Report on speeding up Pointnet++

Xiuye Gu

1 Introduction

Our goal is to to speed up Pointnet++. We tried two directions – (1) using engineering techniques, speed-up Pointnet++ without losing accuracy; (2) adapt the deep neural network's architecture, which may change the accuracy.

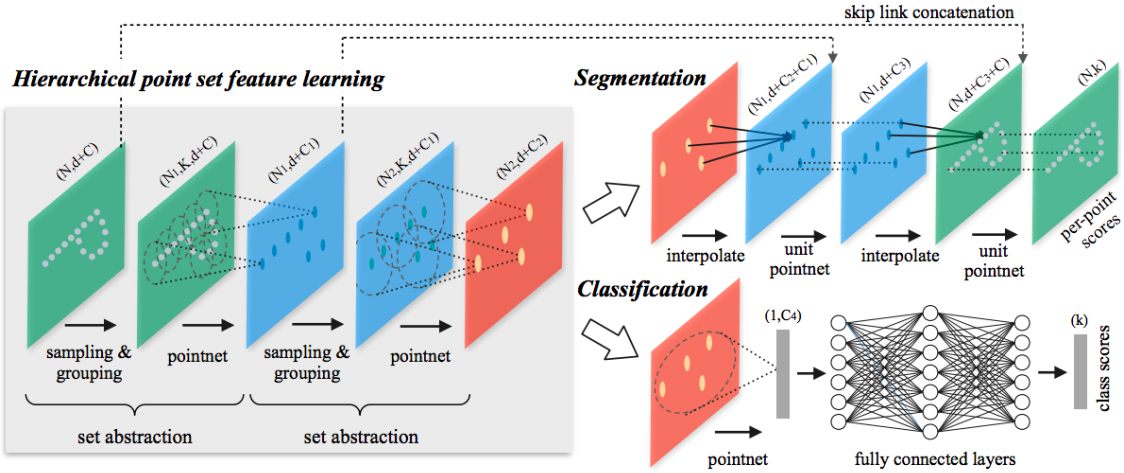

Figure 1: Architecture of pointnet++

As is shown in Figure 1, a quick refresher of Pointnet++'s architecture: before the fully connected layers that lead to the classification scores (we only consider classification nets in this project), the network has a hierarchical structure, and each hierarchical abstraction is composed of a sampling & grouping layer and several convolution layers (including batch normalization and ReLU).

2 Bottleneck for current Pointnet++

We first analyze the bottleneck for current Pointnet++ architecture. The profiling results are shown in Table 1.

From the profiling results, the bottleneck for current Pointnet++ lies in the first set abstraction. The reason is that the first set abstraction has the largest input size, which

Layer	Parameters	Inference Time (secs)
Set abstraction 1	npoint:512, nsample:64, mlp:[64,64,128]	29.472
Set abstraction 2	npoint:128, nsample:64, mlp:[128,128,256]	16.017
Set abstraction 3	npoint:1, nsample:128, mlp:[256,512,1024]	1.446
FC1	num outputs: 512	0.335
DP1		0.166
FC2	num outputs: 256	0.250
DP2		0.146
FC3	num outputs: 40	0.155

Table 1: Profiling for each layer of current PointNet++, where the inference time is the inference time for one epoch on Modelnet40 and is measured by averaging 10 trials. (FCs denote fully connected layers; DPs denote dropout layers.)

makes both the Sample & Grouping and the convolution layers take longer inference time. And the fully connected and dropout layers' inference time is trivial compared to that of the set abstraction layers. So focusing on optimizing the set abstraction layers, especially the first one, will lead to better speed up.

3 Without-losing-accuracy techniques

3.1 Techniques

- **1st conv & approx BN:** In each sampling & grouping layer, n centroids are sampled, and then $nsample$ points are grouped to each centroid; each group of points are translated to a local frame relative to the centroid.

  For the original Pointnet++, in each hierarchical abstraction, the input to the several convolution layers are of size $(batch\_size, n, nsample, nchannels)$, where the translation increases of the input size. Since the first convolution layer is still linear, we can convolve on the untranslated points and convolve on the translation values, and then combine the two convolution results and thus speed up the computation. The new input size is $(batch\_size, ndataset, nchannels)$ and $(batch\_size, n, 3)$.

  The computation of batch normalization (BN) is expansive and correlates with the input size, but it can be approximated by the untranslated points, *i.e.*, we first do all the convolution & BN & ReLU on the untranslated points to obtain their mean, variance, β, γ, and use these variables to approximate the BN computation on the translated points. The convolution on the untranslated points and the translated points share parameters. This step is an approximation and affects the accuracy slightly, but in our next step, we replace the speed-up on BN with another lossless technique.

  Another common speed-up technique is that, under inference mode, the batch normalization is only a linear computation, so it can be incorporated into the convolution

layer.

- **Fused BN:** Fused batch norm combines the multiple operations needed to do batch normalization into a single kernel. Using fused batch norm can result in a 12%-30% speedup[1].

- **NCHW:** N,H,W,C refers to the number of images in a batch, the number of pixels in the vertical (height) dimension, the number of pixels in the horizontal (width) dimension, the number of channels respectively. Within TensorFlow there are two naming conventions representing the two most common data formats: NCHW and NHWC. NCHW is the optimal format to use when training on NVIDIA GPUs using cuDNN[2].

- **Broadcasting:** Broadcasting allows us to perform implicit tiling which makes the code more memory efficient[3], and thus speeds up the program. The Pointnet++ architecture involves many broadcasting operations.

- **Input pipeline:** Instead of waiting for the data to be loaded from memory, we use TensorFlow new Dataset API to make the data-loading & preprocessing and training parallel. Furthermore, placing input pipeline operations on the CPU can significantly improve performance – utilizing the CPU for the input pipeline frees the GPU to focus on training. This technique will be more useful when the dataset is larger.

Techniques	Train one epoch time (secs)	Test one epoch time (secs)
Original Pointnet++	190.76	11.54
1st conv & approx BN	138	10.9
Fused BN	111.24	10.68
NCHW	105.03	9.92
Broadcasting	102.53	9.46
Input pipeline	99.88	9.12

Table 2: Speeding up results on Modelnet 40, adding modifications one by one. (The time measurement can fluctuate by 0.5 secs)

3.2 Experiments

Results on Modelnet 40 are shown in Table 2, the techniques are added one by one. The evaluation criteria is the time used to train one epoch (going through the whole dataset) and the time used to test one epoch.

[1] `https://www.tensorflow.org/performance/performance_guide#common_fused_ops`
[2] `https://www.tensorflow.org/performance/performance_guide#data_formats`
[3] `https://github.com/vahidk/EffectiveTensorflow#broadcast`

Layer	Parameters	Inference Time (secs)
Set abstraction 1	npoint:512, nsample:64, mlp:[64,64,128]	19.737
Set abstraction 2	npoint:128, nsample:64, mlp:[128,128,256]	9.926
Set abstraction 3	npoint:1, nsample:128, mlp:[256,512,1024]	2.491
FC1	num outputs: 512	0.153
DP1		0.130
FC2	num outputs: 256	0.139
DP2		0.117
FC3	num outputs: 40	0.102

Table 3: Profiling for each layer of the speeded-up PointNet++, where the inference time is the inference time for one epoch on Modelnet40 and is measured by averaging 100 trials. (FCs denote fully connected layers; DPs denote dropout layers.)

We can observe that the acceleration on the first convolution layer and the fused BN makes the most significant improvements. The final speed-up is 52% of the original training time and 75% of the original testing time, while the accuracy is not affected at all.

Profiling for the speeded-up Pointnet++ on Modelnet 40 is shown in Table 3.

4 Adaptation on architectures

4.1 Using point coordinates implicitly

In the original Pointnet++, it combines the translated coordinates with the computed features to form the new input in every hierarchical abstraction. Since the Sample & Grouping layer already takes coordinates into consideration – the centroids are sampled using farthest point sampling and we group nearby points only. So except for the first abstraction (where the coordinates are the only input), in the following abstractions, we can use the point coordinates implicitly via Sample & Grouping and not use them as explicit input to the convolution layers. In this way, the translated points ($n \times nsample$) share the same features, so we do the convolution on untranslated points ($ndataset$) first, and then assign translated points their corresponding new features.

The adaptation may reduce the accuracy, so we conduct experiments on Shapenet 55, which is a larger and more difficult dataset than Modelnet 40. Experimental results are shown in Table 4. "use xyz" denotes whether we use the point coordinates explicitly. According to the results, the evaluation accuracy does not decrease much when we use the point coordinates implicitly, and the speed-up is not trivial.

A detailed profiling of the inference time of most components of the deep neural network is in [4].

[4] https://docs.google.com/a/stanford.edu/spreadsheets/d/1FW2-h3ho5V9TFkhViSWEKt4OBc2I6sHCq_9h2jR9CJQ/edit?usp=sharing

use xyz	Train one epoch time (secs)	Test one epoch time (secs)	Evaluation accuracy
True	405.18	16.84	0.877717
False	323.13	13	0.8713

Table 4: Speeding up results on Shapenet 50, using point coordinates implicitly.

4.2 Translating the features

Translating the coordinates increases the input size of the convolution layers. One tentative adaptation is to translate the features, *i.e.* the output of the convolution layers. It may reduce the accuracy, and because of the nonlinearity of the convolution layers, the reasoning behind it is different.

Experimental results on Modelnet 40 are shown in Table 5. The speed-up and the decrease in GPU Memory usage are significant.

Tranlate	Train time (secs)	Test time (secs)	GPU memory usage (MiB)	Accuracy
Coordinates	190	11	7900	0.905032
Features	75	7	3047	0.891234

Table 5: Speeding up results on Modelnet 40, translating the features.