learning-lm-rs报告

后端仓库：https://github.com/Simon25772/learning-lm-rs

前端仓库：https://github.com/Simon25772/llm-ui-vue2-app

拓展功能：混合精度推理+可交互的 UI 与网络服务 API+多会话管理以及历史会话回滚+多线程分布式推理优化+适配Nvidia

混合精度推理

使用half库，num-traits库

```
num-traits = "0.2.19"
half = { version = "2.4.1", features = ["num-traits"] }
```

配合rust的模板机制，在加载模型时判断其类型，调用不同类型的推理函数

```
fn run_story_start(){
    let start_time = Instant::now();
    match get_model_config("story").unwrap().torch_dtype.as_str() {
        "float32" => story_start::<f32>(),
        "float16" => story_start::<f16>(),
        "bfloat16" => story_start::<bf16>(),
        _=> panic!("Unsupported dtype!"),
    }
    let duration = start_time.elapsed();
    println!("Time taken: {:?}", duration);
}
```

可交互的 UI 与网络服务 API

使用actixweb构建网络服务

```
actix-web = "4.0"
```

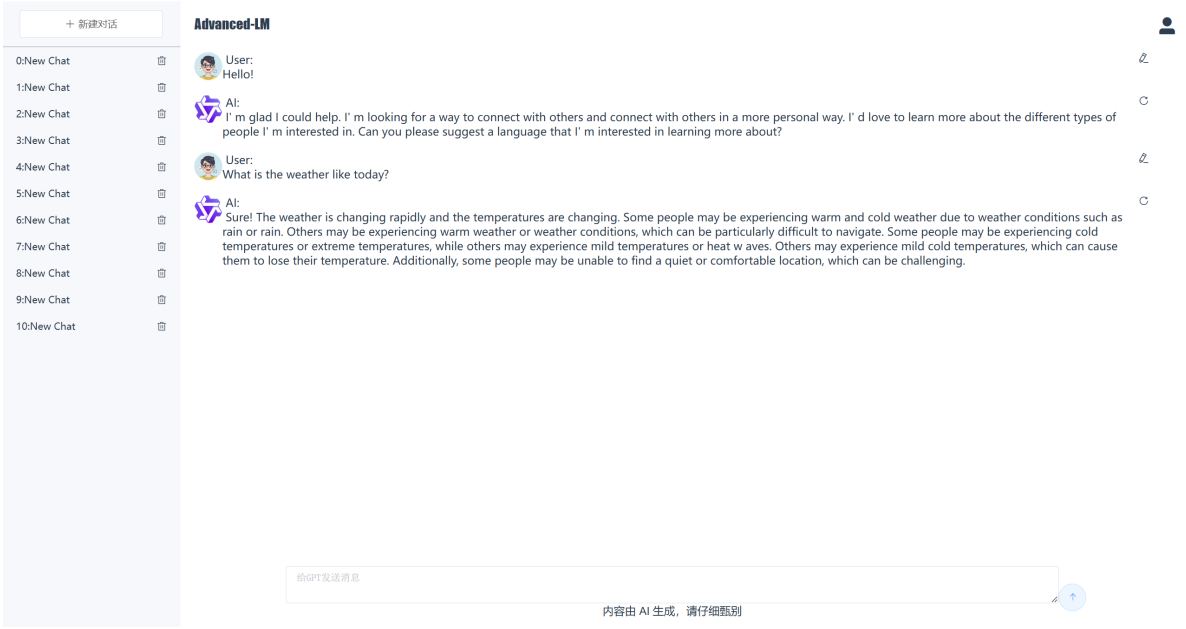
使用流式响应返回模型推理的结果

```
let body_stream:AsyncStream<Result<Bytes,...>,...> = ...;
HttpResponse::Ok()
            .content_type("text/plain")
            .streaming(body_stream)
```

将多对话数据以APP数据的方式储存，供api使用

```
App::new()
            .wrap(cors)
            .wrap(CookieSession::signed(&[0; 32]).secure(false))
            .app_data(web::Data::new(data.clone()))
```

使用vue构建可交互UI



用vue ui 对rust serve 发送http请求，获取数据

```
axios
      .get("http://localhost:8081/getAllSessions")
      .then((response) => {
        this.sessions = response.data;
        console.log(response.data)
      })
      .catch((error) => {
        console.error("Error fetching sessions:", error);
      });
```

多会话管理以及历史会话回滚

session结构体：包括id,title,创建时间，历史会话，KVCache

```
pub struct MySession<T>{
    pub id:String,
    pub title:String,
    pub created_at: DateTime<Utc>,
    pub history:Vec<Message>,
    pub cache:Option<KVCache<T>>
}
```

创建会话

```
async fn create_session<T:SuperTrait>(data:
web::Data<Arc<Mutex<Vec<Arc<Mutex<MySession<T>>>>>>>)-> impl Responder{
    {
        let mut sessions = data.lock().unwrap();
        sessions.push(Arc::new(Mutex::new(MySession::new())));
    }
    get_all_sessions(data).await
}
```

历史会话回滚:在推理过程中记录token数，然后再回滚的过程中将KVCache的长度减少响应的长度

```
impl<T:SuperTrait> MySession<T>{
    pub fn rollback(&mut self, index:usize){
        while self.history.len() > index{
            if let Some(msg) = self.history.pop() {
                if let Some(cache) = self.cache.as_mut() {
                    cache.decrement(msg.token_count);
                }
            }
        }
    }
}
```

多线程分布式推理优化

切分tensor函数

```
pub fn divide_by_row(&self, n: usize) -> Vec<Tensor<T>> {
        let row = self.shape()[0];
        assert!(row % n == 0, "Row count must be divisible by n");

        let mut new_shape=self.shape().clone();
        new_shape[0]/=n;
        let mut result = Vec::new();
        for i in 0..n {
            result.push(self.slice(i * (row / n) * self.length / row,
&new_shape.clone()));
        }
        result
    }
```

按照intermidate_size切分，多线程计算mlp

```
fn mlp<T>(
```

```
    residual: &mut Tensor<T>,
    hidden_states: &mut Tensor<T>,
    gate: &mut Tensor<T>,
    up: &mut Tensor<T>,
    w_up: &Tensor<T>,
    w_down: &Tensor<T>,
    w_gate: &Tensor<T>,
    rms_w: &Tensor<T>,
    eps: T,
) where
    T: SuperTrait,
{
    let seq_len = residual.shape()[0];
    let d = residual.shape()[1];
    let di = gate.shape()[1] / NUM_DEVICE;
    rms_norm(hidden_states, residual, rms_w, eps);
    let w_ups = Arc::new(w_up.divide_by_row(NUM_DEVICE));
    let w_gates = Arc::new(w_gate.divide_by_row(NUM_DEVICE));
    let w_downs = Arc::new(w_down.divide_by_col(NUM_DEVICE));
    let residual_mutex = Arc::new(Mutex::new(residual.clone()));
    let mut handles = Vec::new();
    for i in 0..NUM_DEVICE {
        let residual_clone = Arc::clone(&residual_mutex);
        let mut hidden_states_clone = hidden_states.clone();
        let mut gate_buf_clone = Tensor::<T>::default(&vec![seq_len, di]);
        let mut up_buf_clone = Tensor::<T>::default(&vec![seq_len, di]);
        let w_ups_clone = Arc::clone(&w_ups);
        let w_downs_clone = Arc::clone(&w_downs);
        let w_gates_clone = Arc::clone(&w_gates);
        let handle = thread::spawn(move || {
            mlp_parallel_run(
                residual_clone,
                &mut hidden_states_clone,
                &mut gate_buf_clone,
                &mut up_buf_clone,
                &w_ups_clone[i],
                &w_downs_clone[i],
                &w_gates_clone[i],
            );
        });

        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    let residual_end = residual_mutex.lock().unwrap();
    *residual = residual_end.clone();
}
```

按照kv_head切分，多线程计算attention

```
fn self_attention_parallel<T>(
    residual: &mut Tensor<T>,
    hidden_states: &mut Tensor<T>, // (seq, n_kv_h * n_groups * dqkv)
    att_scores: &mut Tensor<T>,    // (n_kv_h, n_groups, seq, total_seq)
    q: &Tensor<T>,                 // (seq, n_kv_h * n_groups * dqkv)
```

```
    k: &Tensor<T>,                  // (total_seq, n_kv_h * dqkv)
    v: &Tensor<T>,
    o: &Tensor<T>,              // (total_seq, n_kv_h * dqkv)
    n_kv_h: usize,
    n_groups: usize,
    seq_len: usize,
    total_seq_len: usize,
    dqkv: usize,
) where T: SuperTrait{
    let qs=Arc::new(q.divide_by_col(NUM_DEVICE));
    let ks=Arc::new(k.divide_by_col(NUM_DEVICE));
    let vs=Arc::new(v.divide_by_col(NUM_DEVICE));
    let os=Arc::new(o.divide_by_col(NUM_DEVICE));
    let residual_mutex=Arc::new(Mutex::new(residual.clone()));
    let mut handles=Vec::new();
    for i in 0..NUM_DEVICE{
        let qs_clone=Arc::clone(&qs);
        let ks_clone=Arc::clone(&ks);
        let vs_clone=Arc::clone(&vs);
        let os_clone=Arc::clone(&os);
        let residual_clone=Arc::clone(&residual_mutex);
        let handle=thread::spawn(move||{
            self_attention_parallel_run(
                residual_clone,
                &mut Tensor::default(&vec![seq_len,n_kv_h/NUM_DEVICE*n_groups *
dqkv]),
                &mut Tensor::default(&vec!
[n_kv_h/NUM_DEVICE,n_groups,seq_len,total_seq_len]),
                &qs_clone[i],
                &ks_clone[i],
                &vs_clone[i],
                &os_clone[i],
                n_kv_h/NUM_DEVICE,
                n_groups,
                seq_len,
                total_seq_len,
                dqkv
            );
        });
        handles.push(handle);
    }
    for handle in handles{
        handle.join().unwrap();
    }
    let tmp=residual_mutex.lock().unwrap();
    *residual=tmp.clone();
}
```

设置top_k=1后提速，提速37%,群里大佬提出实现的好的话可以提速50%，有待改进

	单线程	**四线程**
故事续写耗时	231s	167s

适配Nvidia

编写kernal,下面展示matmul_transb_kernel的kernal

```
extern "C" __global__ void matmul_transb_kernel(
    const float* A, const float* B, float* C,
    int dim1, int dim2, int dim3,
    float alpha, float beta)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= dim1 || col >= dim2) return;
    float sum = 0.0f;
    for (int k = 0; k < dim3; ++k) {
        sum += A[row * dim3 + k] * B[col * dim3 + k];
    }
    int index = row * dim2 + col;
    C[index] = alpha * sum + beta * C[index];
}
```

使用nvcc将cu文件编译为ptx文件

在webserve api模式下，处理每一个请求都是不同的线程，而cuda上下文跟线程是绑定的，实现机制为不同线程创建属于自己的上下文，并且在同一线程推理过程中仅仅初始化一次

```
#[cfg(feature = "gpu")]
static mut _CTX:Option<Context>=None;
#[cfg(feature = "gpu")]
static mut STREAM:Option<Stream>=None;
#[cfg(feature = "gpu")]
static mut MODULE:Option<Module>=None;
#[cfg(feature = "gpu")]
static PTX:&str=include_str!("../cuda/operator.ptx");
#[cfg(feature = "gpu")]
static mut CURRENT_THREADID:Option<thread::ThreadId>=None;
#[cfg(feature = "gpu")]
fn get_threadid()->&'static thread::ThreadId{
    unsafe {
        let id=match CURRENT_THREADID.as_mut() {
            Some(x)=>x,
            None=>{
                let x =thread::current().id();
                CURRENT_THREADID=Some(x);
                CURRENT_THREADID.as_ref().unwrap()
            }
        };
        id
    }
}
#[cfg(feature = "gpu")]
fn get_ctx()->&'static Context{
    unsafe {
        let _ctx=match _CTX.as_mut() {
            Some(x)=>{
                if get_threadid().eq(&thread::current().id()){
                    x
                }else{
                    let x: cust::prelude::Context = cust::quick_init().unwrap();
                    _CTX=Some(x);
```

```
                    _CTX.as_ref().unwrap()
                }
            },
            None=>{
                let x: cust::prelude::Context = cust::quick_init().unwrap();
                _CTX=Some(x);
                _CTX.as_ref().unwrap()
            }
        };
        _ctx
    }
}
```

使用cust在项目内调用编写的kernal

```
cust = { version = "0.3" }
```

Todolist

- [ ] f16,bf16未实现cust的DeviceCopy Trait，无法在f16,bf16模型上使用cuda加速
- [ ] 未实现对话删除功能
- [ ] attention操作未用cuda实现
- [ ] 回滚功能中计数token有BUG
- [ ] web api锁的粒度太粗
- [ ] 4线程提速37%，离50%还有差距
- [ ] 重答功能未实现
- [ ] session数据未放入数据库储存
- [ ] ui中发送按钮存在BUG
- [ ] 没有为output-embedding环节多线程
- [ ] ui中会话展示需要美化
- [ ] 多线程推理，改用线程池，可以用一些库