Task 3: Custom Implementation

整体思路

1. 为了充分利用作业四的自动微分框架，我定义 `Conv2D` 和 `MaxPooling` 两个继承自 `TensorOp` 的算子，其中 `compute` 正向传播和 `gradient` 反向传播均使用作业三编译好的卷积层和池化层的cuda实现（ `MaxPooling` 反向传播在作业三没有实现CUDA版本，在此处我实现了 `for-loop` 和 `CUDA` 两个版本，并进行实验比较）。
2. 利用作业五的优化器，实现三种模型架构：两层线性层的 `pure_linear` 架构、一层卷积层两层线性层的 `simple_conv` 架构、 `LeNet` 模型架构。具体来讲，对于每种架构，需要在 `set_structure` 中增加模型权重，更改 `forward` 函数，在优化器中更新每一层的权重。
3. 为了适配各种模型架构，避免模型架构改变之后需要相应改变优化器的权重更新代码，我通过遍历模型所有权重，实现适配所有模型架构的更一般的参数更新策略。
4. 实验发现我写的for-loop版本池化层反向传播耗时较多，所以我实现了CUDA并行的池化层反向传播，实验发现能显著提高运行速度。

代码结构

- CUDA代码在 `./MyTensor` 中，运行 `python setup.py develop` 即可编译
- 模型训练的代码位于 `./task1_optimizer.py` 中，运行 `python task1_optimizer.py --model simple_conv` 可训练具有一层卷积层两层线性层的 `simple_conv` 架构的模型， `--model` 还可以选择 `pure_linear` 和 `LeNet` 模型架构

运行方法与结果

如果出现报错，可以尝试 `cd ./MyTensor` 运行 `python setup.py develop` 重新编译

pure_linear

```
python task1_optimizer.py --model pure_linear
```

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
0	0.22435	0.06470	0.22508	0.06670	0.94579
1	0.14339	0.04090	0.15038	0.04400	0.73822
2	0.10491	0.02972	0.11988	0.03450	0.73497

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
3	0.08210	0.02337	0.10400	0.03100	0.67366
4	0.06755	0.01910	0.09560	0.02900	0.66912
5	0.05743	0.01633	0.09076	0.02730	0.73319
6	0.05035	0.01430	0.08808	0.02680	0.69270
7	0.04512	0.01305	0.08606	0.02610	0.70101
8	0.04096	0.01138	0.08459	0.02540	0.67365
9	0.03789	0.01022	0.08348	0.02480	0.68011
10	0.03506	0.00897	0.08228	0.02400	0.69014
11	0.03228	0.00802	0.08065	0.02410	0.66409
12	0.02955	0.00692	0.07888	0.02290	0.67286
13	0.02747	0.00615	0.07734	0.02260	0.74242
14	0.02596	0.00552	0.07636	0.02260	0.69701
15	0.02488	0.00525	0.07582	0.02270	0.64889
16	0.02410	0.00498	0.07548	0.02210	0.65099
17	0.02358	0.00487	0.07537	0.02200	0.72380
18	0.02325	0.00473	0.07532	0.02200	0.68885
19	0.02304	0.00465	0.07533	0.02200	0.69484

simple_conv

```
python task1_optimizer.py --model simple_conv
```

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
0	0.22450	0.06557	0.22500	0.06870	4.74313
1	0.14129	0.04207	0.14670	0.04590	3.82161
2	0.10393	0.03257	0.11608	0.03600	4.41619
3	0.07885	0.02458	0.09998	0.03100	3.74715

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
4	0.06369	0.02007	0.09137	0.02800	3.64345
5	0.05231	0.01648	0.08724	0.02600	4.47715
6	0.04462	0.01400	0.08595	0.02530	4.18614
7	0.03778	0.01160	0.08470	0.02450	3.89238
8	0.03306	0.01043	0.08597	0.02510	3.80926
9	0.02982	0.00972	0.08827	0.02500	3.72024
10	0.02689	0.00833	0.08952	0.02460	4.40014
11	0.02264	0.00687	0.08822	0.02400	3.86780
12	0.01881	0.00515	0.08613	0.02290	3.90654
13	0.01623	0.00425	0.08498	0.02230	3.68627
14	0.01437	0.00360	0.08466	0.02190	3.52239
15	0.01296	0.00313	0.08437	0.02200	3.93964
16	0.01189	0.00277	0.08417	0.02200	3.79612
17	0.01113	0.00247	0.08417	0.02170	3.68573
18	0.01063	0.00225	0.08426	0.02180	4.57585
19	0.01031	0.00215	0.08432	0.02180	3.90759

LeNet

```
python task1_optimizer.py --model LeNet
```

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
0	0.12592	0.04002	0.11623	0.03650	273.73302
1	0.07156	0.02250	0.07054	0.02180	271.74698
2	0.05404	0.01745	0.05941	0.02020	272.00233
3	0.05036	0.01662	0.06030	0.01930	272.11799
4	0.03622	0.01162	0.05037	0.01680	272.21837

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
5	0.03278	0.01103	0.05229	0.01790	259.61811
6	0.02956	0.01058	0.05080	0.01600	225.49822
7	0.02046	0.00718	0.04375	0.01330	227.44083
8	0.01514	0.00523	0.04190	0.01190	228.30170
9	0.01241	0.00428	0.04323	0.01140	229.07684
10	0.00949	0.00317	0.04298	0.01080	225.78679
11	0.00867	0.00290	0.04391	0.01150	228.54179
12	0.00779	0.00245	0.04425	0.01120	232.68834
13	0.00618	0.00183	0.04338	0.01130	226.14438
14	0.00507	0.00148	0.04325	0.01060	228.71153
15	0.00424	0.00113	0.04321	0.01010	230.34282
16	0.00348	0.00090	0.04283	0.01030	226.27458
17	0.00281	0.00062	0.04210	0.01000	228.54181
18	0.00234	0.00038	0.04137	0.00950	232.39875
19	0.00204	0.00028	0.04076	0.00930	228.27878

分析

- 从loss和err的变化，可以看到在模型变得更加复杂后，过拟合现象有所减弱，在测试集的表现也逐渐增强
- 从每个epoch的用时可以看出，加入卷积层后耗时明显变长。测量前向传播和反向传播的耗时可以看出，反向传播速度明显慢于前向传播，而反向传播过程中耗时主要集中在两个池化层的反向传播。这是因为在作业三中我们没有用CUDA实现并行的池化层反向传播，我在 `MaxPooling` 算子类中，用 `for-loop` 写的反向传播效率很低，导致耗时较长

```
Using LeNet & for-loop version max-pooling backpropagation
forward: 0.005983706563711166
back pool: 0.1325874626636505
back pool: 0.2950657308101654
back: 0.4422866702079773
```

优化：池化层反向传播的CUDA实现

- 将池化层反向传播写成cuda并行，代码实现位于 `./MyTensor/max_pooling.cu` 。实验发现，池化层反向传播速度显著加快

```
Using LeNet & CUDA version max-pooling backpropagation
forward: 0.005050960928201675s
back pool: 0.00039035454392433167s
back pool: 0.00021830201148986816s
backward: 0.011350210756063461s
```

- 运行结果： `LeNet` 运行速度显著提升

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
0	0.12182	0.03867	0.11171	0.03810	12.43567
1	0.09300	0.02955	0.09290	0.03130	10.14336
2	0.06708	0.02175	0.07414	0.02490	10.56740
3	0.04720	0.01587	0.05948	0.02000	11.49345
4	0.03722	0.01222	0.05350	0.01740	9.54382
5	0.02723	0.00885	0.04619	0.01460	10.21054
6	0.02569	0.00862	0.04632	0.01440	10.71043
7	0.02027	0.00713	0.04293	0.01300	9.08935
8	0.01958	0.00703	0.04597	0.01310	10.70305
9	0.01724	0.00625	0.04619	0.01250	10.34052
10	0.01540	0.00542	0.05072	0.01230	8.74703
11	0.01436	0.00493	0.05529	0.01270	9.36119
12	0.01031	0.00353	0.05160	0.01110	9.06001
13	0.00801	0.00277	0.05111	0.01030	9.56766
14	0.00547	0.00182	0.04967	0.01020	9.85894
15	0.00370	0.00125	0.04940	0.00970	10.34537
16	0.00264	0.00053	0.04929	0.00960	9.43985
17	0.00200	0.00035	0.04883	0.00970	10.09448

Epoch	Train Loss	Train Err	Test Loss	Test Err	Epoch Time
18	0.00163	0.00028	0.04859	0.00950	10.13023
19	0.00144	0.00018	0.04830	0.00960	9.29091

TODO

实现imagenet结构