NORTHEASTERN UNIVERSITY

EECE5698

PARALLEL PROCESSING FOR DATA ANALYTICS

Final Project

Author:	*Student Number:*
Tong Jian	001224147
Zifeng Wang	001211627

April 18, 2019

1 Problem and Background

Given temporal radio signals in the form of IQ samples coming from different devices, design a classifier to predict exactly which devices they are from. As each device might add a certain internal influence on the original signal due to they are from different manufacturer or their hardware have slight difference etc. So different than straightforward classification problems, the information signals carry are not important for us, on the contrary, the slight 'noise' due to device difference is the key factor we should focus on.

Recently, deep learning has proven its power on different tasks such as computer vision, natural language processing thanks to the advances in the computational powers. Vanilla neural networks consist of layers of linear mapping and non-linear activation functions, so they are able to approximate nearly any kind of functions as long as data is abundant. Also, there are specific designed structures to capture certain features in data, such as convolutional neural networks for spatial relationship and translation invariance, recurrent neural networks for temporal relationship etc.

For this task, we decide to use convolutional neural network to capture the device information by using a sliding window on the whole example so that the neural network combines both the local and the global feature of the signal as in Fig 1. We take advantage of the computation power of the Graphics Processing Unit (GPU) to train and test our model.

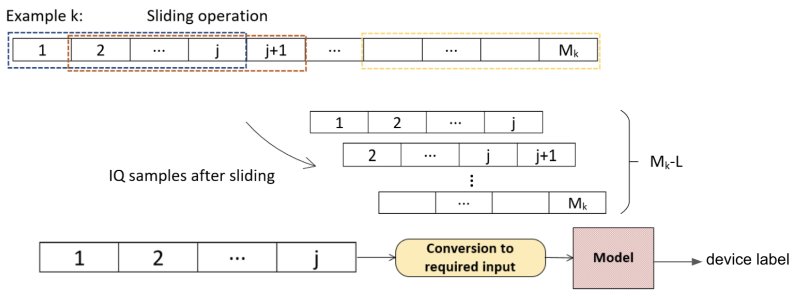

Figure 1: Training procedure

2 Parallelism on GPU

A graphics processing unit (GPU) [1] is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. However, during these days, GPU has more general computational purpose. While CPU is cache heavy, focusing on individual thread performance, GPU is ALU heavy, pursuing overall throughput. So GPU is a perfect match for the Backpropagation algorithm for training deep learning models due to its great acceleration ability on matrix calculations.

CUDA [2] is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. By following the

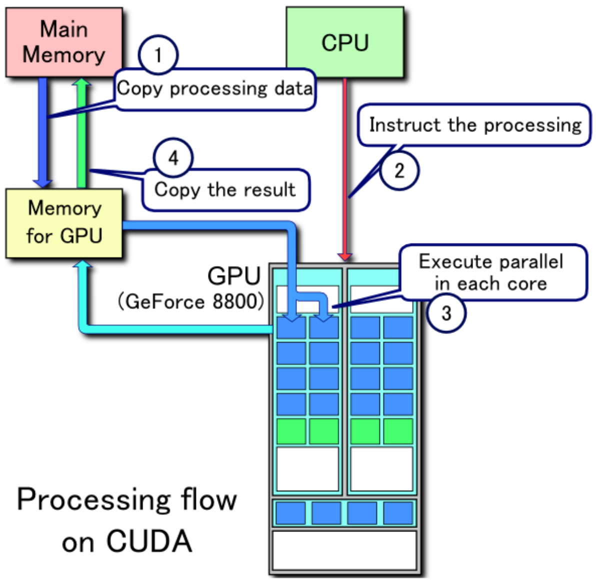

Figure 2: Working mechanism of CUDA

workflow in Fig 2, we are able to traing and test neural networks much faster than doing it on CPUs.

There are many deep learning frameworks which are based on GPU and CUDA, but wrap them in high-level programming languages like Python. Popular frameworks includes TensorFlow, PyTorch and so on. There are even more convenient frameworks like Keras which integrates basic frameworks so that users are able to implement neural networks in just lines of code. In this project, we use Keras with TensorFlow backend to implement our classification model. To show that GPU really accelerates the process, we conduct the same experiment on a a single CPU, a single GPU and multiple GPUs (here we use 4 GPUs together). By comparing the execution time, we demonstrates the helpfulness of using GPUs.

3 Dataset

We have 88,000 examples from 500 devices. Specifically, there are 141 training examples and 35 testing examples per device. Each example is a $l \times 2$ vector. l, the length of each example may vary. We encode each device to a unique number represents the label, $e.g.$, for the 500 devices we have, we encode them to $0 \sim 499$. Also, to do model selection and avoid overfitting, we partition the whole training set into 80% training and 20% validation. We train the model on training set, monitor its performance on validation at the end of every training epoch. Finally we evaluate the performance on test set.

4 Solution

4.1 Designed Network

We design a Convolutional Neural Network(CNN) [3] as the classifier using Keras based on TensorFlow [4] framework, an example architecture could be Fig 3.

CNN is developed from neural network, inspired by living organisms and based on the shift

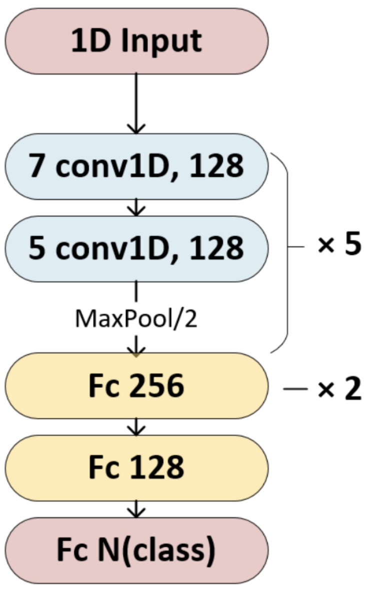

Figure 3: A possible architecture of proposed CNN model

invariant property of natural images. A CNN typically consists of different types of certain layers.

Convolution layer Convolution layer is the most important layer of a CNN. It has a number of different kernels(filters) with the same size and these kernels are convolved with each area across the input with a certain stride(this controls how far should a kernel move across the input each time). The output of the convolution layer is a number of feature maps with decreased size compared to the original input. Convolution layer has an important property that it is locally connected, which means each neuron in a convolution layer does not connect to all neurons in the last layer.

Pooling layer The function of this layer is down-sampling the input and reducing features in order to reduce parameters as well as the computation burden. It is also used to achieve feature invariance. Typically, the input is divided into several sub-areas and the output is computed by all values in the area. The most commonly used one is max pooling which outputs the maximum value of a area. And the size of a sub area is often chosen as 2 without overlapping. By max pooling we can not only reduce features but also increase the non-linearity of the model.

Fully connected layer After some convolution layers and pooling layers, there are often fully connected layers whose neurons are fully connected to the neurons of the last layer. This is the same to the layer in a normal neural network. The fully connected layer helps to generate high-level features and compute loss at the end of the network.

There are other related layers involved to improve performance. Activation layer introduces the non-linearity of the model. ReLU [5] function is one of the most widely used activation function. Loss layer at the end of the CNN computes the loss which we try to minimize during training. For classification, categorical cross entropy loss is often chosen. And there are some regularization methods to avoid over-fitting like dropout method.

4.2 Proposed Solution

The whole solution is designed as following. We first partition the examples to slices with equal length, so that we have a fixed input size. Then we feed batches of slices to our designed CNN with a softmax output and categorical cross entropy loss. Then we run SGD [6] on that loss to optimize weights of our CNN.

This could be done by one CPU/GPU or multiple GPUs, which means both parallelism and distributed systems are involved. Running on multiple GPUs is non-trivial, we need to handle the distributed weights by constructing **multi_gpu_model**.

5 Experiments

5.1 Validation Analysis

We divide the whole dataset into training, validation and test sets, use training and validation sets during training phase to find the best parameter settings and test set to do final evaluation.

We report per-slice accuracy on validation set to get the best parameter configurations. Once parameter settings fixed, per-slice classification results on test set would be predicted by the best model. We use Majority Vote to compute per-example accuracy, where the model makes a prediction(vote) for each test slice and the final output prediction for each example is the one that receives the most votes from its slices.

Some parameters which contribute to the accuracy the most might be configurations of network architecture, slice size, and number of selected training slices. To better illustrate the number of selected training slices, we introduce a **sampling factor** κ which indicates the number of slices to use. It can take values between 1 and slice size. The total number of slices is $\frac{\kappa}{\text{slice\_size}}$ smaller. Thus, larger the κ, larger the number of selected training slices. We change one parameter at a time, keep others the same, and generate the best parameter combinations on validation set.

Table 1 shows the validation accuracy of different network architecture with slice size $= 1024$ and $\kappa = 10$. It tells us that using 10 conv layers and 3 fc layers can provide best performance so far.

# conv layers	# fc layers	per-slice accuracy on validation
8	2	0.198
10	2	0.183
10	**3**	**0.202**

Table 1: Validation accuracy of different network architecture with slice size$= 1024$ and $\kappa = 10$

Fixing the network architecture, Table 2 shows the validation accuracy of different parameter settings.

slice size	κ	per-slice accuracy on validation
128	10	0.275
256	**10**	**0.356**
512	10	0.329
1024	10	0.202
256	1	0.218
256	5	0.278
256	10	0.356
256	**16**	**0.369**

Table 2: Validation accuracy of different parameters with fixed network architecture

Using the best parameter configurations, we test our model on test set and get $(0.364, 0.454)$ for per-slice and per-example accuracy respectively, here to calculate per example accuracy, we use majority vote to decide example assignment based on the slices it contains.

5.2 Performance Analysis

We compare the running time of our best model (both training and testing) on 1 CPU, 1 GPU, and 4 GPUs, to analysis performance in terms of parallelism. Like shown in Table 3, there is 2.6 times speed-up of 4 GPUs comparing to 1 GPU and 123 times than 1 CPU. Apparently, parallelism helps a lot on this computationally intensive task.

Hardware	Number	Configuration	Time Per Epoch(s)	Total Time(h)	# Epochs
CPU	1	Dual Xeon E5-2699 v4	44524	148.41	12
GPU	1	V100	961	3.22	12
GPU	4	V100	359.5	1.20	12

Table 3: Performance analysis

6 Work Split

We share work on the code base and split tasks in terms of performance analysis, specifically,

- Tong runs experiments and does analysis on 1 GPU and 1 CPU
- Zifeng runs experiments and does analysis on multiple GPUs

References

[1] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. 2008.

[2] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[5] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[6] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.