Chapter 1

A Faster ScatterNet

The drive of this thesis is in exploring if wavelet theory, in particular the DT$\mathbb{C}$WT, has any place in deep learning and if it does, quantifying how beneficial it can be. The introduction of more powerful GPUs and fast and popular deep learning frameworks such as PyTorch, Tensorflow and Caffe in the past few years has helped the field of deep learning grow very rapidly. Never before has it been so possible and so accessible to test new designs and ideas for a machine learning algorithm than today. Despite this rapid growth, there has been little interest in building wavelet analysis software in modern frameworks.

This poses a challenge and an opportunity. To pave the way for more detailed research (both by myself in the rest of this thesis, and by other researchers who want to explore wavelets applied to deep learning), we must have the right foundation and tools to facilitate research.

A good example of this is the current implementation of the ScatterNet. While ScatterNets have been the most promising start in using wavelets in a deep learning system, they are orders of magnitude slower and significantly more difficult to run than a standard convolutional network.

Additionally, any researchers wanting to explore the DWT in a deep learning system have to rewrite the filter bank implementation themselves, ensuring they correctly handle boundary conditions and ensure correct filter tap alignment to achieve perfect reconstruction.

This chapter describes how I have built a fast ScatterNet implementation in PyTorch with the DT$\mathbb{C}$WT as its core. At the core of that is an efficient implementation of the DWT. The result is an open source library that provides all three, available on GitHub as *PyTorch Wavelets*

In parallel with my efforts, the original authors of the ScatterNet have improved their implementation, also building it on PyTorch. My proposed DT$\mathbb{C}$WT ScatterNet is $20-30$x faster than their improved implementation, while using less memory.

1.1 The Design Constraints

The original authors implemented their ScatterNet in matlab using a Fourier based Morlet wavelet transform.

The standard procedure for using ScatterNets in a deep learning framework up until recently has been to:

1. Pre scatter a dataset and store the features to disk. This can take several hours to several

days depending on the size of the dataset and the number of CPU cores available.

2. Build a network in another framework, usually Tensorflow or Pytorch
3. Load the scattered data from disk and train on it.

We saw that this approach was suboptimal for a number of reasons:

- It was slow and needed to be run on CPUs.
- It was inflexible to any changes you wanted to investigate in the Scattering design; you would have to re-scatter all the data and save elsewhere on disk.
- You could not easily do preprocessing techniques like random shifts and flips, as each of these would change the scattered data.
- The scattered features were often larger than the original images, and required you to store entire datasets twice (or more) times.
- The features were fixed and could only be used as a front end to any deep learning system.

To address these shortcomings, all of the above limitations became design constraints. In particular, the new software should be:

- Able to run on GPUs (ideally on multiple GPUs in parallel).
- Flexible and fast so that it could be run as part of the forward pass of a neural network (allowing preprocessing techniques like random shifts and flips).
- Able to pass gradients through, so that it could be part of a larger network and have learning stages before scattering.

To achieve all of these goals, we choose to build our software on PyTorch, a popular open source deep learning framework that can do many operations on GPUs with native support for automatic differentiation. PyTorch uses the CUDA and cuDNN libraries for its GPU-accelerated primitives. Its popularity is of key importance, as it means users can build complex networks involving ScatterNets without having to use or learn extra software.

As mentioned earlier, the original authors of the ScatterNet also noticed the shortcomings with their Scattering software, and recently released a new package that could do Scattering in PyTorch **kymatio** addressing the above design constraints. The key difference between our proposed and their improved packages is the use of the DT$\mathbb{C}$WT as the core (their software still uses the original Morlet wavelet design).

As part of our explanation we give pseudo code for the forward and backward passes of all our key layers. This will help us later when we want to build on simpler designs.

1.2 Fast Calculation of the DWT

To have a fast implementation of the Scattering Transform, we need a fast implementation of the DT$\mathbb{C}$WT. For a fast implementation of the DT$\mathbb{C}$WT we need a fast implementation of the DWT. Later in our work will we also explore the DWT as a basis for learning, so having an implementation that is fast and can pass gradients through will prove beneficial in and of itself.

Writing a DWT in PyTorch lowlevel calls is not theoretically difficult to do. There are only a few things to be wary of. Firstly, a 'convolution' in most deep learning packages is in fact a correlation. This does not make any difference when learning, but when using preset filters, as we want to do, it means that we must take care to reverse the filters beforehand. Additionally, as with many packages that provide automatic differentiation, it is common that they will unnecessarily save intermediate activations in a multiscale transform. We should avoid this as GPU memory is precious. This is possible to do with careful definition of the forward and backward pass.

Appendices

References