のepisodeで32個データをサンプルして生成させてトークンにする

ecpochで同じstate_actionトークンをlogi,reward,value,adavantagesを計算して最初のデータをこれからの過去の行動

確率、valuesにする これは次からのloopでバッチの順番で一つ一つ計算して更新していくからだ

強化学習トレーニング

```
for i in range(len(prompts)):  
    seq=seq_tokes[i].view(1,-1)  
  
    action_logi=actor(seq,0)  
    action_prob=action_logi.softmax(dim=-1)  
    values=critic(seq,start_pos=0)  
    a_log_prob=(action_prob.gather(-1,seq[...,None].to(device)).squeeze(-1).log())  
    entropies = (action_prob* action_prob.log()).sum(dim = -1)  
    kl_divs=(action_prob*(action_prob.log()-old_action_probs.log())).sum(dim=-1)  
    reward=rewards(action_logi)  
    reward=reward-kl_divs  
    ada=reward - old_values  
    ratios = (a_log_prob - old_log_probs).exp()  
  
    surr1 = ratios * (advantages[i]+ada)  
    surr2 = ratios.clamp(1-eps_clip, 1 + eps_clip) * (advantages[i]+ada)  
    policy_loss = - torch.min(surr1, surr2) - beta_s* entropies  
    loss= policy_loss.mean()  
    loss.backward(retain_graph=True)  
  
actor_optimizer.step()  
actor_optimizer.zero_grad()
```

```
value_clipped=old_values+ (values - old_values).clamp(-value_clip, value_clip)  
value_loss_1=(value_clipped.flatten() - reward) ** 2  
value_loss_2 = (values.flatten() - reward) ** 2  
value_loss = torch.mean(torch.max(value_loss_1, value_loss_2)).mean()
```

Lossに必要なものを計算する
Adaは現在のパラメーターで現在のadavantagesを計算し
前のloopで事前に計算したadavantages足す 一つのデータに一度更新するため
2回目からこのadaは新しいパラメータに影響されるはず

まずはpolicy(actorのlossを計算して更新するあとはvalue(critic)のlossを計算更新する

このやり方は言語モデルのトレーニングのサンプルコードと普通のppoのコードを参考して作った

最後

- ・今日は大雑把でモデルを作ってトレーニングした これからはもう少し細かくする トレーニングの方法もいくつ試してみる
- ・今回なぜできなかったのを後で調べてみる
- ・強化学習の部分に問題が出たかもしれない
- ・データのトレーニングにも問題あるかもしれない
- ・これからは 詳しく資料を調べてみる

参考 :

ウェブページ

- <https://nlp.seas.harvard.edu/annotated-transformer/>
- <https://neptune.ai/blog/how-to-code-bert-using-pytorch-tutorial>
- <https://github.com/huggingface/transformers/tree/main/src/transformers/models/llama>
- <https://github.com/facebookresearch/llama/tree/57b0eb62de0636e75af471e49e2f1862d908d9d8/llama>
- <https://colab.research.google.com/drive/1iERDk94Jp0UErsPf7vXyPKeiM4ZJUQ-a?usp=sharing#scrollTo=Cc8354XxCIWI>
- https://pytorch.org/tutorials/beginner/chatbot_tutorial.html#preparations
- <https://github.com/lucidrains/PaLM-rlhf-pytorch/tree/main>
- <https://huggingface.co/blog/rlhf>
- <https://huggingface.co/learn/deep-rl-course/unit8/hands-on-cleanrl#lets-code-ppo-from-scratch-with-costa-huang-tutorial>

論文

- <https://arxiv.org/pdf/2104.09864.pdf>
- <https://arxiv.org/pdf/2104.09864v4.pdf>
- <https://arxiv.org/pdf/2302.13971.pdf>
- <https://arxiv.org/pdf/1706.03762.pdf>