1. Use trillions of tokens, only publicly available data (no proprietary or inaccessible datasets)
2. Llama-13B outperforms GPT3(175B)

Background:

1. A relatively small model trained on more data maybe better performed than the larger model.
2. A smaller one trained longer (more training cost) but ultimately cheaper at inference.

Goal:

Decoder-ONLY

TO train a series of language models that achieve the best possible per-formance at various inference budgets, by training on more tokens than what is typically used

特点：
仅用公有数据，可以用小模型打败大模型，或者跟相同量级的state of art 持平。

Approach：
Dataset：(粗体是与其他NLP训练所用不同的数据库)

1. only public data, no duplication, quality filtering with heuristics.
2. **GitHub dataset(BSD and MIT licensed)** , quality filtering, and unduplicated.
3. **20 languages (Wikipedia)** and arXiv scientific data
4. Question and answer dataset （**Stack exchange 2%**）

Tokenizer：
pair encoding (BPE) algorithm (Sennrich et al., 2015), using the implementation from Sentence-Piece (Kudo and Richardson, 2018).

Architecture

1. Pre-normalization [GPT3] : normalize the input of each transformer sub-layer, instead of normalizing the output. (RMSNorm)

```
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def _norm(self, x):
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x):
        output = self._norm(x.float()).type_as(x)
        return output * self.weight
```

2. SwiGLU activation function [PaLM].

$$\text{SwiGLU}(x, W, V, b, c, \beta) = \text{Swish}_\beta(xW + b) \otimes (xV + c)$$

```
class FeedForward(nn.Module):
    def __init__(
        self,
        dim: int,
        hidden_dim: int,
        multiple_of: int,
    ):
        super().__init__()
        hidden_dim = int(2 * hidden_dim / 3)
        hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)

        self.w1 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )
        self.w2 = RowParallelLinear(
            hidden_dim, dim, bias=False, input_is_parallel=True, init_method=lambda x: x
        )
        self.w3 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )

    def forward(self, x):
        return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

3. Rotary Embeddings [GPTNeo]. (At each layer)

加性变乘性

```

def precompute_freqs_cis(dim: int, end: int, theta: float = 10000.0):
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[: (dim // 2)].float() / dim))
    t = torch.arange(end, device=freqs.device)  # type: ignore
    freqs = torch.outer(t, freqs).float()  # type: ignore
    freqs_cis = torch.polar(torch.ones_like(freqs), freqs)  # complex64
    return freqs_cis


def reshape_for_broadcast(freqs_cis: torch.Tensor, x: torch.Tensor):
    ndim = x.ndim
    assert 0 <= 1 < ndim
    assert freqs_cis.shape == (x.shape[1], x.shape[-1])
    shape = [d if i == 1 or i == ndim - 1 else 1 for i, d in enumerate(x.shape)]
    return freqs_cis.view(*shape)


def apply_rotary_emb(
    xq: torch.Tensor,
    xk: torch.Tensor,
    freqs_cis: torch.Tensor,
) -> Tuple[torch.Tensor, torch.Tensor]:
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2))
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))
    freqs_cis = reshape_for_broadcast(freqs_cis, xq_)
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(3)
    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(3)
    return xq_out.type_as(xq), xk_out.type_as(xk)
```

https://arxiv.org/abs/2302.13971

Attention mask generation （Masked-self Attention）:

```
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)
```

> **torch.triu**(*input*, *diagonal=0*, *, *out=None*) → Tensor

Returns the upper triangular part of a matrix (2-D tensor) or batch of matrices `input`, the other elements of the result tensor `out` are set to 0.

```
>>> a = torch.randn(3, 3)
>>> a
tensor([[ 0.2309,  0.5207,  2.0049],
        [ 0.2072, -1.0680,  0.6602],
        [ 0.3480, -0.5211, -0.4573]])
>>> torch.triu(a)
tensor([[ 0.2309,  0.5207,  2.0049],
        [ 0.0000, -1.0680,  0.6602],
        [ 0.0000,  0.0000, -0.4573]])
>>> torch.triu(a, diagonal=1)
tensor([[ 0.0000,  0.5207,  2.0049],
        [ 0.0000,  0.0000,  0.6602],
        [ 0.0000,  0.0000,  0.0000]])
>>> torch.triu(a, diagonal=-1)
tensor([[ 0.2309,  0.5207,  2.0049],
        [ 0.2072, -1.0680,  0.6602],
        [ 0.0000, -0.5211, -0.4573]])

>>> b = torch.randn(4, 6)
>>> b
tensor([[ 0.5876, -0.0794, -1.8373,  0.6654,  0.2604,  1.5235],
        [-0.2447,  0.9556, -1.2919,  1.3378, -0.1768, -1.0857],
        [ 0.4333,  0.3146,  0.6576, -1.0432,  0.9348, -0.4410],
        [-0.9888,  1.0679, -1.3337, -1.6556,  0.4798,  0.2830]])
>>> torch.triu(b, diagonal=1)
tensor([[ 0.0000, -0.0794, -1.8373,  0.6654,  0.2604,  1.5235],
        [ 0.0000,  0.0000, -1.2919,  1.3378, -0.1768, -1.0857],
        [ 0.0000,  0.0000,  0.0000, -1.0432,  0.9348, -0.4410],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.4798,  0.2830]])
>>> torch.triu(b, diagonal=-1)
tensor([[ 0.5876, -0.0794, -1.8373,  0.6654,  0.2604,  1.5235],
        [-0.2447,  0.9556, -1.2919,  1.3378, -0.1768, -1.0857],
        [ 0.0000,  0.3146,  0.6576, -1.0432,  0.9348, -0.4410],
        [ 0.0000,  0.0000, -1.3337, -1.6556,  0.4798,  0.2830]])
```

LLAMA 中没有cross-attention，都是masked self-attention