详解DNN下的tensorflow优化模块设计

项目将结合***Tensorflow***的特点,详述优化模块设计的关键难点.

Requirements

- Tensorflow $\geq$ 1.3
- Python $\geq$ 3.5

Idea

What is *vec*?

一个按照定义顺序出现的张量集合被称为是***vec***.

- 例如demo中的网络由2个卷积层(卷积核为K_1, K_2)+一个全连接层(权值矩阵为W)组成,那么
$$[K_1, K_2, W] = tf.trainable\_variables()$$
就是一个***vec***.
- 梯度也可以是一个***vec***:
$$\left\{\frac{\partial L}{\partial K_1}, \frac{\partial L}{\partial K_2}, \frac{\partial L}{\partial W}\right\} = tf.gradients(loss, tf.trainable\_variables())$$

Why *vec*?

将张量集合拉成向量是一件非常破坏并行效率的行为,这也是为什么我们需要重新定义一种新的计算单元***vec***的缘故.

What is *mat*?

具有相同长度,且对应位置张量形状相同的张量集合的集合被称作是***mat***.

- 例如demo中cost function对变量的近似"Hessian矩阵"就是一个***mat***.
$$\left\{\begin{matrix} \left\{\frac{\partial^2 L}{\partial K_1 \partial K_1}, \frac{\partial^2 L}{\partial K_1 \partial K_2}, \frac{\partial^2 L}{\partial K_1 \partial W}\right\}, \\ \left\{\frac{\partial^2 L}{\partial K_2 \partial K_1}, \frac{\partial^2 L}{\partial K_2 \partial K_2}, \frac{\partial^2 L}{\partial K_2 \partial W}\right\}, \\ \left\{\frac{\partial^2 L}{\partial W \partial K_1}, \frac{\partial^2 L}{\partial W \partial K_2}, \frac{\partial^2 L}{\partial W \partial W}\right\}, \end{matrix}\right\}$$

What is *eye*?

eye指的是一类特殊的张量$T_{ijk}^{uvw} = \delta(i,u)\delta(j,v)\delta(k,w)$,如果$T_{ijk}^{uvw}$上下标对应的维度并不相等,则定义$T_{ijk}^{uvw} \equiv 0$.

Why *eye*?

对于***Newton or quasi-Newton's method***而言经常需要用到单位阵,而如果真的在计算中添加单位阵则其维度是非常夸张的,所以采用***eye***的好处就是只关心对角块上的'子单位阵'.降低了运算负担.

What is *tensor_prod*?

```python
def tensor_prod(x, y):
    return tensor
```

$$U_{ijk} \otimes V^{uvw} = T_{ijk}^{uvw}$$

What is *tensor_transuvection*?

```python
def tensor_transuvection(x, y, mode='all'):
    if mode == 'all':
        return numeric
    elif mode == 'right':
        return tensor
    elif mode == 'auto':
        return tensor
```

$$\begin{cases} U_{ijk} \odot V^{ijk} = T, & \text{mode} = \text{all} \\ U_{ijk}^{uvw} \odot V_{uvw} = T_{ijk}, & \text{mode} = \text{right} \\ U_{ijk}^{uvw} \odot V_{uvw}^{rst} = T_{ijk}^{rst}, & \text{mode} = \text{auto} \end{cases}$$

What is *vec_outer_prod*?

```python
def vec_outer_prod(xs, ys):
    return {{tensor, ...}, ...}
```

$xs = \{X_1, \ldots\}, ys = \{Y_1, \ldots\}, X_i, Y_i$ are all tensors
$$vec\_outer\_prod(xs, ys) = \{\{X_1 \otimes Y_1, \cdots, X_1 \otimes Y_n\}, \cdots, \{X_n \otimes Y_1, \cdots, X_n \otimes Y_n\}\}$$

What is *vec_inner_prod*?

```python
def vec_inner_prod(xs, ys, mode):
    if mode == 'all':
        return numeric
    elif mode == 'right':
        return tensor
    elif mode == 'auto':
        return tensor
```

$$\sum_i tensor\_transuvection(X_i, Y_i, mode), \quad X_i \in xs, Y_i \in ys$$

What is *mat_vec_prod*?

```python
def mat_vec_prod(ms, xs, mode):
    return {tensor, ...}
```

$xs = \{\{M_{11}, \cdots, M_{1n}\}, \cdots, \{M_{m1}, \cdots, M_{mn}\}\}, xs = \{X_1, \ldots, X_n\}, M_{ij}, X_i$ are all tensors.
$$mat\_vec\_prod(ms, xs, mode) = \{vec\_inner\_prod(\{M_{11}, \cdots, M_{1n}\}, xs, mode), \cdots\}$$

What is *mat_mat_prod*?

```python
def mat_mat_prod(mxs, mys, mode):
    return {tensor, ...}
```

Think of mys as column vectors $\{ys_1, \cdots, ys_m\}$,then use ***mat_vec_prod***.

Limits

存在以下客观事实:

- [1] TensorFlow(包括大部分基于计算图的框架)每一次运行都是强制并行的,除了较少的控制依赖以外.
- [2] TensorFlow原生代码应当是不依赖我们设计的优化模块的.
- [3] 优化模块应当独立于网络结构的设计流程.
- [4] 优化模块应当使得优化算法的实现得到最大限度的并行.
- [5] 优化模块应当尽可能避免计算图的扩大.
- [6] 优化模块附加产生的计算图不应当存在大量结构重复的子图.
- [7] 优化模块尽可能的减少设备间的数据传输,例如除样本以外的张量的设备间传输是不被允许的(即便是样本也应当尽可能避免反复传输同一组样本),但是极少的标量或者bool值的数据传输是被允许的.
- [8] 优化模块应当尽可能的减少存储节点(un-trainable Variable)的增加.缩减不必要的计算开销,因为正常情况下,反向传播的浮点数计算开销是会大于正向传播的.这种情况在拟牛顿法中特别严重,如果参数有1e+6个,那么计算开销就会提高1e+6倍.

尤其值得注意的一点是,***对于单device****而言*,***线****搜索非常的不友好*:

```python
def step_length(f, g, xk, pk, alpha=1.0, is_newton=False, iters=20):
    low = 0.0
    high = 1.0

    c1 = 1e-4
    c2 = 0.9 if is_newton else 0.1

    f_0 = f(xk)
    g_pk = np.dot(g(xk), pk)

    for i in range(iters):
        cond1 = f(xk + alpha * pk) <= f_0 + c1 * alpha * g_pk
        cond2 = abs(np.dot(g(xk + alpha * pk), pk)) <= c2 * abs(g_pk)
        if cond1 and cond2:
            return alpha

        f_high = f(xk + high * pk)
        f_low = f(xk + low * pk)
        g_low_pk = np.dot(g(xk + low * pk), pk)

        alpha = - g_low_pk * (high**2) / 2 / (f_high - f_low - g_low_pk * high)
        if alpha < low or alpha > high:
            alpha = (low + high) / 2

        g_alpha_pk = np.dot(g(xk + alpha * pk), pk)
        if g_alpha_pk > 0:
            high = alpha
        elif g_alpha_pk <= 0:
            low = alpha

    return alpha
```

在并行程序中类似$f_high = f(xk + high * pk), \quad f_low = f(xk + low * pk)$这样的计算过程是本质不并行的.只有2种思路可以解决这2个式子的计算:

- $x \leftarrow xk + high * pk$ and output $f(x)$, and then $x \leftarrow xk + low * pk$ and output $f(x)$
- execute assign op $x \leftarrow xk + high * pk$ in ***Graph/Session 1*** and output $f(x)$ in ***Graph/Session 1***,then execute assign op $x \leftarrow xk + low * pk$ in ***Graph/Session 2*** and output $f(x)$ in ***Graph/Session 2***.

第一种本质串行.第二种虽然本质是并行的,但是计算图的规模至少要扩大一倍,而且计算过程中会中断至少一次进行设备间的数据传输.可见[1~8]想全部实现是不现实的.