Parallel Implementation of Gabor Wavelet Processing in PyTorch

Master's Project

Huiran Duan, Zelin Wu
21-737-507, 20-744-249

Submitted on
January 15 2024

Master's Project Supervisor
Prof. Dr. Manuel Günther
Roger Jeasy Bavibidila

University of Zurich
Department of Informatics

Master's Project

Author: Huiran Duan, Zelin Wu

Project period: January 15, 2023 - January 15, 2024

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

Acknowledgements

We would like to express our deepest gratitude to Prof. Dr. Manuel Günther for his unwavering support throughout this project. Prof. Dr. Manuel Günther's commitment to providing comprehensive theoretical insights significantly contributed to our understanding of the project's foundations. His meticulous review of numerous versions of our code, coupled with his vast experience and exceptional scholarly guidance, provided us with a wealth of enlightening suggestions that greatly enriched the quality of our work.

We are also indebted to Roger Jeasy Bavibidila for his contributions. His guidance on code references and step-by-step assistance during meetings helped us navigate unforeseen challenges and ensured our steady progress.

Abstract

Image processing, crucial for information extraction from image data, finds historical roots in the application of Gabor filters. These filters, modeled after the human visual system's early processing stages, have been integral to tasks such as face recognition. Despite their efficacy, existing open-source implementations face challenges, including deactivation and lack of parallelization. This project aims to revitalize a deprecated Gabor wavelet processing package by rewriting it in Python, implementing it in PyTorch, and incorporating parallelization for seamless integration into modern deep learning methods. The enhanced package allows Gabor filtering and similarity computation within PyTorch networks, bridging traditional methods with contemporary deep learning frameworks. This project contributes not only to reactivating the Gabor processing package but also to extending its applicability in the evolving landscape of image processing and deep learning.

Contents

Chapter 1

Introduction

Image processing is the extraction of information from image data and has a long history in research. One particular traditional approach of image processing is the application of Gabor filters (Daugman, 1985), which have been used for the task of face recognition and other face-related tasks (Wiskott et al., 1996; Zhang et al., 2005; Günther, 2012; Günther et al., 2012; Schneider et al., 2011). Gabor wavelets are filters that are applied to images, and they model the first levels of the human visual system. Even when training deep networks, it has been shown that the learned filters have large similarities with Gabor wavelets (Krizhevsky et al., 2012). Early approaches of incorporating Gabor wavelets directly into deep learning systems, however, only model parts of the whole processing chain (Luan et al., 2018).

There exists an open-source package[1] that implements the whole chain of Gabor wavelet processing for images in C++ with Python bindings. This package was implemented by Prof. Dr. Manuel Günther some years ago as part of Bob (Anjos et al., 2012, 2017), but it has been deactivated in the last version due to restructuring and removal of all old implementations.[2] Another problem with the existing implementation is that it is done in C++ and without parallelization. Moreover, the integration of Gabor wavelet processing into contemporary deep learning methods, as per the existing implementation, exhibits suboptimal processing speed. The goal of this Master's project is to rewrite this package in Python so that it can be reactivated. We implement the Gabor wavelet processing in PyTorch. This includes 1) defining the frequency and spatial filters within the PyTorch Module, 2) using these filters to apply transform to input images, 3) extracting Gabor jets from the transformed images, and 4) computing the similarity scores between Gabor jets. We implemented the whole processing in parallel by tensor operations, enabling the integration of Gabor filtering and similarity computation into modern deep learning methods. Additionally, We added some use cases for this new package and updated the documentation.

The report is structured in the following way:

- Gabor Wavelet Processing (Chapter 2): This chapter provides a brief introduction to the piepline of Gabor wavelet processing. It also includes a comparison between the legacy `bob.ip.gabor` package and the new `pytorch_gabor` package implemented by us.
- Milestones and Deliverables (Chapter 3): This chapter outlines the milestones of our project and details the distribution of work among team members.
- Setting Up the Working Environment (Chapter 4): This chapter covers the process of setting up the working environment.

[1] https://gitlab.idiap.ch/bob/bob.ip.gabor
[2] https://gitlab.idiap.ch/bob/bob.ip.gabor/-/issues/6

- The details of our package modules related to the following topics: Gabor Wavelet Transform (Chapter 5), Gabor Jets and Activation (Chapter 6), Grid Graph (Chapter 7), and Similarity Functions and Disparity (Chapter 8). These sections cover both the theoretical aspects and practical implementation, including discussions on interface designs, encountered problems, and the solutions we provided.
- Full Network Example (Chapter 9): This chapter presents a use case demonstrating a full network example with a standard Gabor wavelet processing pipeline and shows a proof of concept.
- Evaluation (Chapter 10): This chapter evaluates the performance enhancement achieved by our new `pytorch_gabor` through parallelization and discusses the results we obtained.
- Conclusion and Future Work (Chapter 11).

Chapter 2

Gabor Wavelet Processing

2.1 Gabor wavelet

Gabor wavelet, named after physicist Dennis Gabor, is widely used in image processing and signal analysis. Constructed by combining complex sinusoidal functions with a Gaussian envelope, the Gabor wavelet serves as a complex-valued filter, excelling in the domain of feature extraction.

Nevertheless, a single Gabor wavelet has its limitations in feature extraction. To address a wider array of features, the Gabor wavelet family comes into play. This family encompasses Gabor wavelets characterized by different scales and directions (cf. Chapter 5). The effectiveness of the Gabor wavelet family lies in its ability to capture a broader spectrum of features using various daughter wavelets, rendering it valuable for discerning patterns and structures in both images and signals.

2.2 Gabor wavelet processing pipeline

The Gabor wavelet processing pipeline is a multi-stage methodology employed in image analysis. This pipeline involves 1) applying the Gabor wavelet transform (GWT) to images; 2) extracting Gabor jets from the Gabor wavelet transformed images; 3) applying similarity functions to compute the similarity scores between Gabor jets. A visual representation of the entire processing pipeline is presented in Figure 2.1.

In Gabor wavelet transform (Chapter 5), we have the option to generate a discrete family of Gabor wavelets either in the spatial domain or in the frequency domain as filters (Günther, 2012). These wavelets allow GWT to extract local variations and structural features within the input image. The Gabor wavelet transformed image obtained through the use of either spatial or frequency domain Gabor filters should be equivalent, as depicted in Figure 2.2.

After generating the Gabor wavelet transformed image (Figure 2.2), the subsequent step involves the extraction of Gabor jets (Chapter 6) at specific positions within the transformed image. Gabor jets comprise aggregation of Gabor wavelet responses at specific positions (Buhmann and Lange, 1989). These Gabor jets also serve as features, with a common choice for their location being the grid graph detailed (Chapter 7). The Gabor jets extracted from the data can be effectively utilized for computing similarity scores using various dedicated similarity functions (Chapter 8). The obtained similarity scores encompass diverse applications, including landmark detection, disparity estimation, and classification tasks (Günther, 2012).

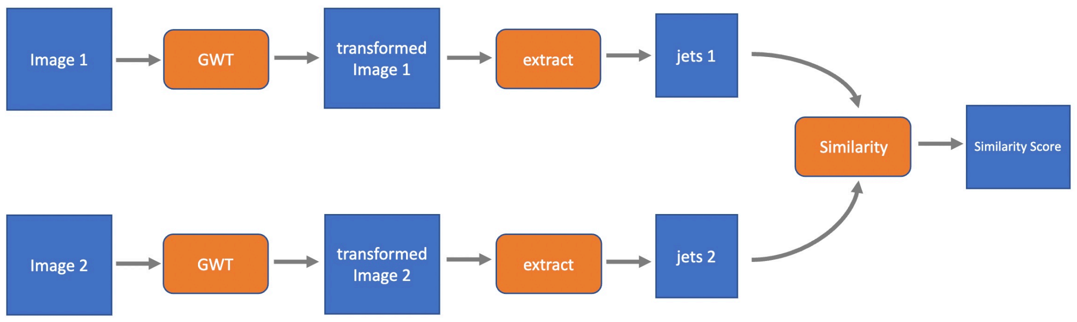


Figure 2.1: THE GABOR WAVELET PROCESSING PIPELINE. *The pipeline includes Gabor wavelet transform (GWT), extracting Gabor jets from transformed image and computing similarity score utilizing the extracted Gabor jets.*

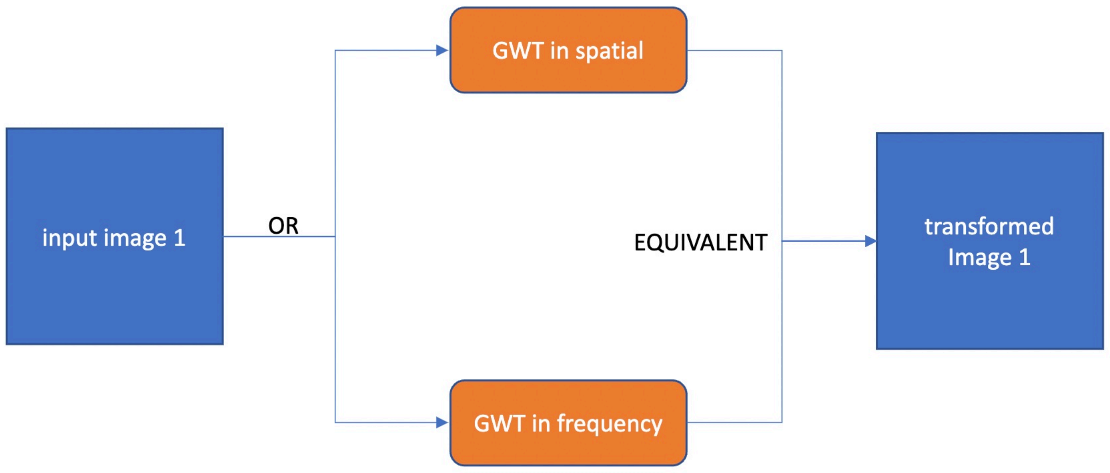


Figure 2.2: TWO WAYS OF APPLYING GABOR WAVELET TRANSFORM. *This figure shows we can apply Gabor wavelet transform to an input image with filters in spatial domain or frequency domain, and their results are equivalent.*

2.3 bob.ip.gabor

`bob.ip.gabor` is part of the signal-processing and machine learning toolbox Bob.[1] It contains a set of C++ code and Python bindings for Bob's image processing tools concerning Gabor wavelets, the Gabor wavelet transform, Gabor jet extraction in a grid graph structure, and Gabor jet similarity functions, including a Gabor jet disparity estimation.

2.3.1 Functionality of bob.ip.gabor

The modules and functions of the `bob.ip.gabor` are outlined below:

1. `bob.ip.gabor.Wavelet`: A class that represents a Gabor wavelet in the frequency do-

[1] https://www.idiap.ch/software/bob/docs/bob/bob.ip.gabor/master/index.html

main.

2. `bob.ip.gabor.Transform`: A class that represents a family of Gabor wavelets in the frequency domain that can be used to perform a Gabor wavelet transform.
3. `bob.ip.gabor.Jet`: A class to manage Gabor jets. As outlined above, a Gabor jet comprises the responses from all Gabor wavelets belonging to the Gabor wavelet family at a specific position within the image.
4. `bob.ip.gabor.Jetstatistics`: A class to compute statistics of a list of Gabor jets and do further calculations.
5. `bob.ip.gabor.Similarity`: A class that computes different kinds of similarity functions between Gabor jets, for example, disparity corrected phase differences.
6. `bob.ip.gabor.Graph`: A class to extract Gabor jets from multiple positions of a Gabor transformed image.
7. `bob.ip.gabor.load_jets`: A function that loads a list of Gabor jets from the given HDF5 file; the file needs to be open for reading.
8. `bob.ip.gabor.save_jets`: A function that saves the given list of Gabor jets to the given HDF5 file; the file needs to be open for writing.

The `bob.ip.gabor` package encapsulates the entire Gabor wavelet processing pipeline. However, this package has some issues:

1. `bob.ip.gabor` is done in C++ and without parallelization.
2. The integration of Gabor filtering into modern deep learning methods under `bob.ip.gabor` implementation is too slow.

2.4 pytorch_gabor

Within the `pytorch_gabor` package, we implemented each stage of the entire Gabor wavelet processing pipeline using PyTorch in a manner that maximizes parallelization. However, we still retained most of the object-oriented approaches as employed in `bob.ip.gabor`. Therefore, the new `pytorch_gabor` maintains a close connection in structure and functionality with the old `bob.ip.gabor`, while introducing many more efficient options.

2.4.1 The difference between bob.ip.gabor and pytorch_gabor

Building upon the previously identified issues with `bob.ip.gabor`, we have implemented several changes in our new package to address these concerns. The corresponding classes and functions between `bob.ip.gabor` and `pytorch_gabor` are shown in Table 2.1.

Parallelization: Data flow in `bob.ip.gabor` primarily relies on arrays, along with basic types like lists and tuples. Although the integration with Numpy arrays is supported through Python binding, utilizing the GPU for processing is not currently feasible.

In `pytorch_gabor`, parallelization is accomplished through `torch.Tensor` operations, facilitating the efficient handling of a batch of images. This package introduces the capability for parallel processing of all positions, departing from the previous method of extracting and processing similarity with a single jet at a time, as observed in the old Bob package.

Modules in bob.ip.gabor	**pytorch_gabor**
bob.ip.gabor.Wavelet	-
bob.ip.gabor.Transfrom	pytorch_gabor.compute_frequency_centers pytorch_gabor.GaborFilterSpatial pytorch_gabor.GaborFilterFrequency
bob.ip.gabor.Jet	pytorch_gabor.Jet pytorch_gabor.GaborFilterAct
bob.ip.gabor.Graph	pytorch_gabor.Graph pytorch_gabor.GridExtract
bob.ip.gabor.Jetstatistics	pytorch_gabor.jetstatistics
bob.ip.gabor.Similarity	pytorch_gabor.Similarity

Table 2.1: THE CORRESPONDING MODULES BETWEEN `BOB.IP.GABOR` AND `PYTORCH_GABOR`.

Integration with deep learning: `bob.ip.gabor` does not allow seamless integration of Gabor filtering into modern deep learning methods. Gabor filters in `pytorch_gabor` inherit from `torch.nn.Module`, facilitating direct usage as PyTorch network layers. This enables easy integration into modern deep learning workflows.

Package structure: `bob.ip.gabor` has a specialized `Wavelet` class as an object-oriented representation of an individual wavelet. These `Wavelet` objects were contained within the class `Transform`. In `pytorch_gabor`, adopting this approach is proven unnecessary. Storing the entire Gabor family directly as a `torch.nn.Parameter` in `self.weight` has proven to be more meaningful and efficient, aligning with modern PyTorch module structures.

Additional functionality in `pytorch_gabor`: When representing the entire data flow in package `pytorch_gabor` using `torch.Tensor`, additional classes inherited from `torch.nn.Module` like `GaborFilterAct` and `GridExtract` are required to process these tensors. In this framework, input tensors and return tensors are handled without the need for encapsulation through any intermediary objects.

Chapter 3

Milestones and Deliverables

3.1 Timeline

Initially, this Master's project was expected to require about 15 weeks if pursued full-time, based on an estimated workload of 30 hours per week and a total of 15 ECTS, averaging 30 hours per ECTS. Despite primarily working part-time on this project, we maintain the use of full-time week structures in the descriptions below to ensure clarity and consistency.

Milestone 1: Gabor wavelet transform in different domains. An image can be filtered using a family of Gabor wavelets, resulting in similar outcomes between Gabor wavelet transforms in spatial and frequency domains.

Week 1-2: Familiarization and Scope Understanding.

- **Deliverables:**
  - Understanding the project's scope and aligning theoretical concepts and formulas with planned functionalities.
  - Executing the existing `bob.ip.gabor` locally to comprehend it and identify areas for refactoring and enhancement.

Week 3-4: Setting up the work environment, installing all required tools, and building a joint software design and interface.

- **Deliverables:**
  - A working environment with all necessary dependencies pushed to the remote repository.
  - An initial draft of UML class diagrams.

Week 5-6: Implementing Gabor wavelet transform in the frequency domain using `torch.fft` for the FFT.

- **Deliverables:**
  - The parallel implementation of `GaborFilterFrequency` class, capable of performing Gabor wavelet transforms in the frequency domain and generating transformed images identical to those produced by the old `bob.ip.gabor.Transform`.

  - A use case `transform.ipynb` demonstrating the generated Gabor wavelets and the Gabor wavelet transform process.

Week 7-8: Implementing the Gabor wavelet family as an extension of the PyTorch Conv2d layer.

- **Deliverables:**
  - The parallel implementation of `GaborFilterSpatial` class, which can transform images in the spatial domain with results consistent with `GaborFilterFrequency`.
  - Successful migration and execution of all old tests related to Gabor wavelet transform on the new implementations (i.e., `GaborFilterFrequency` and `GaborFilterSpatial`).

Milestone 2: Transformed image activation and feature extraction. The activation layer can be applied to the Gabor transformed images with tensors in parallel. Gabor jets can be extracted from Gabor transformed images using the Jet and Graph classes.

Week 9: Implementing the Gabor activation layer to turn complex-valued responses into Euler representations.

- **Deliverables:**
  - Implementation of `GaborFilterAct` that can function as an activation layer for transformed images.
  - A use case `spatial_frequency_activation.ipynb`, demonstrating the application of GaborFilterAct on transformed images using both `GaborFilterFrequency` and `GaborFilterSpatial` classes.

Week 10: Implementing classes to represent the Gabor jet and Gabor graph, including I/O functionality.

- **Deliverables:**
  - Implementation of `Jet` and `Graph` classes to extract, store, save, and load batched jets at single and multiple positions, respectively.
  - Successful migration and execution of all old tests related to Gabor jets and graphs on the new implementations.

Milestone 3: Similarity functions. Gabor jet disparities and other similarity functions are implemented.

Week 11-12: Implementing similarity functions for two lists of Gabor jets.

- **Deliverables:**
  - Development of the `Similarity` class capable of computing graph similarities using various functions for two lists of Gabor jets.
  - Creation of test cases to ensure that the outcomes from the new `Similarity` class are the same as those acquired from the old `bob.ip.gabor.Similarity`.

Week 13: Implementing similarity functions for two tensors of activated transformed images in parallel.

- **Deliverables:**
  - Enhancing the `Similarity` class to accommodate parallel processing of two tensors containing Gabor jets extracted from multiple positions.
  - Two use cases, `similarity.ipynb` and `disparity.ipynb`, showcasing the computation of Gabor jet similarity and disparity maps.

Milestone 4: Full network example. A grid graph to extract Gabor jets as tensors (rather than objects) is implemented. A full network example can show a proof of concept.

Week 14: Implementing the grid graph inherited from `torch.nn.Module` and demonstrating a full application as a use case, showcasing a basic pipeline adaptable for integration into standard PyTorch neural networks.

- **Deliverables:**
  - Development of the `GridExtract` class to extract jets as tensors from transformed images using a specified stride.
  - The use case `full_network_example.ipynb`, showcasing a small network as a PyTorch Module that can run on a GPU. This demonstration utilizes `GaborFilterFrequency`, `GaborFilterSpatial`, `GridExtract`, `GaborFilterAct`, and `Similarity` classes.

Milestone 5: Finalization. All test cases pass, the documentation is up-to-date, and the final report is completed.

Week 15: Updating the documentation of the package and improving the test cases. Writing the final report and preparing the presentation.

- **Deliverables:**
  - Test cases for greater relevance.
  - Comprehensive documentation for each class and function within the new package.
  - Final report.

3.2 Work Distribution

Huiran Duan:

- Interface design
- Refinement of the working environment (based on the draft version from Prof. Dr. Manuel Günther)
- Implementation of `GaborFilterSpatial`
- Review and refinement of `GaborFilterFrequency`

- Implementation of `GaborFilterAct`
- Review and refinement of `Jet` and `Graph`
- Parallel implementation of `Similarity`
- Implementation of `GridExtract`
- Implementation of full network examples
- Implementation of all Jupyter notebook use cases
- README document
- Relevant parts of the test cases
- Relevant parts of the final report
- Relevant parts of the presentation

Zelin Wu:

- Raw version of `Jet`
- Raw version of `Graph`
- Raw version of `Similarity`
- Implementation of `Jetstatistics`
- Implementation of `utils`
- Documentation for most classes
- Code review and bug fixes
- Relevant parts of the test cases
- Relevant parts of the final report
- Relevant parts of the presentation

Chapter 4

Setup Working Environment

Our work primarily unfolded within the macOS system, though our package is compatible with Windows and Linux systems. We centered our development process around the Python Programming Language (version >=3.10) and set up the working environment using Conda.

4.1 Dependencies

We created a Conda environment with the following dependencies:

```
dependencies:
  - python>=3.10
  - numpy>=1.22.3
  - matplotlib>=3.5.2
  - pytorch>=2.0.0
# - pytorch-cuda>=1.11.0 # if CUDA is available
  - torchvision>=0.12.0
  - tqdm>=4.64.0
  - pyyaml
  - pip
```

Please note that CUDA and MKL are unavailable on macOS. To ensure compatibility, we employed `pytorch>=2.0.0` as an alternative. However, users requiring CUDA or MKL capabilities can opt to install them via Conda independently. For instance, on Windows systems with CUDA, users can refer to the corresponding command line:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c
    nvidia
```

which is provided on the PyTorch official website.[1] Additionally, our package can also be used with GPU in other cloud-based environments like Google Colab.

4.2 Problems and Solutions

Given that our primary working environment was macOS, it posed a challenge as our package also needed parallelization in a GPU environment. To address this, we opted for the more versatile and convenient Google Colab cloud environment for testing. The workflow is as follows:

[1] https://pytorch.org/

1. Set Google Colab's hardware accelerator to GPU (CUDA).

2. Install Conda in Colab. We can utilize a dedicated library, `condacolab`, to easily install Conda and associated packages in Google Colab.

   ```
   !pip install -q condacolab
   import condacolab
   condacolab.install() # expect a kernel restart

   # check if conda has been installed
   import condacolab
   condacolab.check()
   ```

3. Clone the remote repository.

   ```
   !git clone $repo_url
   %cd bob.ip.gabor/
   ```

4. Update the Conda environment. Please note that due to the limitations of Google Colab, we can only use and update the `base` environment instead of creating new ones.

   ```
   # Use mamba to update the base environment (it is faster than conda,
       but it will still take a few minutes)
   !mamba env update -f environment.yaml
   ```

With these configurations in place, running and testing our package with GPU capabilities was as straightforward as in any local environment.

Chapter 5

Gabor Wavelet Transform

The essence of the Gabor wavelet transform (GWT) revolves around meticulously creating a Gabor wavelet family within the spatial or frequency domain. These wavelets are then applied to grayscale input images, undergoing transformation through convolution (in the spatial domain) or element-wise multiplication (in the frequency domain). Ultimately, the GWT yields complex-valued Gabor transformed images as its output.

5.1 Theoretical Analysis

5.1.1 Discrete Gabor Wavelet Family

In the initial step of analysis, it is imperative to clarify the characteristics of an individual Gabor wavelet. The Gabor wavelet family that we have implemented in our code all comes from the variants of the mother Gabor wavelet. It in the spatial domain can be expressed as:

$$\psi(\vec{x}) = \frac{1}{\sigma^2} e^{-\frac{\vec{x}^2}{2\sigma^2}} e^{i\vec{e}_h^T \vec{x}} \tag{5.1}$$

which represents a complex-valued filter divided into two overlaying parts. The first part consists of a Gaussian with a standard deviation σ, imparting localization of the Gabor wavelet in both spatial and frequency domains. The second part is the complex-valued even wave $e^{i\vec{e}_h^T \vec{x}}$ with spatial frequency $\vec{e}_h = (1,0)^T$, pointing along the horizontal axis. On the other hand, the mother Gabor wavelet in the frequency domain can be represented as:

$$\psi(\vec{\omega}) = e^{\frac{-\sigma^2(\vec{\omega}-\vec{e}_h)^2}{2}} \tag{5.2}$$

which is just a Gaussian shifted to frequency coordinates $\vec{e}_h$.

By introducing different scales (k) and directions (θ), we can modify the mother Gabor wavelet, generating new daughter Gabor wavelets. In the spatial domain, it can be written as:

$$\psi_{\vec{k}}(\vec{x}) = \frac{\vec{k}^2}{\sigma^2} e^{-\frac{\vec{k}^2\vec{x}^2}{2\sigma^2}} [e^{i\vec{k}^T\vec{x}} - e^{-\frac{\sigma^2}{2}}] \tag{5.3}$$

and the frequency domain formula now reads as:

$$\psi_{\vec{k}}(\vec{\omega}) = e^{-\frac{\sigma^2(\vec{\omega}-\vec{k})^2}{2\vec{k}^2}} - e^{-\frac{\sigma^2}{2}} e^{-\frac{\sigma^2\vec{\omega}^2}{2\vec{k}^2}} \tag{5.4}$$

where

$$\vec{k} = \begin{Bmatrix} k_h \\ k_v \end{Bmatrix} = \begin{Bmatrix} k \cdot cos(\theta) \\ k \cdot sin(\theta) \end{Bmatrix} \tag{5.5}$$

As a result, the discrete Gabor wavelet family can be constructed based on the following parameter set:

$$\Gamma = (\nu_{\text{max}}, \zeta_{\text{max}}, k_{\text{max}}, k_{\text{fac}}, \sigma) \tag{5.6}$$

where ν_{max} represents the number of directions, ζ_{max} denotes the number of scale levels, k_{max} is the highest frequency, and k_{fac} is the logarithmic distance between two scale levels. Consequently, each wavelet in a Gabor wavelet family can be described in terms of its direction and scale:

$$\theta_\nu = \frac{\nu\pi}{\nu_{\text{max}}}, \qquad \nu = \{0, \dots, \nu_{\text{max}} - 1\} \tag{5.7}$$

$$k_\zeta = k_{\text{max}} k_{\text{fac}}^{\zeta}, \qquad \zeta = \{0, \dots, \zeta_{\text{max}} - 1\} \tag{5.8}$$

For simplicity, we can employ $j \in \{0, ..., J - 1\}$ to index for each Gabor wavelet in a family:

$$j = \zeta\nu_{\text{max}} + \nu \tag{5.9}$$

where $J = \zeta_{\text{max}}\nu_{\text{max}}$ represents the total number of Gabor wavelets generated with the parameter set Γ. The resulting Gabor wavelet is labeled as $\psi_{\vec{k}_j}$ in the spatial domain and $\check{\psi}_{\vec{k}_j}$ in the frequency domain, with the center at $\vec{k}_j$ and a radius of $\frac{1}{\sigma_{\text{eff}}}$, where $\sigma_{\text{eff}} = \frac{\sigma}{k_j}$ represents the effective standard deviation.

The default parameter set Γ_{default} used in Günther (2012) and is defined by (5.10), and it generates the common discrete Gabor wavelet family as depicted in Figure 5.1.

$$\Gamma_{\text{default}} = (8, 5, \frac{\pi}{2}, 2^{-\frac{1}{2}}, 2\pi) \tag{5.10}$$

5.1.2 Transform Using Discrete Gabor Wavelet Family

The historical application of the discrete family of Gabor wavelets in processing the image $\mathcal{I}$, particularly in the context of face detection and recognition, involves depicting a face against a more or less cluttered background. This results in the generation of the Gabor transformed image $\mathcal{T}$ from the original image, composed of J layers denoted as $\mathcal{T}_{\vec{k}_j}$. Each of these layers is derived through the convolution of the input image $\mathcal{I}$ with its corresponding Gabor wavelet in the spatial domain:

$$\begin{aligned}
\mathcal{T}_{\vec{k}_j} &= \left(\psi_{\vec{k}_j} * \mathcal{I}\right)(\vec{t}) && (5.11)\\
&= \sum_{\vec{x}} \psi_{\vec{k}_j}(\vec{t} - \vec{x})\, \mathcal{I}(\vec{x}) && (5.12)\\
&= \sum_{\vec{x}} \overline{\psi_{\vec{k}_j}(\vec{x} - \vec{t})}\, \mathcal{I}(\vec{x}) && (5.13)
\end{aligned}$$

Alternatively, the Gabor transformed image layer $\mathcal{T}_{\vec{k}_j}$ can be obtained by transforming the image $\mathcal{I}$ to the frequency domain, multiplying $\check{I}$ and $\check{\psi}_{\vec{k}_j}$ pixel by pixel:

$$\check{\mathcal{T}}_{\vec{k}_j} = \check{\psi}_{\vec{k}_j}(\vec{\omega})\check{\mathcal{I}}(\vec{\omega}) \tag{5.14}$$

Subsequently, the inverse Fourier transform of $\check{\mathcal{T}}_{\vec{k}_j}$ is applied to return to the spatial domain. Therefore, the Gabor transformed image layer $\mathcal{T}_{\vec{k}_j}$ retains the same resolution as the input image $\mathcal{I}$, but the pixels $\mathcal{T}_{\vec{k}_j}(\vec{t})$, representing the responses of Gabor wavelet $\psi_{\vec{k}_j}$ to image $\mathcal{I}$ at the offset point $\vec{t}$, are complex-valued.

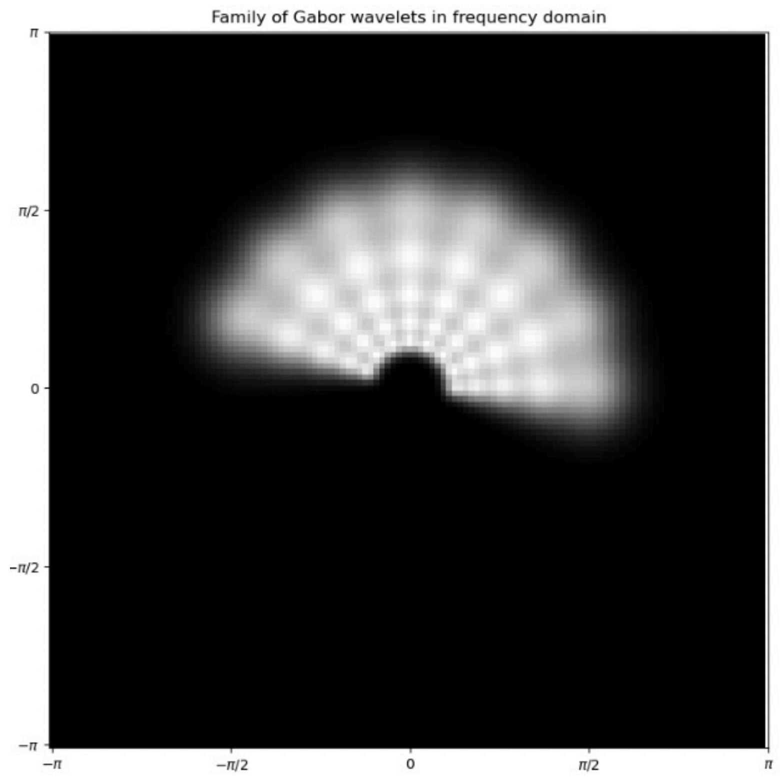


Figure 5.1: THE DISCRETE GABOR WAVELET FAMILY. *This figure depicts the discrete Gabor wavelet family in the frequency domain under the default parameter set given by (5.10). Each Gabor wavelet in the frequency domain, denoted as $\check{\psi}_{\vec{k}j}$, is represented by a circle with a radius of $\frac{1}{\sigma_{\mathit{eff}}}$ centered at $\vec{k}_j$.*

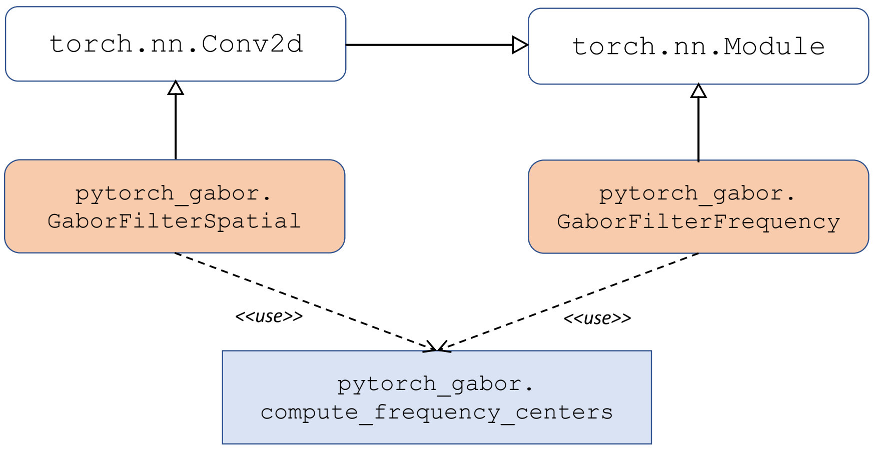


Figure 5.2: CLASS DIAGRAM OF GABOR WAVELET TRANSFORM.

The aforementioned methods represent two approaches to obtain the Gabor wavelet transformed image, providing theoretical guidance for our code implementation. Whether conducting the transform in the spatial or frequency domain, the complex-valued responses (comprising real and imaginary parts) should remain consistent.

5.2 Implementation

In our new `pytorch_gabor` package, the corresponding implementations for the theoretical components reside in the `GaborFilterSpatial` and `GaborFilterFrequency` classes. Figure 5.2 illustrates the relationship between these two classes, where `GaborFilterFrequency` solely inherits from `torch.nn.Module`, while `GaborFilterSpatial` inherits from `torch.nn.Conv2d`. Both classes, in the process of generating their respective Gabor wavelet families, rely on a function named `compute_frequency_centers` to assist in computing a crucial intermediary value, the frequency centers in Equation (5.5).

5.2.1 pytoch_gabor.compute_frequency_centers

Within our `pytorch_gabor` package, computing the frequency centers for daughter wavelets serves as a pivotal step in constructing the Gabor wavelet family.

The process involves several factors and parameters influencing these centers: The direction index ν and the number of directions ν_{max} determine the phases through $\theta_\nu = \pi \cdot (\nu/\nu_{\text{max}})$.

For various scale indices ζ, the number of scales ζ_{max}, the highest frequency k_{max}, and the logarithmic distance between scale levels k_{fac}, frequencies are calculated as $k_\zeta = k_{\text{max}} \cdot k_{\text{fac}}^{\zeta}$.

The computation of frequency centers involves determining the h and v of the vector (5.5) for each wavelet indexed by j in the frequency domain.

The final output comprises a tensor stacked with $[k_{jh}, k_{jv}]$ for all j, encapsulating the frequency centers for the entire Gabor wavelet family.

Problems and solutions

In the initial version, the process of preparing frequency centers was strongly coupled with the remaining steps of generating the wavelet family. Essentially, the same computation for frequency centers was redundantly distributed within both the `GaborFilterSpatial` and `GaborFilterFrequency` classes, resulting in significant code duplication. Upon further theoretical analysis and code reviews, we discovered that these processes could be effectively decoupled. This became evident as equations (5.3) and (5.4) both directly utilize $\vec{k}$. Consequently, we extracted this functionality of computing frequency centers into a separate function interface, which was then called independently by `GaborFilterSpatial` and `GaborFilterFrequency`. It is noteworthy that soon after decoupling, we could focus on optimizing the internal logic of this function itself. This enabled us to easily identify possibilities for refining its algorithm. For instance, we leveraged methods such as `torch.tile` and `torch.repeat_interleave` to avoid any explicit for-loop iterations, rendering the internal algorithm entirely parallelized and efficient:

```
theta = torch.arange(number_of_directions) * torch.pi / number_of_directions
theta_v = torch.tile(theta, [number_of_scales])
scale = torch.arange(number_of_scales)
scale = scale.repeat_interleave(number_of_directions)
k_mu = k_max * (k_fac ** scale)
k_x = k_mu * torch.cos(theta_v)
k_y = k_mu * torch.sin(theta_v)
frequency = torch.stack((k_y, k_x), 1)
```

5.2.2 pytoch_gabor.GaborFilterSpatial

`GaborFilterSpatial` is a class that represents a layer housing a family of Gabor filters (wavelets) in the spatial domain, capable of conducting the Gabor wavelet transform using convolution. During the instantiation of this class, numerous parameters come into play, bifurcating into two main categories for discussion.

The first category encompasses the parameter set Γ (5.6), responsible for generating the Gabor wavelet family. Within the code, these parameters correspond to (`number_of_scales`, `number_of_directions`, `k_max`, `k_fac`, `sigma`). As mentioned earlier, the first four parameters are utilized for computing and generating frequency centers, while sigma, representing Gaussian standard deviation, localizes each Gabor wavelet in both spatial and frequency domains.

The second category of parameters originates from the requirements after inheriting class `torch.nn.Conv2d`. These parameters include `in_channels`, `kernel_size`, `stride`, `padding`, and `padding_mode`. It is important to note that while `out_channels` is also a mandatory parameter in `torch.nn.Conv2d`, its quantity should not be controlled by the user; instead, it should be autonomously calculated during the instantiation process of `GaborFilterSpatial`, where `out_channels = number_of_scales * number_of_directions`. Moreover, we have introduced a new parameter, `number_of_sigma_eff` (cf. (5.18)), to assist the `GaborFilterSpatial` in automatically calculating its kernel size if the user does not specify a specific kernel size.

Subsequently, based on equation (5.3), `GaborFilterSpatial` constructs the complete Gabor wavelet family and stores it in `self.weight`. It is worth noting that the initialization process of `GaborFilterSpatial` has been entirely parallelized by us (meaning, the generation of the

Gabor wavelet family does not depend on any for loop). Additionally, the Gabor wavelet family is stored in `self.weight` as a `torch.nn.Parameter`. This is essential as it allows them to be automatically added to the list of parameters. For instance, when we change the Module's device from CPU to CUDA, the registered parameter `self.weight` will be seamlessly placed on CUDA correctly. Then, owing to inheritance, the specific convolution process entirely delegates to the `forward` function of `torch.nn.Conv2d`.

The `weight` shape in `GaborFilterSpatial` is conventionally specified as `[out_channels, in_channels, *kernel_size]`, where `out_channels` is determined by the product of the `number_of_scales` and `number_of_directions`, denoted as the `number_of_wavelets` (i.e., `J`). Consequently, the shape of the output transformed image is expected to be `[B, J, H', W']`, where `B` represents the batch size, and `H'` and `W'` signify the height and width of the transformed image after convolution. It is noteworthy that, if the padding is appropriately configured, `H'` and `W'` can align with the resolution of the input image.

We have additionally recognized that there is a subtle distinction between the terminologies commonly utilized in theory and that employed in practical code implementation. Nevertheless, we found that these terms essentially convey the same meaning. Therefore, we have employed the `@property` notation to align the terminologies of both Gabor wavelet transform and PyTorch Module, ensuring compatibility without any loss of generality:

```
@property
def wavelets(self):
    return self.weight

@property
def number_of_wavelets(self):
    return self.out_channels
```

Problems and solutions

1. cross-correlation Despite strictly adhering to equation (5.3) to construct the spatial Gabor wavelet family, our tests revealed that `GaborFilterSpatial` consistently failed to perform the Gabor wavelet transform accurately. Specifically, the transformed image by `GaborFilterSpatial` and the transformed image by the old `bob.ip.gabor.Transform` exhibited inconsistency in their imaginary parts, displaying an exact opposite relationship. This prompted us to reevaluate the theoretical analysis of performing GWT in the spatial domain.

The key point here is that, while the term "convolution" is commonly employed in the deep learning domain, strictly speaking, in mathematical terms, what we are actually performing is a "cross-correlation". According to the official documentation of PyTorch, `torch.nn.Conv2d` does not implement 2D convolution in the precise mathematical sense; instead, it executes 2D cross-correlation.

In our code, we should actually consider the conjugate form of the Gabor wavelet (cf. Equation (5.13)), which is corresponding to the definition of cross-correlation. Furthermore, according to the strong symmetric nature of the Gabor wavelet:

$$\psi_{\vec{k}}(-\vec{x}) = \psi_{-\vec{k}}(\vec{x}) = \overline{\psi_{\vec{k}}(\vec{x})} \tag{5.15}$$

the transformed image can be expressed as:

$$\mathcal{T}_{\vec{k}_j}(\vec{t}) = \sum_{\vec{x}} \psi_{-\vec{k}_j}(\vec{x} - \vec{t})\mathcal{I}(\vec{x}) \tag{5.16}$$

This conclusion indicates that we should generate a family of $\psi_{-\vec{k}_j}$ instead of $\psi_{\vec{k}_j}$ during the instantiation of `GaborFilterSpatial`, if we intend to utilize the ready-made cross-correlation provided by the `forward` function in `torch.nn.Conv2d`.

After completing the theoretical analysis above, debugging the code became relatively straightforward. We simply added a negative sign to the frequency centers $\vec{k}_j$. Subsequent tests confirmed that `GaborFilterSpatial` could perform GWT correctly, generating a transformed image consistent with the results obtained from both the old `bob.ip.gabor.Transform` and the new `GaborFilterFrequency`.

2. set_wavelets After our initial implementation of `GaborFilterSpatial`, we discovered that the `__init__` function contained excessive code, primarily due to the intricate process involved in generating Gabor wavelets. This approach was not optimal in software engineering, as it resulted in convoluted code, unclear responsibilities, and diminished readability. Thus, we aimed to devise a method to segregate the Gabor wavelet generation process.

During the subsequent optimization phase, we streamlined the code by isolating the Gabor wavelet generation logic within the `set_wavelets` member function. This adjustment offered another clear advantage: users can now modify specific parameters of a `GaborFilterSpatial` instance to reset its internal wavelets, such as the `number_of_scales`, without creating a new `GaborFilterSpatial` object.

3. compute_spatial_kernel_size and padding_mode = 'circular' In the prior implementation of `GaborFilterSpatial`, users were required to define the `kernel_size` by themselves, with the default `padding_mode` set to `'zeros'`. Although these arguments align with a typical `torch.nn.Conv2d` setup, they are suboptimal for our `GaborFilterSpatial`. Specifically, 2D convolution layers often utilize small kernel sizes to reduce computational load, such as `(3,3)` and `(5,5)` by convention, which are significantly insufficient within `GaborFilterSpatial`.

However, in practical applications of the GWT within the spatial domain, the kernel size of Gabor wavelets is typically limited to exclude pixels further than $3-5\ \sigma_{\text{eff}}$ (5.17) from the center, where the Gaussian envelope becomes negligible. When the overlay of the Gabor wavelet and image extends beyond the image border, cyclic boundary conditions are commonly applied in convolution instead of using `'zeros'`. This is because the cyclic boundary can replicate what happens in the frequency domain.

Therefore, we modified `padding_mode` to `'circular'`, representing the cyclic boundary and changed the default `kernel_size` to `None`. Now, users are not obligated to define a specific `kernel_size`. Instead, `GaborFilterSpatial` delegates the initialization of `kernel_size` to the `compute_spatial_kernel_size` function (5.18). Thus, `GaborFilterSpatial` can automatically compute the `kernel_size` based on the effective standard deviation of the wavelet with the largest wavelength (or lowest frequency) if a user does not specify a `kernel_size` by themselves:

$$\sigma^*_{\text{eff}} = \frac{\sigma}{k_{\max} \cdot k_{\text{fac}}^{\nu_{\max}-1}} \tag{5.17}$$

$$K_{\text{size}} = 2 \cdot \lceil \sigma^*_{\text{eff}} \cdot n_s \rceil + 1 \tag{5.18}$$

where σ^*_{eff} represents the effective standard deviation of the largest wavelet, $\nu_{\max}$ denotes the number of scales, n_s is the number of effective standard deviations away from the center (defaulting to 5), and K_{size} indicates the kernel size. Another consequential detail is that if the a user sets `padding` to `None`, `GaborFilterSpatial` will also automatically compute it as half of the kernel size for each dimension (i.e., $\lfloor K_{\text{size}}/2 \rfloor$).

```
def __update_padding_by_kernel_size(self):
   if isinstance(self.padding, str):
      self._reversed_padding_repeated_twice = [0, 0] * len(self.kernel_size)
      if self.padding == 'same':
         for d, k, i in zip(self.dilation, self.kernel_size,
             range(len(self.kernel_size) - 1, -1, -1)):
            total_padding = d * (k - 1)
            left_pad = total_padding // 2
            self._reversed_padding_repeated_twice[2 * i] = left_pad
            self._reversed_padding_repeated_twice[2 * i + 1] = (total_padding
                - left_pad)
   else:
      self._reversed_padding_repeated_twice =
          _reverse_repeat_tuple(self.padding, 2)
```

Listing 5.1: Private method `__update_padding_by_kernel_size` in `GaborFilterSpatial`

According to this computation, the kernel size often tends to be large. This is one of the primary reasons why performing GWT in the spatial domain is significantly slower than in the frequency domain. Nevertheless, with these adjustments, `GaborFilterSpatial` functions well and provides us with more precisely transformed images.

4. __update_padding_by_kernel_size As previously discussed, we outlined how we extracted the `set_wavelets` function, computed `kernel_size` automatically, and set convolution to operate with cyclic boundary conditions. However, these modifications come with some side effects. Directly manipulating certain internal states of `torch.nn.Conv2d` causes discrepancies where some internal states do not update in conjunction with others. During comprehensive testing, we identified an issue when `padding_mode='circular'` and the `kernel_size` was altered, leading to misbehavior within the inherited `_ConvNd`, resulting in incorrect padding updates if a user set `padding='same'` or `padding=None`. This is because the padding did not adjust in accordance with changes in the kernel size.

Understanding the root cause of this bug, we delved into the `torch.nn.Conv2d` source code and extracted a corresponding code snippet to rectify the `_reversed_padding_repeated_twice` private attribute:

Within `GaborFilterSpatial`, the `set_wavelets` method now invokes this private method `__update_padding_by_kernel_size` (shown in Listing 5.1) to ensure timely updates of relevant internal states.

5. in_channels From a theoretical perspective, the Gabor wavelet transform is not inherently designed to handle color images (i.e., when `in_channels > 1`). In response to such cases, `GaborFilterSpatial` issues a warning to users. However, if a user insists on using multi-channel images as input, we can also allow it. This is made possible by the fact that, in the Conv2d operation, the shape of `self.weight` should be `[out_channels, in_channels, *kernel_size]`, allowing us to easily duplicate `self.weight` along the second dimension `in_channels` times. This is a straightforward way to handle this edge case. Due to the nature of convolution, this is essentially equivalent to adding up the images along the `in_channels` dimension at the first step.

In future work, it might be interesting to apply the Gabor filters channel-wise instead of summing them.

5.2.3 pytoch_gabor.GaborFilterFrequency

The Gabor wavelet transform within the old `bob.ip.gabor.Transform` was conducted exclusively in the frequency domain. Consequently, it served as a fundamental reference for the development of our new `GaborFilterFrequency`. While the interface design of `GaborFilterSpatial` also provided a reference, it is important to note that there exists several key differences between `GaborFilterFrequency` and `GaborFilterSpatial`.

The generated Gabor wavelet family in `GaborFilterFrequency` is also stored in the attribute `self.weight`. However, the key difference lies in the instantiation process: `GaborFilterFrequency` necessitates the generation of the Gabor wavelet family within the frequency domain during instantiation, as opposed to the spatial domain. As per equation (5.4), the Gabor wavelet family produced in `GaborFilterFrequency` consists of real values rather than complex values.

`GaborFilterFrequency` solely inherits from `torch.nn.Module`, not `torch.nn.Conv2d`, which means it does not require parameters like `stride` or `padding` during instantiation. Nevertheless, `GaborFilterFrequency` needs an additional argument called `resolution` upon instantiation, which represents the resolution of the input image. In the case of executing Gabor wavelet transform in the frequency domain, the resolutions of the Gabor wavelet and the image must be identical. This requirement stems from using pixel-wise multiplication instead of convolution to execute GWT. Thus, we have customized its `forward` function accordingly. The implementation of this forward function is straightforward, following the steps outlined in equation (5.14): 1) apply `torch.fft.fft2` to the input image; 2) perform pixel-to-pixel multiplication between the input image and wavelets with batch broadcasting; 3) apply `torch.fft.ifft2` to the output. Hence, the resulting transformed image has the same shape `[B, J, H, W]` as that of `GaborFilterSpatial`.

In `GaborFilterFrequency`, we have also employed `@property` notation to align with terminologies from both PyTorch Module and Gabor wavelet transform. They are the same as those in `GaborFilterSpatial`.

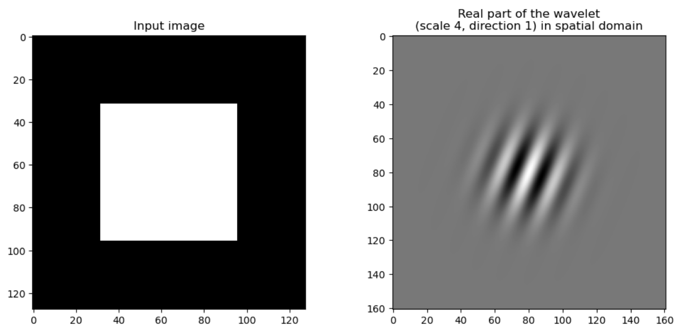
Problems and solutions

1. set_wavelets For `GaborFilterFrequency`, there is also a dedicated member function, `set_wavelets`, which is designed to generate a Gabor wavelet family in the frequency domain. This approach helps to avoid code logic coupling. Additionally, by maximizing the use of tensor operations, as exemplified in Listing 5.2 with functions like `meshgrid` and `expand`, we have parallelized this function to create the entire family of wavelets (weights) at once.

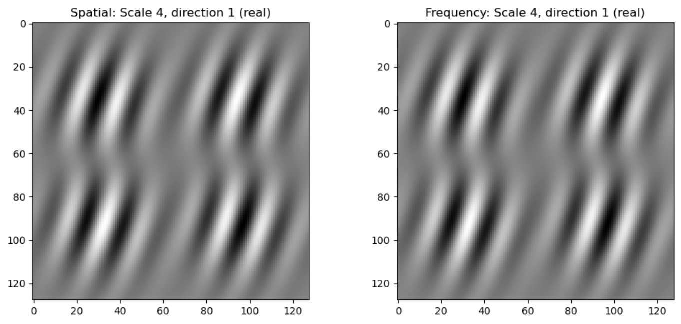
The `set_wavelets` function in `GaborFilterFrequency` involves less complexity compared to the one in `GaborFilterSpatial`, as we do not need to manage convolution-related edge cases, thus, we will not elaborate too much detail on this function here.

2. in_channels As mentioned before, Gabor wavelet transform does not support color images. However, in practice, we have adopted a simple way to address this special case in the spatial domain—simply duplicating the weights along `in_channels`. In such cases, ensuring the consistency in the interfaces and computational outcomes between `GaborFilterFrequency` and `GaborFilterSpatial` posed a challenge.

Upon analysis, we found that because `GaborFilterSpatial` inherits from `torch.nn.Conv2d`, it fundamentally includes an additional summation process along the `in_channels` di-



(a) Sample Input Image and the Chosen Wavelet



(b) Gabor Transformed Images

Figure 5.3: THE AGREEMENT OF GWT IN SPATIAL AND FREQUENCY DOMAINS. *(a) By selecting a sample input image and choosing a single Gabor wavelet at scale 4 and direction 1, (b) the Gabor transformed images in both spatial and frequency domains exhibit consistency.*

mension when `in_channels` of the convolution exceeds one. Therefore, in our implementation of `GaborFilterFrequency.forward()`, we allow the `in_channels` greater than 1 but then automatically sum the input image along the `in_channels` dimension. This approach aligns `GaborFilterFrequency.forward()` behavior with that of `GaborFilterSpatial.forward()`. Although users can specify `in_channels` greater than one for `GaborFilterSpatial` and `GaborFilterFrequency` during instantiation, we issue a runtime warning against this practice due to the lack of theoretical support.

Overall, this approach strikes a balance between theoretical principles and practical implementation, optimizing the trade-off in GWT.

5.2.4 Use Cases and Agreement

According to theory, whether executing GWT in the spatial or frequency domain, the transformed images should agree. Hence, demonstrating and verifying the equivalence between `GaborFilterSpatial` and `GaborFilterFrequency` are necessary.

In the use case `spatial_frequency_activation.ipynb`, we utilized a simple input image and applied GWT using both `GaborFilterSpatial` and `GaborFilterFrequency` under default parameter settings. By choosing a specific Gabor wavelet in the family, Figure 5.3 visually displays their transformed images respectively, illustrating their consistency.

In addition to visualization in a use case, we have designed thorough test cases to confirm this consistency. These tests encompass three different categories, each reflecting different scenarios of the input images:

- `batch_size = 1, in_channels = 1;`
- `batch_size > 1, in_channels = 1;`
- `batch_size > 1, in_channels > 1;`

The first two categories of tests successfully pass the `torch.allclose()` assertion, using very low `rtol` and `atol` thresholds. Furthermore, the differences between the transformed images in the spatial and frequency domains are mainly around the borders of the images, due to padding in the spatial domain. If we ignore the borders of the transformed image when testing, the differences will be even smaller:

```
assert torch.allclose(freq_transformed, spat_transformed, rtol=1e-05,
    atol=1e-3)
# If we ignore the borders
assert torch.allclose(freq_transformed[:, :, 20:108, 20:108],
                  spat_transformed[:, :, 20:108, 20:108], rtol=1e-05,
                      atol=1e-4)
```

However, with multi-channel input images, we had to significantly increase the `rtol` and `atol` levels (`atol=0.9`, `rtol=0.5`) to make the tests pass. The primary reason is that `torch.nn.Conv2d` does not support `'circular'` padding mode when `in_channels > 1`.

Based on the demonstrations and comprehensive testing conducted, we can conclude that our implementations of `GaborFilterSpatial` and `GaborFilterFrequency` are correct. Whether performing GWT in the spatial or frequency domain, the results of the transformed images remain substantially consistent.

```
self.out_channels = self.number_of_scales * self.number_of_directions
# To ensure compatibility between frequency and the device in similarity,
# we need to wrap it within a Parameter.
self.frequency = torch.nn.Parameter(
   data=compute_frequency_centers(self.number_of_scales,
       self.number_of_directions, self.k_max, self.k_fac),
   requires_grad=False)
height, width = self.kernel_size

# To ensure that coordinates are centered
x = torch.arange(-(width // 2), (width + 1) // 2)
y = torch.arange(-(height // 2), (height + 1) // 2)
yy, xx = torch.meshgrid(y, x, indexing="ij")
y_omega, x_omega = (2 * torch.pi / height) * yy, (2 * torch.pi / width) * xx
omega_squared = x_omega ** 2 + y_omega ** 2
sigma_squared = self.sigma ** 2

self.weight = torch.nn.Parameter(
   data=torch.empty(self.number_of_scales * self.number_of_directions,
       self.kernel_size[0], self.kernel_size[1]),
   requires_grad=False)

k_y = self.frequency[:, 0, None, None].expand(-1, self.kernel_size[0],
    self.kernel_size[1])
k_x = self.frequency[:, 1, None, None].expand(-1, self.kernel_size[0],
    self.kernel_size[1])
omega_minus_k_squared = (x_omega - k_x) ** 2 + (y_omega - k_y) ** 2
k_squared = k_x ** 2 + k_y ** 2
omega_sqrd_plus_k_sqrd = omega_squared + k_squared
first_part = torch.exp(-sigma_squared * omega_minus_k_squared / (2 *
    k_squared))
second_part = torch.exp(-(sigma_squared * omega_sqrd_plus_k_sqrd) / (2 *
    k_squared))

self.weight[:, yy, xx] = first_part - second_part
```

Listing 5.2: Parallel implementation for initializing Gabor wavelets in the frequency domain

Chapter 6

Gabor Jets and Activation

6.1 Theoretical Analysis

The concept of jet was introduced by Buhmann and Lange (1989), a Gabor jet $\mathcal{J}^{\mathcal{I}}$ of a image $\mathcal{I}$ is created by concatenating the responses of all Gabor wavelets at a specific position $\vec{t}$ in the Gabor transformed image $\mathcal{T}$ into a single vector (Figure 6.1). The elements of $\mathcal{J}^{\mathcal{I}}$ are addressed by the index j of the corresponding Gabor wavelet $\psi_{\vec{k}_j}$:

$$\left(\mathcal{J}^{\mathcal{I}}(\vec{t})\right)_j = \mathcal{T}_{\vec{k}_j}(\vec{t}) \tag{6.1}$$

In order to integrate it with the subsequent stage of the Gabor wavelet processing pipeline, which involves computing similarity (Chapter 8). Activation is applied to the extracted jet, which converts the complexed-valued Gabor jet into the absolute value part:

$$a_j = \left|(\mathcal{J})_j\right| \tag{6.2}$$

and the phase part:

$$\phi_j = \arg\left[(\mathcal{J})_j\right] \tag{6.3}$$

6.2 Implementation

In our new `pytorch_gabor` package, the corresponding implementations for the theoretical components can be found in the `Jet` and `GaborFilterAct` classes.

`GaborFilterAct` applies activation function to the transformed image for all images within a batch and at all positions, producing a `torch.Tensor` as output. In contrast, `Jet` extracts a Gabor jet from a specified position of all images within a batch, then generates an encapsulated `Jet` object. As a result, `GaborFilterAct` seamlessly integrates into PyTorch networks, whereas `Jet` does not.

The `Jet` and `GaborFilterAct` classes exhibit several common features: both utilize activation functions to convert the complex-valued input derived from Gabor filters into Euler form, encompassing absolute value and phase. Additionally, normalization for the absolute value is optional for both classes. These classes operate on Gabor wavelet transformed images as input, demonstrating the capability to process all images within a batch concurrently.

However, distinctions arise in their outputs and functionalities: `GaborFilterAct` facilitates complete parallelization, enabling concurrent processing for all images within the batch and all positions. On the other hand, `Jet` is confined to extracting Gabor jets from individual positions

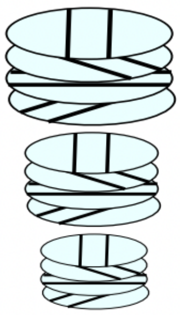


Figure 6.1: SIMPLIFIED DEPICTION OF A GABOR JET (GÜNTHER, 2012). *The size of circles stands for scale, the different orientations of stride show different directions.*

```python
def normalize(self) -> None:
      self.jet[:, 0, :] = torch.nn.functional.normalize(self.jet[:, 0, :],
          dim=-1)
```

Listing 6.1: Normalization for absolute value of Jet

for every image in the batch in a single operation. While the advantage of `GaborFilterAct` in parallelization has been discussed earlier, the strength of `Jet` lies in its additional and useful member functions. For instance, it provides the capability to manually extract a single jet, a functionality not available with `GaborFilterAct`.

6.2.1 pytoch_gabor.Jet

As mentioned earlier, the shape of the transformed image is `[B, J, H, W]`, where `B` represents the batch size, `J` represents different Gabor wavelets, and `H` and `W` are height and width of the image. The `Jet` class is responsible for extracting Gabor jets from a specific position in the transformed image, yielding a complex-valued `torch.Tensor` with a shape of `[B, J]`. Subsequently, activation is applied to the extracted jet to integrate it with the subsequent stage of the pipeline which leverages both the absolute value and phase of the complex values. The output of activation is a real-valued `torch.Tensor` with a shape of `[B, 2, J]`. This structure takes into account `Jet` in the old package that processed a single image with the shape `[2, J]`, and the batch dimension comes first in standard PyTorch networks.

As mentioned earlier, normalization for the absolute value is optional for the `Jet` class. The `Jet` class provides a method called `normalize` specifically designed to apply L_2 normalization to the absolute part of the jet along dimension `J`. We implement this normalization process entirely relying on PyTorch's built-in methods (Listing 6.1). This function is crucial because the `Similarity` class assumes that all jets are normalized.

As outlined above, the `Jet` class takes input from Gabor wavelet transformed images in tensor form, extracts Gabor jets from a specific position, and applies activation. This class utilizes tensor operations to handle extraction and activation, achieving parallelization over all images within a batch naturally through tensor operations. However, it is important to note that the `Jet` class is designed to extract jets only from one specified position in transformed images, and it cannot extract Gabor jets from multiple positions in one shot.

problems and solution

The `bob.ip.gabor.Jet` class features multiple constructors, a functionality not directly supported by Python. To overcome this limitation, we strategically implemented one constructor as `init` method and other constructors as class methods, streamlining their integration into the class structure (Table 6.1). To enhance the simplicity of the code, the `init` function serves primarily for encapsulation, with other class methods invoking it. This design choice streamlines the code structure and promotes a clear separation of concerns, contributing to a more maintainable and comprehensible implementation. The constructors in `bob.ip.gabor.Jet` and corresponding methods in `pytorch_gabor.Jet` are outlined below:

1. The constructor 1 in `bob.ip.gabor.Jet` creates `Jet` from jet data. In our new implementation, we replace this constructor with `init` method to streamline the process.

   When the input `data` is complex-valued and has a shape of `[B, J]`, the `init` method transforms the complex values into their absolute value and phase, storing them accordingly. This mirrors the functionality of the constructor in the old Bob package that creates a jet from a list of complex values.

   If the input data is of type float and has a shape of `[B, 2, J]`, the init method directly saves the input as jet data.

2. The constructor 2 in `bob.ip.gabor.Jet` is utilized to create a `bob.ip.gabor.Jet` object filled with zeros of a specified length. In our new package, we provide a corresponding method called `zeros` to achieve the same result, creating a Gabor jet with all zeros. Given our approach to handling a batch of images in the new package, the `batch_size` is also a parameter for this method.

3. The constructor 3 in `bob.ip.gabor.Jet` is responsible for creating a `bob.ip.gabor.Jet` object by averaging the complex forms of the provided Gabor jets within the list. In our new package, we offer a comparable method named `average` to achieve the same functionality.

4. The constructor 4 in `bob.ip.gabor.Jet` is responsible for extracting a Gabor jet from the provided Gabor transformed image at the specified location. In our new package, we offer a corresponding method named `extract` to perform the same operation. As discussed earlier, the new package is designed to handle batched images. Consequently, the `extract` method can efficiently extract Gabor jets from a batch of images.

5. The constructor 5 in `bob.ip.gabor.Jet` is designed to load a Gabor jet from an HDF5 file, relying on the `bob.io.base.HDF5File` module. The new `load` method now fully relies on `torch.load` and no longer depends on `bob.io.base.HDF5File`, as illustrated in Listing 6.2.

.

The use of the `@classmethod` annotation, coupled with forward declarations denoting the return type as `-> "Jet"`, provides an organized and anticipatory structure for these constructor functionalities.

Remarkably, within the `Jet` class, we have also incorporated methods such as `save_jets` and `load_jets`, effectively internalizing these processes and reducing reliance on external functions.

```
def load(cls, f: FILE_LIKE, **kwargs) -> "Jet":
    # f is a file-like object (has to implement write and flush) or a string or
        os.PathLike object containing a file name.
    data = torch.load(f=f, **kwargs)
    return cls(data, normalize=False)
```

Listing 6.2: The method `Jet.load`

ID	The constructors of bob.ip.gabor.Jet	The corresponding method in pytorch_gabor.Jet
1	bob.ip.gabor.Jet (complex: Array, normalize: bool = True)	def __init__(self, data: Tensor, normalize: bool = True)
2	bob.ip.gabor.Jet (length: int = 0)	@classmethod def zeros(cls, length: int, batch_size: int = 1) -> "Jet"
3	bob.ip.gabor.Jet (to_average: List["Jet"], normalize: bool = True)	@classmethod def average(cls, to_average: List["Jet"], normalize: bool = False) -> "Jet"
4	bob.ip.gabor.Jet (trafo_image: Array, position: Tuple[int,int], normalize: bool = True)	@classmethod def extract(cls, trafo_image: Tensor, position: Union[Tuple[int, int], Tensor], normalize: bool = True) -> "Jet"
5	bob.ip.gabor.Jet(hdf5: "bob.io.base.HDF5File")	@classmethod def load(cls, f: FILE_LIKE, **kwargs) -> "Jet"

Table 6.1: THE COMPARISON OF `BOB.IP.GABOR.JET` AND `PYTORCH_GABOR.JET`. *This table shows the constructors of `bob.ip.gabor.Jet` and corresponding methods in `pytorch_gabor.Jet`.*

6.2.2 pytoch_gabor.GaborFilterAct

The `GaborFilterAct` class is specifically crafted as an activation layer for Gabor filters, inheriting from `torch.nn.Module`. This class can convert the complex values of transformed images into absolute values and phases, allowing the selection of the desired part as the output.

Derived from `torch.nn.Module`, the class employs tensor operations to manage the activation process. This allows the class to activate the transformed images for all images in a batch and all positions in a single operation.

This class provides flexible control over the output type, output shape, and normalization through various parameters during its setup:

The `out_type` parameter defines the desired output type and offers three choices: `abs`, `phase`, or `abs_phase`. Each option corresponds to the absolute value, the phase, or a combination of both in the complex output. The default configuration is `abs_phase`, which means returning both "absolute and phase" values, reflecting the common use of both aspects in most Gabor jet similarity functions.

The stack_abs_phase parameter helps to control the shape of the output when out_type is configured as "abs_phase." In this configuration, it intelligently stacks the absolute and phase components along dimension 1. If set to false while out_type remains "abs_phase", the forward method will output a tuple (abs_part, phase_part). The default value for this parameter is true, aligning with PyTorch's network structure, where the data output is a tensor rather than a tuple. This adherence to standards enhances compatibility and facilitates computational optimization.

The output shape, presented in tensor form, for both GaborSpatial and GaborFreq (which also functions as the input for GaborFilterAct), is [B, J, H, W]. If the out_type is set to "abs_phase" and stack_abs_phase=True, the resulting output tensor of GaborFilterAct will have shape [B, 2, J, H, W], if the out_type is set to abs or phase, the resulting output will have shape [B, J, H, W].

The normalize parameter governs whether the absolute part of the output undergoes L_2 normalization. This parameter holds significance only when the "out_type" is specified as "abs" or "abs_phase". If the activated jets are to be utilized in the Similarity class later, normalization becomes necessary. This is due to the fact that the jets provided as input to the Similarity class are already normalized by default (Chapter 8).

Problems and solution

In the context of the project's inception, the main goal of GaborFilterAct was to transform complex responses from Gabor wavelets into Euler form. However, the initial interface design overlooked the inclusion of the normalize argument. Subsequent testing highlighted the essential nature of normalization. This modification played a crucial role in fulfilling the normalization requirement in Similarity and aligning its functionality with that of the 'Jet' in this specific context.

6.2.3 pytoch_gabor.JetStatistics

This class is not integrated into the Gabor wavelet processing pipeline; instead, it functions as an experimental analysis tool. Its essential functions and interfaces closely resemble those of bob.ip.gabor.JetStatistics. The primary distinction lies in its capability for batch processing, which is similar to Jet and GaborFilterAct. Therefore, we will not delve into the details of this aspect in the current discussion. We only show parameter of init method here:

1. The jets parameter accommodates a list of "Jet" objects, representing the entities subjected to analysis.
2. The gwt parameter accepts either "GaborFilterFrequency" or "GaborFilterSpatial." This parameter plays a pivotal role in the computation of disparity vectors.

Chapter 7

Grid Graph

7.1 Theoretical Analysis

The grid graph concept involves extracting a memorizable object from an image in the form of a model graph. This is accomplished by overlaying a rectangular grid of points onto the object and recording the corresponding jets, as proposed by Lades et al. (1993).

Remarkably, the grid graph exhibits superior recognition accuracy compared to the face graph (Günther et al., 2012), even when the number of nodes in the grid is fewer than that in the face graph (Günther et al., 2012).

7.2 Implementation

In the new `pytorch_gabor` package, the corresponding implementations for the theoretical components reside in the `Graph` and `GridExtract` classes.

`GridExtract` takes a batch of activated and transformed images as input, then produce a `torch.Tensor`. In contrast, `Graph` only receives a batch of transformed images as input, generating a list of `Jet` objects through the `Jet.extract` method, with activation occurring in `Jet`. Consequently, `GridExtract` seamlessly integrates into PyTorch networks, while `Graph` lacks this capability.

Both classes exhibit similarities as they play a role in constructing a rectangular grid graph using Gabor wavelet transformed images, following the theoretical analyses provided earlier. Following the graph construction, both classes proceed to extract Gabor jets from the points of the rectangular grid graph. During the jet extraction process, parallel processing is achieved over all images in a batch through efficient `torch.Tensor` operations.

However, there are distinctions between them: `GridExtract` enables complete parallelization, efficiently extracting Gabor jets for all images in a batch and every position in the grid graph in parallel. Conversely, `Graph` supports only partial parallelization, achieving parallelization for images in a batch. The benefit of `GridExtract` in parallelization has been discussed above. The advantage of `Graph` lies in its additional, useful member functions. For example, `Graph` allows users to create custom graphs, not limited to grid graphs. This flexibility is not found in `GridExtract`.

7.2.1 pytoch_gabor.Graph

The `Graph` class serves as a tool for extracting Gabor jets from Gabor wavelet transformed images.

```python
def load(cls, f: FILE_LIKE, **kwargs) -> "Graph":
    # f is a file-like object (has to implement write and flush) or a string or
        os.PathLike object containing a file name.
    data = torch.load(f=f, **kwargs)
    return cls(data)
```

Listing 7.1: The method `Graph.load`

The `Graph` class supports various methods to generate graphs. This class can then utilize the `Graph.extract` method to extract Gabor jets for all images in a batch and all positions in the graph. As outlined above, `Graph` supports only partial parallelization, achieved for images in a batch. The extraction of different points on the entire graph relies on a for loop, as the `Graph.extract` method depends on `Jet.extract`, which can only extract jets from a single position at a time. Then the `Graph.extract` method returns a list of `Jet`. The reliance on `Jet.extract` and the chosen output format are crafted in this manner to uphold consistency with the original Bob implementation.

Another noteworthy aspect is that the `Graph` class stores the absolute positions of its nodes (points in the graph). Ensuring that these nodes, when used in the `extract` method, stay within the image boundaries is crucial to prevent errors.

Problems and solutions

The `bob.ip.gabor.Graph` class encompasses multiple constructors, a feature not directly supported by Python. Similar to `pytorch_gabor.Jet`, we address this limitation by implementing one constructor as `init` method and other constructors as class methods, and the `init` method serves same purpose, with other class methods calling it. This approach contributes to a more maintainable and comprehensible implementation. The constructors in `bob.ip.gabor.Graph` and corresponding methods in `pytorch_gabor.Graph` are discussed below:

1. Constructor 1 in `bob.ip.gabor.Graph` specify the positions as a list of tuples, which are then used to generate a `Graph` object. Similarly, the `init` method in the new package accepts a list of tuples or a `torch.Tensor` with the shape `[L, 2]` as input, creating a `Graph` object accordingly.

2. The `bob.ip.gabor.Graph` class in the old package provides two different parameter sets to generate a regular rectangle graph: constructor 2 and constructor 3. In the new package, we follow the same approach and use the `create_by_border` and `create_by_eyes` methods to achieve the same functionality.

3. The `bob.ip.gabor.Graph` class in the old package loads a `bob.ip.gabor.Graph` object from a specific `bob.io.base.HDF5File` file through constructor 4. In our new implementation, we have a corresponding `load` method that serves the same purpose. The new `load` method now exclusively relies on `torch.load` and no longer depends on `bob.io.base.HDF5File`, as illustrated in Listing 7.1.

.

We utilize the `@classmethod` annotation and employ forward declarations for return types. The notation `-> "Graph"` serves as a forward declaration, signifying the return of an instance of the '`Graph`' class.

ID	**The constructors of bob.ip.gabor.Graph**	**The corresponding method in pytorch_gabor.Graph**
1	bob.ip.gabor.Graph (self, nodes: List[Tuple[int, int]])	def__init__(self, nodes: Union[List[Tuple[int, int]], Tensor])
2	bob.ip.gabor.Graph (first: int, last: int, step: int)	@classmethod def create_by_border(cls, top_left: Tuple[int, int], bottom_right: Tuple[int, int], step: Tuple[int, int]) -> "Graph"
3	bob.ip.gabor.Graph (righteye: int, lefteye: int, between: int , along: int, above: int, below: int)	@classmethod def create_by_eyes(cls, righteye: Tuple[int, int], lefteye: Tuple[int, int], between: int, along: int, above: int, below: int) -> "Graph"
4	bob.ip.gabor.Graph (hdf5: "bob.io.base.HDF5File")	@classmethod def load(cls, f: FILE_LIKE, **kwargs) -> "Graph"

Table 7.1: THE COMPARISON OF `BOB.IP.GABOR.GRAPH` AND `PYTORCH_GABOR.GRAPH`. *The table shows constructors of* `bob.ip.gabor.Graph` *and corresponding methods in* `pytorch_gabor.Graph`.

7.2.2 pytoch_gabor.GridExtract

The `GridExtract` class is designed for the extraction of jets from a rectangle grid graph within PyTorch networks, utilizing a user-defined `stride` parameter; this parameter determines the distance between neighboring points in the graph in two directions. It is notable that this class starts indexing the grid always at (0,0). In future work, it may be meaningful to enable users to set a different starting point and a end point for indexing.

This class is designed as a subclass of `torch.nn.Module`, in comparison to Graph, the functionality of this class is straightforward. It solely implements a function that extracts specific entries from a given tensor based on a specified stride, and it lacks many additional member functions. Nevertheless, `pytoch_gabor.GridExtract` is convenient, making it suitable for use in standard PyTorch networks. For a comprehensive understanding of the network's overall functionality, please refer to Chapter 9.

Chapter 8

Similarity Functions and Disparity

8.1 Theoretical Analysis

The Gabor jet serves as a repository of texture information and finds utility in applications such as landmark detection and disparity estimation. When comparing Gabor jets extracted from different images, various similarity measures come into play. Here, assuming all jets are already normalized, we introduce the similarity functions utilized in our code:

1. Scalar product:

$$S_a(\mathcal{J}, \mathcal{J}') = \sum_j a_j a_{j'} \tag{8.1}$$

2. Canberra distance:

$$S_C(\mathcal{J}, \mathcal{J}') = \sum_j \frac{a_j - a'_j}{a_j + a'_j} \tag{8.2}$$

3. Abs phase:

$$S_\phi(\mathcal{J}, \mathcal{J}') = \sum_j a_j a'_j \cos(\phi_j - \phi'_j) \tag{8.3}$$

4. Disparity:

$$S_D(\mathcal{J}, \mathcal{J}') = \sum_j a_j a'_j \cos(\phi_j - \phi'_j - \vec{k}_j^T \vec{d}) \tag{8.4}$$

5. PhaseDiff:

$$D_P(\mathcal{J}, \mathcal{J}') = \sum_j \cos(\phi_j - \phi'_j - \vec{k}_j^T \vec{d}) \tag{8.5}$$

6. PhaseDiffPlusCanberra:

$$S_{P+C}(\mathcal{J}, \mathcal{J}') = \sum_j \left[\frac{a_j - a'_j}{a_j + a'_j} + \cos(\phi_j - \phi'_j - \vec{k}_j^T \vec{d}) \right] \tag{8.6}$$

In (8.4), (8.5), (8.6), $\vec{k}_j^T$ represents the frequency center of the jth wavelet, and $\vec{d}$ is the disparity vector. The disparity vector is calculated as suggested by Günther (2012).

The disparity vector is calculated by following rules iteratively (we estimate them from highest frequency $\zeta_{\max}$ to lowest frequency ζ):

$$\vec{d_\zeta} = \Gamma_\zeta^{-1}\Phi_\zeta \tag{8.7}$$

with

$$\Gamma_{\zeta;h,v} = \sum_{j=\zeta\nu_{\max}}^{\zeta_{\max}\nu_{\max}-1} k_{j;h}k_{j;v}a_j a'_j \tag{8.8}$$

$$\Phi_{\zeta;h} = \sum_{j=\zeta\nu_{\max}}^{\zeta_{\max}\nu_{\max}-1} a_j a'_j k_{j;h}(\phi_j - \phi'_j - m_j 2\pi) \tag{8.9}$$

$$m_j = \left\lfloor \frac{\phi_j - \phi'_j - \vec{k}_j^T \vec{d}_{\zeta+1}}{2\pi} \right\rceil \tag{8.10}$$

The final disparity vector $\vec{d} = \vec{d_0}$ is simply the estimated disparity using all Gabor wavelet levels.

For graph similarity assessment, we calculate the mean across all graph nodes (Günther, 2012), L means different positions of the graph.

$$S_{[\cdot]}^G(G_1, G_2) = \frac{1}{L}\sum_{l=0}^{L-1} S_{[\cdot]}(J_l^1, J_l^2) \tag{8.11}$$

$S_{[\cdot]}$ can be different kinds of similarity functions:

$$S_{[\cdot]} \in \{S_a, S_C, S_\phi, S_D, S_P, S_{P+C}...\} \tag{8.12}$$

8.2 Implementation

In the initial version, the implementation followed the theoretical analysis and referred to the source code of `bob.ip.gabor.Similarity` to incorporate its core functionality. This entails the ability to take two jet objects as input and produce the correct result.

The class `pytorch_gabor.Similarity` in our new package inherits from `torch.nn.Module`, considering to operate within the Gabor wavelet processing pipeline. Additionally, the class `Similarity` is explicitly designed to accommodate input in the form of either two lists of jet objects or two tensors of jets. This flexibility corresponds to the utilization of two distinct Gabor wavelet processing pipelines.

Another noteworthy aspect is parallelization; this class is tasked with computing similarity between jet tensors with shape `[B, 2, J, ...]` in one shot.

8.2.1 pytoch_gabor.Similarity

The `Similarity` class is versatile, designed for computing various similarity functions between Gabor jets. It is important to note that all input Gabor jets to compute similarity functions are expected to be pre-normalized.

The `similarity_type` parameter determines the specific similarity function calculated by the class.

The types "`ScalarProduct`" (8.1) and "`Canberra`" (8.2) utilize the absolute values of the Gabor jets to calculate the similarity function.

The types "`AbsPhase`" (8.3), "`Disparity`" (8.4) and "`PhaseDiffPlusCanberra`" (8.6) directly utilize both the absolute values and the phase values of Gabor jets to calculate the similarity function.

The type "`PhaseDiff`" (8.5) requires to compute disparity vector, the computing of disparity vector utilizes both the absolute values and the phase values of Gabor jets.

In (8.4), (8.5), and (8.6), we require $\vec{k}_j^T$ to calculate disparity vector, then use disparity vector to calculate similarity. The frequency center $\vec{k}_j^T$ corresponds to the `frequency` parameter in Gabor filters (referred to as parameter `transform`, which should be an instance of `GaborFilterSpatial` or `GaborFilterFrequency`). Therefore, the `transform` parameter is essential when parameter `similarity_type` is set to "`Disparity`" or "`PhaseDiff`" or "`PhaseDiffPlusCanberra`".

When the parameter `similarity_type` is set to "`Disparity`" (8.4), "`PhaseDiff`" (8.5), or "`PhaseDiffPlusCanberra`" (8.6), the disparity vector (estimated by (8.7), (8.8), (8.9) and (8.10)) is essential for similarity computation.

`Similarity` can be employed in two distinct Gabor wavelet processing pipelines: either within a pure object-oriented Gabor wavelet processing pipeline (as illustrated in Figure 9.1) or as part of a pipeline that relies entirely on `torch.nn.Module` (as illustrated in Figure 9.2).

Within the pipeline, which relies entirely on `torch.nn.Module`, the `Similarity` class utilizes the `forward` method to compute the similarity between jets. The `forward` method takes two previously extracted and activated jet tensors as input. As outlined above, the shape of such input jet tensors is `[B, 2, J, H, W]`, where `[2, J]` represents a single Gabor jet from a single image, `[B]` stands for all images in the batch, and `[H, W]` corresponds to height and width of images. The `forward` method calculates the similarity map for every image in the batch, representing the similarity between every pair of Gabor jets at all positions for each image in the batch; the overall result has shape `[B, H, W]`. Parallelization is automatically achieved through tensor operations.

In the object-oriented pipeline, the `similarity` method of the `Similarity` class is employed. Aligned with its object-oriented design, this method accepts two lists of `Jet` objects as input.

To optimize the code and eliminate redundancy, the `similarity` method stacks the two lists of `Jet` (with shape `[B, 2, J]`) along the last dimension, resulting in the `stacked_jet` with a shape of `[B, 2, J, L]`. Subsequently, these two instances of `stacked_jet` are provided as input to the `forward` method. The `similarity` method then computes the average similarity of the result of the internally called `forward` method (the result had shape `[B, L]`) along the `[L]` dimensions, yielding the graph similarity as defined in (8.11) with shape `[B]`.

Problems and solution

1. broadcast problem and Parallelization As explained earlier, in the pipeline that relies entirely on `torch.nn.Module`, two activated and extracted jet tensors with shape `[B, 2, J, H, W]` are input for `forward` method. A single Gabor jet across images in a batch has a shape of `[B, 2, J]`. In the pure object-oriented pipeline, two lists of `Jet` objects are used as input for the `similarity` method, resulting in a `stacked_jet` with the shape of `[B, 2, J, L]`. If required, all three scenarios will call the `disparity` method within the `forward` method to calculate the disparity vector.

Based on the analysis above, it is imperative to compute the disparity vector in cases where the jet tensors have a shape of `[B, 2, J, ...]`, where `...` can take on values such as `[]`, `[L]`, or `[H, W]`, resulting in a non-fixed shape for the jet tensors. This deviates from conventional PyTorch network approaches.

When updating disparity, the `Similarity` class stores the dimension of `[...]` of jet tensors in `self.dims_of_jets`.

Subsequently, upon updating the disparity vector, the shape of certain parameters will adjust based on `self.dims_of_jets`. Specifically, the shape of `gamma` (corresponding to Γ in (8.8)) and `phi_yx` (corresponding to Φ in (8.9)) varies in accordance with the shape of the parameter `self.dims_of_jets`.

Then we update `gamma`, `phi_yx`, `self._disparity` and `nL` (corresponds to m_j in (8.10)) for every wavelets according to (8.7), (8.8), (8.9) and (8.10). Commencing with the last wavelet, characterized by the highest frequency and largest direction angle, we subsequently update Γ, Φ and m_j for all wavelets sharing the same frequency in reverse order (from large direction angle to small direction angle). When all wavelets that share the same frequency have been traversed, we update $\vec{d}$ for this frequency. This process continues in reverse order, updating these values for the next smaller frequency until all wavelets have been traversed. The resulting $\vec{d}$ represents the disparity vector.

The actual disparity calculation process is only related to the `[2, J]` dimension of the jet tensors. In principle, parallelization can be achieved on `[B]` (dimension of batch) and `[...]` (dimension of jets distribution). However, certain issues arise in this context.

When updating disparity, we need to perform a "broadcasting" operation as follows:

```
[B, 1, ...] * [2, 1] => [B, 2, ...]
```

However, according to the documentation of Pytorch:[1] Two tensors are "broadcastable" if the following rules hold:

1. Each tensor has at least one dimension.

2. When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

In our scenario, "`...`" represents different positions, and it is positioned at the trailing dimension, determined by the shape of the jet tensor. This arrangement poses a challenge for direct broadcasting.

To address this challenge, we used a technique called `torch.einsum`. This method removes the need to manually adjust the dimensions of the corresponding `torch.Tensor` for different shapes of "`...`," achieving our original goal. It also makes the code easier to understand. With this method, we can compute the disparity vector for tensors of different shapes, as long as their shapes follow the pattern `[B, 2, J, ...]`, which also helps with parallelization using tensor operations. The full process of calculating disparity vector is shown in Listing 8.1.

2. device conflict in GPU While trying to execute on the GPU, we ran into a problem where there was a conflict between devices. Specifically, it pointed out that some parameters were not on the same device (CUDA) as other tensors, like the input image. The usual fix for this kind of issue is to put them in `torch.nn.Parameter`. However, there is a complication, these variables have

[1] https://pytorch.org/docs/stable/notes/broadcasting.html

```
self._confidences = jets1[:, 0] * jets2[:, 0] # absolute part | shape [B, J,
    ...]
self._phase_differences = self.adjust_phase(jets1[:, 1] - jets2[:, 1]) # phase
    part | shape [B, J, ...]

# The computing of self._confidences and self._phase_differences are in
    different methods; when calculating disparity, this method is also called
    to calculate these two values for calculating disparity vector; the actual
    structure of the class is different from what we show here. We just show
    how we calculate the disparity vector here.
# The method self.adjust_phase adjust the input phase to [-pi, pi].

self.dims_of_jets = jets1.shape[3:] # can be [], [L] or [H, W]

gamma = torch.zeros(self.batch_size, 2, 2, *self.dims_of_jets, device=device)
phi_yx = torch.zeros(self.batch_size, 2, *self.dims_of_jets, device=device)
self._disparity = torch.zeros(self.batch_size, 2, *self.dims_of_jets,
    device=device)

for j in range(self.transform.number_of_wavelets - 1, -1, -1):
   kj_yx = self.transform.frequency[j, :] # The center of frequency
   conf = self._confidences[:, j]
   diff = self._phase_differences[:, j]
   gamma += torch.einsum('bij..., bij... -> bij...',
       conf.unsqueeze(1).unsqueeze(1), torch.outer(kj_yx, kj_yx).unsqueeze(0))
       # update gamma for every wavelet
   nL = torch.round(((diff - torch.einsum('bi..., bi... -> bi...',
       self._disparity, kj_yx.unsqueeze(0) ).sum(dim=1)) / (2 * torch.pi))) #
       update m_j for every wavelet

   phi_yx += torch.einsum('bi..., bi... -> bi...', ((diff - nL * 2.0 *
       torch.pi) * conf).unsqueeze(1), kj_yx.unsqueeze(0)) #update Phi for
       every wavelet

   if j % self.transform.number_of_directions == 0:.
      gamma_inverse = torch.inverse(
         gamma.permute(0, *range(3, len(gamma.shape)), 1, 2)
      ).permute(0, 1 + len(self.dims_of_jets), 2 + len(self.dims_of_jets),
         *range(1, len(self.dims_of_jets) + 1))

      self._disparity = torch.einsum('bij..., bjk... -> bik...',
          gamma_inverse, phi_yx.unsqueeze(dim=2) ).squeeze(dim=2) # [B, 2,
          ...] # update disparity for every frequency.
```

Listing 8.1: The calculation of disparity vectors

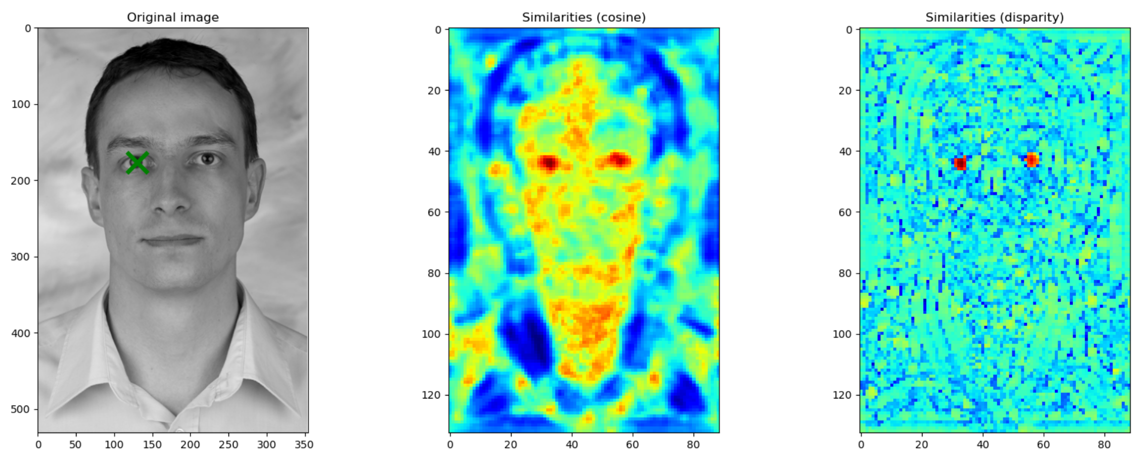


Figure 8.1: SIMILARITY MAP ASSOCIATED WITH THE RIGHT EYE POSITION. *This figure displays the similarity map for types "Disparity" and "ScalerProduct" when using the jet at the right eye position as a reference.*

varying shapes but `torch.nn.Parameter` requires a fixed shape. So, to tackle this problem, we manually specified the `device` for these parameters.

8.3 Use Case

In theory, the results of similarity computation, whether performed with the `Jet` object or with tensors, should align.

In the use case `similarity.ipynb`, we employed two different approaches, computing similarity with two lists of `Jet` objects and two tensors of jets. For each approach, we utilized a simple sample as the input image and extracted the Gabor jet at the right eye location as a reference. Subsequently, every fourth pixel was extracted, and we computed the similarity map. Finally, we displayed the similarity map, illustrating the two approach's consistency.

The approach with `Jet` showed in Listing 8.2 includes 1) extracting reference `Jet` from right eye position, 2) computing the similarity field over the entire image and generating the similarity map using both the "`ScalarProduct`" and "`Disparity`" similarity measures.

It is noteworthy that every fourth pixel from the image was selected to compute the similarity map, consistent with the old use case.

We compare two ways of measuring similarity ("`Disparity`" and "`ScalarProduct`" a.k.a "cosine") in the use case. Both methods focus on the eyes, specifically the right eye's Gabor jet (Figure 8.1).

We obtain the same `cos_image` and `disp_image` as in the old use case. It is noteworthy that the `disparity` method can emphasize only the eye areas more effectively. Next, we will introduce another method that exclusively utilizes tensors, achieving similar `cos_image` and `disp_image` results.

As depicted in Listing 8.3, The approach with tensors includes: 1) employing the `GridExtract` module to extract every fourth pixel, essentially creating jets without the need for encapsulation within `Jets`, 2) using `GaborFilterAct` to transform complex values to absolute values and

```
gabor_filter = GaborFilterFrequency(in_channels=1, resolution=image.shape[-2:])
trafo_image = gabor_filter(image)

# extract reference
pos = (177, 131) # eye
eye_jet = Jet.extract(trafo_image, pos)

# compute similarity field over the whole image
cos_sim = Similarity(similarity_type="ScalarProduct")
disp_sim = Similarity(similarity_type="Disparity", transform=gabor_filter)
cos_image = torch.zeros(1, 1, (image.shape[-2])//4+1, (image.shape[-1])//4+1)
disp_image = torch.zeros(1, 1, (image.shape[-2])//4+1, (image.shape[-1])//4+1)

# compute similarity map
for y in range(0, image.shape[-2], 4):
   for x in range(0, image.shape[-1], 4):
        image_jet = Jet.extract(trafo_image, (y, x))
        cos_image[:, :, y//4, x//4] = cos_sim.similarity([image_jet],
            [eye_jet])
        disp_image[:, :, y//4, x//4] = disp_sim.similarity([image_jet],
            [eye_jet])
```

Listing 8.2: The approach with `Jet`

phases. Like any neural network layer, we directly apply its `forward` method for parallel computation of similarity maps.

It is notable that creating the `reference_tensor` requires a bit of maneuvering. We expand it and ensure its shape matches `jets_tensor`, as `Similarity`'s `forward` method only accepts tensors with the same shape.

To ensure both approaches yield similar results, our tests pass the `torch.allclose` assertion with very low `rtol` and `atol` thresholds.

```
assert torch.allclose(cos_image, cos_image_parallel, rtol=1e-4, atol=1e-4)
assert torch.allclose(disp_image, disp_image_parallel, rtol=1e-4, atol=1e-4)
```

It is evident that whether calculating similarity scores using lists of `Jet` objects or tensors of jets, the results are entirely equivalent. However, it is notable that the method relying solely on tensors exhibits significant improvements in computational performance. We quantified these performance enhancements and provided a detailed discussion in Chapter 10.

```python
gabor_filter = GaborFilterFrequency(in_channels=1, resolution=image.shape[-2:])
trafo_image = gabor_filter(image)

# Employing the GridExtract module to extract every fourth pixel
grid = GridExtract(stride=(4,4))
jets_tensor = grid(trafo_image)

# Create reference tensor
reference_tensor = trafo_image[..., pos[0], pos[1], None,
    None].expand(jets_tensor.shape)

# Transform complex values to absolute values and phases
act = GaborFilterAct(out_type='abs_phase', stack_abs_phase=True,
    normalize=True)
jets_tensor_acted = act(jets_tensor)
reference_tensor_acted = act(reference_tensor)

# Compute similarity map
cos_image_parallel = cos_sim(jets_tensor_acted, reference_tensor_acted)
disp_image_parallel = disp_sim(jets_tensor_acted, reference_tensor_acted)
```

Listing 8.3: The approach with tensors

Chapter 9

Full Network Example

9.1 Pipeline

In previous discussions, we have outlined our new `pytorch_gabor` package, which fundamentally offers two approaches to implement the Gabor wavelet processing pipeline.

The first approach revolves around a pure object-oriented approach. Specifically, the class `pytorch_gabor.Jet` facilitates the extraction, activation, normalization, and encapsulation of a Gabor jet. Meanwhile, the `pytorch_gabor.Graph` class extends the functionalities of the `pytorch_gabor.Jet` class, allowing encapsulation of various positions and extraction of a list of Gabor jets. The `pytorch_gabor.Similarity` class can further process these lists of Gabor jet objects as input. Figure 9.1 illustrates this object-oriented pipeline approach. This approach offers some advantages — leveraging well-encapsulated, object-oriented `Jet` and `Graph` classes that host useful member methods, enhancing readability and interpretability. However, it faces clear drawbacks, notably in integration within modern PyTorch neural network architectures. Therefore, this method poses challenges especially when necessitating parallel Gabor wavelet processing (including batch processing and parallel computation of similarities between two maps of jets with various positions). Another consequence is the difficulty in placing the entire computation process on a GPU.

The second approach relies entirely on `torch.nn.Module`. In this context, the representation of these Gabor jets can be simplified to just a `Tensor`. Every step, from transforming images to activating and normalizing Gabor jets, followed by batch extraction and their utilization as input for `pytorch_gabor.Similarity`, can all be represented using tensors. This approach aligns with the standard adopted by modern PyTorch neural network architectures. Figure 9.2 illustrates how this pipeline is implemented in our package as a PyTorch network. In this pipeline, all the relevant classes are subclasses of `torch.nn.Module`. Additionally, attributes like `weight` are encapsulated using `torch.nn.Parameter`, allowing the entire process to be readily placed on a GPU for accelerated computation in parallel.

9.2 Use Case

To illustrate the aforementioned pipeline in code, we have implemented a use case named `full_network_example.ipynb`. This use case demonstrates a concise network that integrates `GaborFilterFrequency` (can be replaced with `GaborFilterSpatial`), `GaborFilterAct`, `GridExtract` and `Similarity` as a PyTorch Module, making it adaptable for seamless integration into any other neural network.

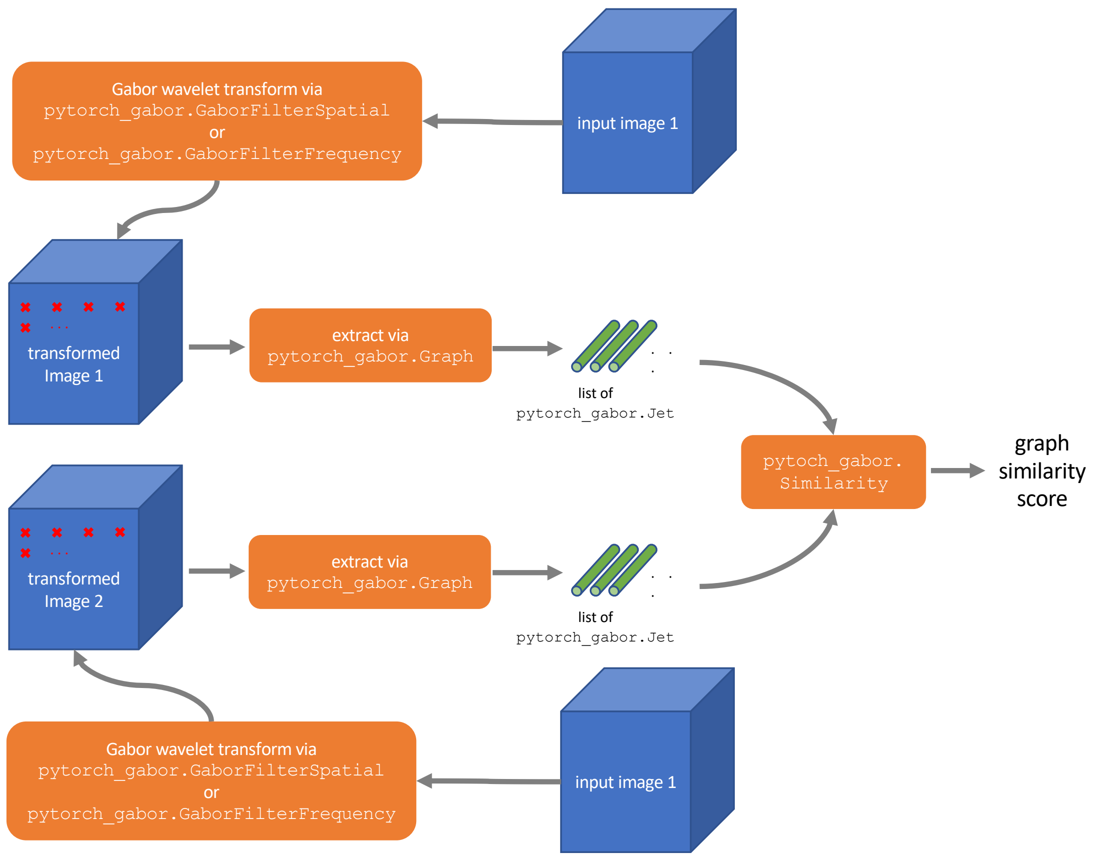

Gabor wavelet transform via
pytorch_gabor.GaborFilterSpatial
or
pytorch_gabor.GaborFilterFrequency
input image 1
transformed Image 1
extract via pytorch_gabor.Graph
list of pytorch_gabor.Jet
pytoch_gabor. Similarity
graph similarity score
transformed Image 2
extract via pytorch_gabor.Graph
list of pytorch_gabor.Jet
Gabor wavelet transform via
pytorch_gabor.GaborFilterSpatial
or
pytorch_gabor.GaborFilterFrequency
input image 1

Figure 9.1: AN OBJECT-ORIENTED PIPELINE OF GABOR WAVELET PROCESSING. *This pipeline relies on `pytorch_gabor.Jet` and `pytorch_gabor.Graph`. Recall that `pytorch_gabor.Jet` internally implements activation (i.e., conversion from complex values to Euler representation).*

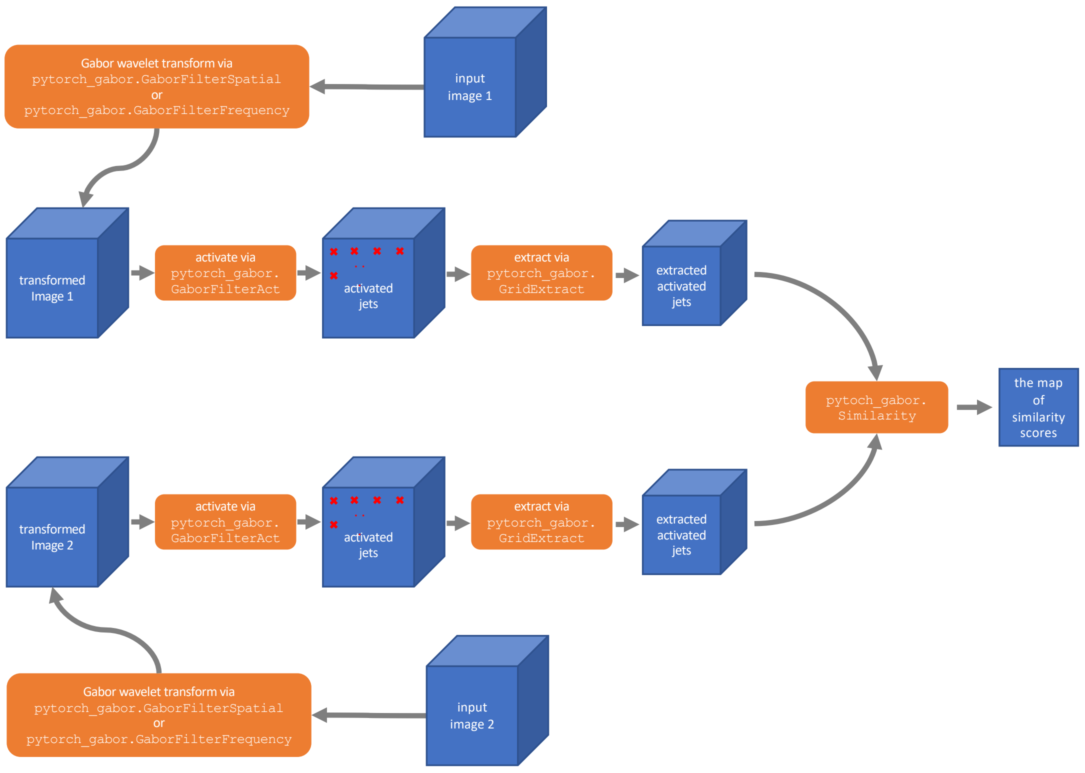

Gabor wavelet transform via
pytorch_gabor.GaborFilterSpatial
or
pytorch_gabor.GaborFilterFrequency
input image 1
transformed Image 1
activate via pytorch_gabor. GaborFilterAct
activated jets
extract via pytorch_gabor. GridExtract
extracted activated jets
pytoch_gabor. Similarity
the map of similarity scores
transformed Image 2
activate via pytorch_gabor. GaborFilterAct
activated jets
extract via pytorch_gabor. GridExtract
extracted activated jets
Gabor wavelet transform via
pytorch_gabor.GaborFilterSpatial
or
pytorch_gabor.GaborFilterFrequency
input image 2

Figure 9.2: AN TENSOR-ORIENTED PIPELINE OF GABOR WAVELET PROCESSING. *This pipeline relies solely on `pytorch.nn.Module`. The blue boxes and squares represent the data flow in `Tensor` format.*

```python
class NetworkFreq(torch.nn.Module):
   def __init__(self, resolution, similarity_type, stride=(4,4)):
      super().__init__()
      self.gwt = GaborFilterFrequency(in_channels=1, resolution=resolution)
      self.grid = GridExtract(stride=stride)
      self.act = GaborFilterAct(out_type='abs_phase', stack_abs_phase=True,
          normalize=True)
      self.sim = Similarity(similarity_type=similarity_type,
          transform=self.gwt)
   def forward(self, img1, img2):
      img1 = self.act(self.grid(self.gwt(img1)))
      img2 = self.act(self.grid(self.gwt(img2)))
      return self.sim(img1, img2)
```

Listing 9.1: The customized network utilizing `GaborFilterFrequency`

The customized `NetworkFreq` (cf. Listing 9.1) starts by utilizing `GaborFilterFrequency` to transform two input images. Given the stability of the texture descriptor against minor shifts, the network employs `GridExtract` to selectively extract every 4th pixel.

Next, the network activates the complex-valued transformed images into tensors comprising absolute and phase values, representing maps of Gabor jets distributed along the y-axis and x-axis.

Finally, the similarity layer computes similarity scores for the Gabor jets derived from both input images. These scores are localized within a similarity map, represented as a `Tensor`. Each entry in this similarity map corresponds to the similarity score of two jets positioned at the same coordinates.

Moreover, we have the option to utilize `GaborFilterSpatial` in another customized network `NetworkSpat` (cf. Listing 9.2), producing outcomes very similar to those using `GaborFilterFrequency`.

When constructing a network with `GaborFilterSpatial`, we can follow a similar approach: first, transform the complete image and then use `GridExtract` to extract every 4th pixel. However, for efficiency in the spatial domain, we do not need to transform every pixel of the original images. Given `GaborFilterSpatial` inherits from `torch.nn.Conv2d`, it inherently supports a stride parameter. This modification is worthwhile for networks based on `GaborFilterSpatial` because it can make the computation 16 times faster in this case.

Finally, we can run these two networks to obtain corresponding similarity maps with six different similarity types (Figure 9.3). As anticipated, whether using `NetworkFreq` or `NetworkSpat`, we achieved nearly identical similarity maps.

```python
class NetworkSpat(torch.nn.Module):
    def __init__(self, similarity_type, stride=(4,4)):
        super().__init__()
        # The kernel_size and padding will be computed automatically when
            initializing GaborFilterSpatial if they are set to None.
        self.gwt = GaborFilterSpatial(in_channels=1, stride=stride,
            kernel_size=None, padding=None,)
        self.act = GaborFilterAct(out_type='abs_phase', stack_abs_phase=True,
            normalize=True)
        self.sim = Similarity(similarity_type=similarity_type,
            transform=self.gwt)
    def forward(self, img1, img2):
        img1 = self.act(self.gwt(img1)) # no need for grid extraction
        img2 = self.act(self.gwt(img2))
        return self.sim(img1, img2)
```

Listing 9.2: The customized network utilizing `GaborFilterSpatial`

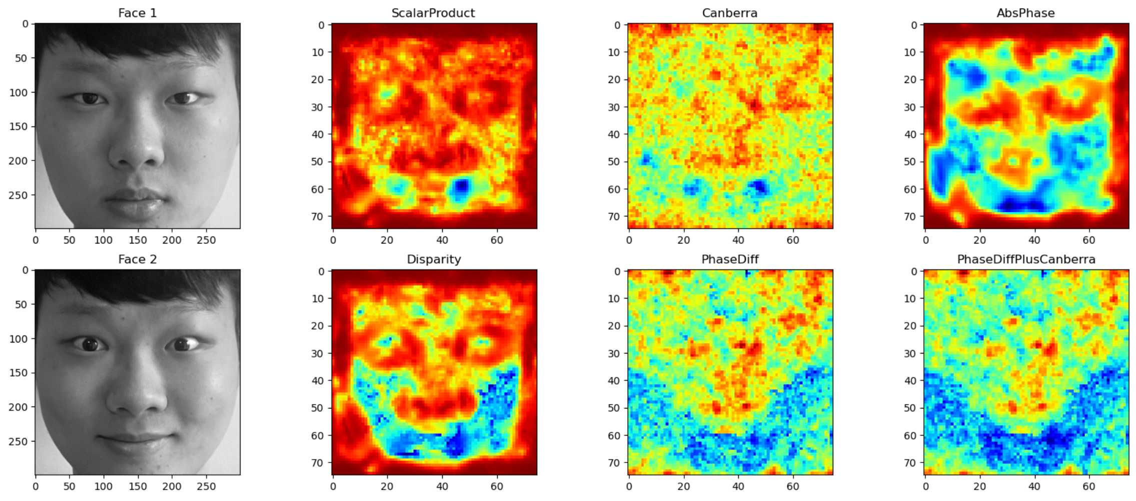


Figure 9.3: SIMILARITY MAPS. *These are the results produced by* `NetworkFreq` *and* `NetworkSpat`*. The 6 similarity maps corresponds to 6 similarity functions. Face 1 and Face 2 are sample input images in this case.*

Chapter 10

Evaluation

Given that one of the main reasons for implementing the `pytorch_gabor` package was parallelization, it is essential to measure how much of a speed boost parallelization offers when performing Gabor wavelet processing using our new `pytorch_gabor` package. Hence, this section primarily focuses on quantifying and demonstrating the performance enhancement and discussing the final outcomes.

10.1 Methodology

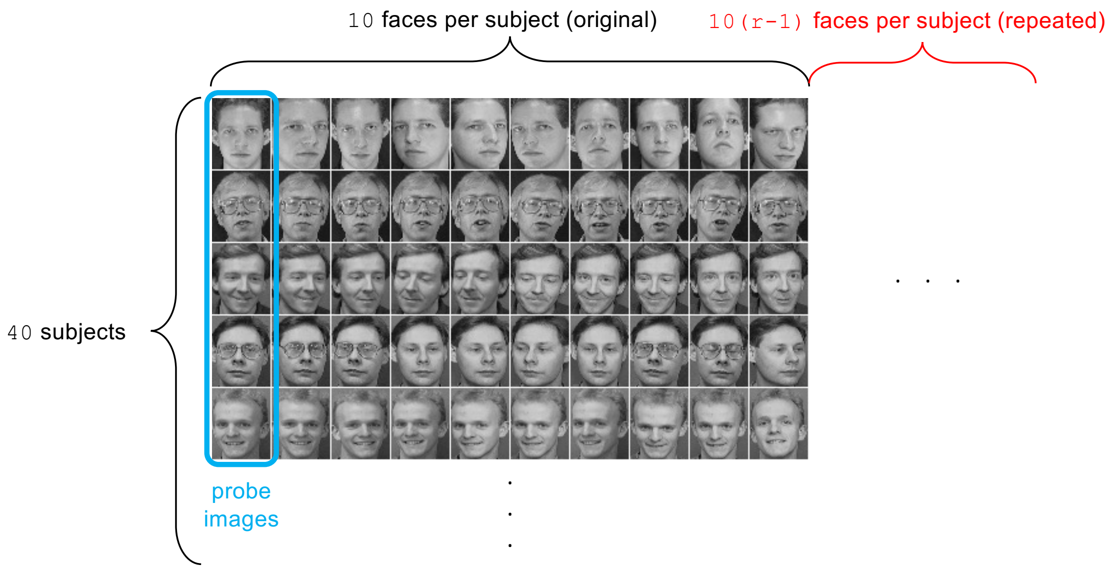
10.1.1 Dataset

We utilized the AT&T dataset[1] for this experiment, which contains 10 different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).

This dataset was previously employed as a toy dataset in the legacy library, `bob.example.faceverify`, to assess the predictive accuracy of Gabor wavelet processing using `bob.ip.gabor` and generate corresponding ROC curves. However, in this experiment, measuring the predictive accuracy of `pytorch_gabor` when performing the Gabor wavelet processing is not our objective, because previous discussions have adequately demonstrated that both the new and old packages yield identical results. Instead, our primary goal in this experiment is to measure the computational speed of both the new and old packages when dealing with the same task but with varying amounts of data.

The original AT&T database contains only 400 facial images. To measure computational speed with different data amounts, we introduced a repeat factor r to control the data volume. As illustrated in Figure 10.1, by assigning different positive integer values to r, the augmented AT&T dataset expands to contain $400 \times r$ facial images for experimentation. These images will serve as the references for the functions of GWT and similarity. To compute similarity scores, we also need to define some probe images. As the simplest approach, we just select the first face from each subject as a probe image. Hence, there are a total of 40 probe images.

[1] `https://www.kaggle.com/datasets/kasikrit/att-database-of-faces`



10 faces per subject (original)
10(r-1) faces per subject (repeated)
40 subjects
probe images

Figure 10.1: ADJUST VOLUME OF DATA. *We adjust the volume of data by controlling the repetition of face images within all subjects, generating the augmented AT&T dataset.*

10.1.2 Pipelines

For this experiment, we have to test the Gabor wavelet processing pipelines on different hardware setups to obtain our experimental results. We have prepared two environments: one on a CPU environment and another on a GPU environment. For the CPU, our setup includes a macOS system with an Apple M1 chip, 8 total cores, and 16GB system RAM. As for the GPU (CUDA) environment, we are utilizing a cloud-based setup on Google Colab, equipped with 12.7GB system RAM and 15GB GPU RAM.

The legacy `bob.ip.gabor` package does not support parallel computation on GPUs, restricting its operation solely to CPU environments. Conversely, our new package can run on both CPU and GPU environments. Hence, we run our experiments using `pytorch_gabor` on both a local CPU setup and the CUDA environment in Google Colab.

For the pipeline design of this experiment, the legacy evaluation code in `bob.example.face-verify` remains a valuable reference. However, there are some interface differences between the new and old packages, and generating ROC curves is not our objective. Hence, we have made necessary modifications to the legacy evaluation code and created a standard pipeline as depicted in Figure 10.2.

In more detail, concerning the process of computing features, the old `bob.ip.gabor` can only transform images one by one and relies on the `bob.ip.gabor.Graph` class to extract and encapsulate Gabor jets (cf. Listing 10.1). However, `pytorch_gabor` can batch-transform images, and relies on the `stride` parameter to extract Gabor jets within `Tensor` (cf. Listing 10.2 and Listing 10.3).

As for the process of computing similarities, the complexity of the old `bob.ip.gabor` is higher. It requires four nested `for` loops to accomplish this task since its similarity function can only handle two single jets at a time (cf. Listing 10.4).

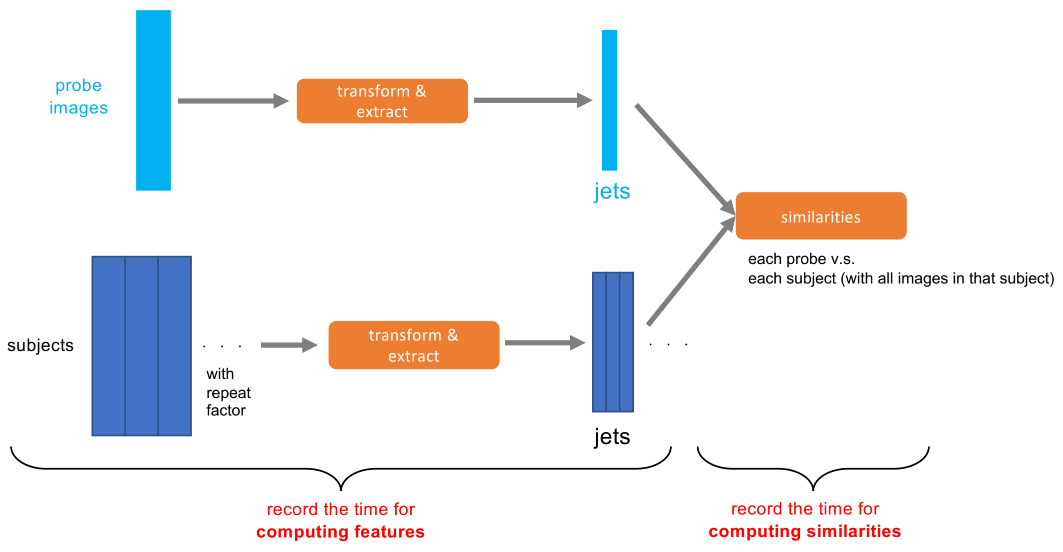

probe images
transform & extract
jets
similarities
each probe v.s. each subject (with all images in that subject)
subjects
with repeat factor
transform & extract
jets
record the time for computing features
record the time for computing similarities

Figure 10.2: A STANDARD PIPELINE TRACKING THE COMPUTING TIME. *This pipeline is applied to track the time taken by different packages for computing features and similarities with increasing data amount. As the outcome of this pipeline, we obtain average similarity scores between each probe image and each subject.*

However, our new `pytorch_gabor` allows us to harness its capability to process batch and graph nodes in parallel, eliminating the need for two nested `for` loops (cf. Listing 10.5). Furthermore, we can also run this process on the GPU, which significantly enhances computational efficiency .

In summary, Table 10.1 comprises five specific experiments, demonstrating how we implemented the predefined standard pipeline using different packages, classes, and hardware. Furthermore, to enhance the credibility of our experimental results, we repeated these experiments 20 times and recorded the average time taken.

10.2 Results

Consistent similarity score results are obtained regardless of whether using the old `bob.ip.gabor` or the new `pytorch_gabor`, as long as the pipeline described earlier is followed. These similarity scores are presented in Figure 10.3 as a heatmap. Since this is not the primary focus of this experiment, there is no need to discuss these similarity scores here.

By progressively increasing the dataset size, we obtained and recorded the time taken for computing features and similarities with 5 different configurations. Figure 10.4 are two line graphs that illustrate the trends in runtime.

In terms of computing features, running Gabor wavelet transform in the spatial domain on a CPU is notably slower compared to any other methods. This is mainly because the default settings in `pytorch_gabor.GaborFilterSpatial` generates a large kernel size (161 $\times$ 161), and the convolution operations are much more computationally intensive than simple pixel-wise

Id	**Relevant package**	**Relevant classes for computing features**	**Relevant classes for computing similarities**	**Hardware**
1	bob.ip.gabor	Transform Graph	Similarity	CPU
2	pytorch_gabor	GaborFilterFrequency GridExtract	Similarity	CPU
3	pytorch_gabor	GaborFilterSpatial (with striding)	Similarity	CPU
4	pytorch_gabor	GaborFilterFrequency GridExtract	Similarity	GPU
5	pytorch_gabor	GaborFilterSpatial (with striding)	Similarity	GPU

Table 10.1: EXPERIMENTAL CONFIGURATIONS.

multiplications. When it comes to `pytorch_gabor.GaborFilterFrequency` on a CPU, it does not perform significantly better than `bob.ip.gabor.Transform`.

Regarding computing similarities, `bob.ip.gabor`, which relies on C++, shows faster runtime compared to `pytorch_gabor.Similarity` on a CPU, if there are smaller data volumes (below 2400). However, as the dataset size exceeds 2400 instances, the advantage of batch processing becomes evident, reducing the runtime of `pytorch_gabor.Similarity` below that of `bob.ip.gabor.Similarity`.

The standout discovery occurs when `pytorch_gabor` operates on a GPU. Whether computing features or similarities, the runtime significantly decreases compared to other setups. Additionally, as the dataset size grows, there is no notable increase in processing time.

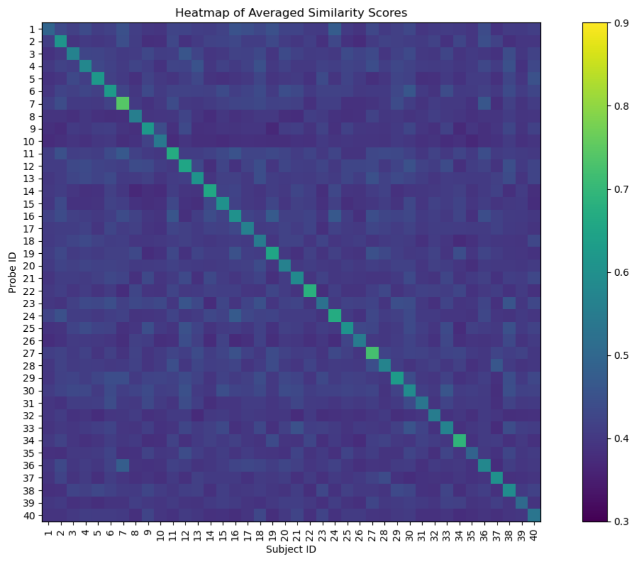


Figure 10.3: RESULTS OF SIMILARITY SCORES. *The scores are obtained through both the legacy* `bob.ip.gabor` *and the new* `pytorch_gabor` *methods following the outlined pipeline and are visualized in the heatmap. Please note that, for the purpose of visualization, this heatmap displays only the average similarity scores between each probe image and all images in each subject set, rather than the similarity scores between each probe image and each subject image.*

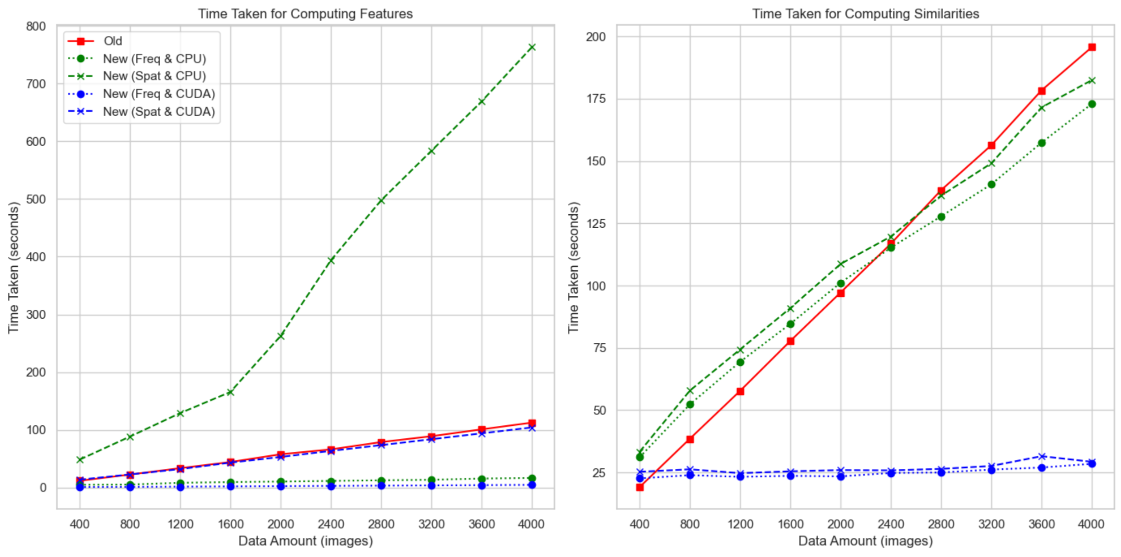


Figure 10.4: EXPERIMENTAL RESULTS. *Time taken for computing features(left) and computing similarities(right) with increasing data amount using different packages, classes, and hardware configurations, where "Old" means* `bob.ip.gabor` *and "New" means* `pytorch_gabor`*.*

```python
image_resolution = DataSetAtnt.image_resolution
gabor_wavelet_transform = bob.ip.gabor.Transform()
# pre-allocate Gabor wavelet transform image in the desired size
trafo_image = np.ndarray((gabor_wavelet_transform.number_of_wavelets, 112,
    92), np.complex128)

def extract_feature(image, extractor: bob.ip.gabor.Graph) ->
    List[bob.ip.gabor.Jet]:
   # perform Gabor wavelet transform on the image
   gabor_wavelet_transform.transform(image, trafo_image)
   gabor_graph = extractor.extract(trafo_image)
   return gabor_graph

def compute_features(db) -> Tuple[Dict, Dict]:
   graph_extractor = bob.ip.gabor.Graph(first=(0, 0), last
       (image_resolution[0] - 1, image_resolution[1] - 1), step=(4, 4))

   subject_files = db.load_images()
   subjects = {}
   for subject_id in subject_files.keys():
      # load enroll images for the current subject ID
      enroll_images = subject_files[subject_id]
      # extract features for all enroll images and store all of them
      subjects[subject_id] = [extract_feature(enroll_image, graph_extractor)
          for enroll_image in enroll_images]

   probe_files = db.load_probes()
   probes = {}
   for probe_id in probe_files.keys():
      probes[probe_id] = extract_feature(probe_files[probe_id],
          graph_extractor)

   return subjects, probes
```

Listing 10.1: Computing features using the old `bob.ip.gabor`

```python
image_resolution = DataSetAtnt.image_resolution
# define Gabor wavelet transform class globally since it is reused for all
    images
gabor_wavelet_transform = pytorch_gabor.GaborFilterFrequency(in_channels=1,
    resolution=image_resolution).to(torch.float64)

class ExtractFeature(torch.nn.Module):
   def __init__(self, stride=(4, 4)):
      super().__init__()
      self.gwt = gabor_wavelet_transform
      self.grid = pytorch_gabor.GridExtract(stride=stride)
      self.act = pytorch_gabor.GaborFilterAct(out_type='abs_phase',
          stack_abs_phase=True, normalize=True)
   def forward(self, img):
      gabor_graph = self.act(self.grid(self.gwt(img)))
      return gabor_graph

def compute_features(db, device='cpu') -> Tuple[Dict, Dict]:
   extract_feature = ExtractFeature().to(device)

   subject_files = db.load_images()
   subjects = {}
   for subject_id in subject_files.keys():
      # load enroll images for the current subject ID
      enroll_images = torch.stack(subject_files[subject_id],
          dim=0).unsqueeze(dim=1)
      # extract features for all enroll images and store all of them
      subjects[subject_id] = extract_feature(enroll_images)

   batch_size = subjects[1].shape[0] # record the batch size

   probe_files = db.load_probes()
   probes = {}
   for probe_id in probe_files.keys():
      # parallelization
      probe_feature = extract_feature(probe_files[probe_id][None, None, ...])
      probes[probe_id] = probe_feature.expand(batch_size, -1, -1, -1, -1)

   return subjects, probes
```

Listing 10.2: Computing features using the `pytoch_gabor.GaborFilterFrequency`

```
gabor_wavelet_transform = pytorch_gabor.GaborFilterSpatial(in_channels=1,
    stride=(4, 4), dtype=torch.complex128)

class ExtractFeature(torch.nn.Module):
   def __init__(self):
      super().__init__()
      self.gwt = gabor_wavelet_transform
      self.act = pytorch_gabor.GaborFilterAct(out_type='abs_phase',
          stack_abs_phase=True, normalize=True)
   def forward(self, img):
      return self.act(self.gwt(img))

def compute_features(db, device='cpu') -> Tuple[Dict, Dict]:
   # This function is exactly the same as the function compute_features() in
       the case of using GaborFilterFrequency
```

Listing 10.3: Computing features using the `pytoch_gabor.GaborFilterSpatial`

```
def compute_similarities(subjects, probes):
   scores = defaultdict(lambda: defaultdict(list))
   for subject_id, subject_features in subjects.items():
      for subject_feature in subject_features:
         for probe_id, probe_feature in probes.items():
            sims = []
            for jet_index in range(len(probe_feature)):
               sims.append(SIMILARITY_FUNCTION(subject_feature[jet_index],
                                                probe_feature[jet_index]))
            sim = np.mean(sims)
            scores[probe_id][subject_id].append(sim)
   return scores
```

Listing 10.4: Computing similarities using the old `bob.ip.gabor`

```
def compute_similarities(subjects, probes, device='cpu'):
   sim_layer = SIMILARITY_FUNCTION.to(device)
   scores = defaultdict(lambda: defaultdict(torch.Tensor))
   for subject_id, subject_features in subjects.items():
      for probe_id, batched_probe_feature in probes.items():
         # similarity function in pytorch_gabor can accept batched jets and
         # compute graph similarity with all graph nodes in parallel
         scores[probe_id][subject_id] = sim_layer(subject_features,
                                       batched_probe_feature).mean(dim=(-2, -1))
   return scores
```

Listing 10.5: Computing similarities using the new `pytorch_gabor`

Chapter 11

Conclusion and Future Work

11.1 Conclusion

In this master project, we undertook the re-implementation of the `bob.ip.gabor` package in pure Python and PyTorch, achieving comparable results against specified criteria. Our modifications, detailed in Table 2.1, ensured parallel processing and smooth integration into modern deep learning methods.

Handling "convolution" in `GaborFilterSpatial` posed unexpected challenges, given that the standard PyTorch convolution layer actually performs "cross-correlation" instead of true convolution (5.2.2). We addressed this issue through theoretical analysis and implemented a straightforward solution in the code. By incorporating a mechanism for automatic calculation of kernel size and padding, we enabled `GaborFilterSpatial` to generate results identical to those of `GaborFilterFrequency`. This agreement holds even in various edge cases, such as when handling multi-channel images. We have further endeavored to achieve parallelization for both `GaborFilterSpatial` and `GaborFilterFrequency` by decoupling the code as much as possible.

Implementing parallel processing in `Jet`, `Graph`, and `Jetstatistics` proved straightforward. However, challenges arose in parallelizing the `Similarity` component due to the variable input shape. In order to overcome this, we leveraged PyTorch functionalities like `torch.einsum` for effective parallelization.

Our implementation not only reproduces old use cases of `bob.ip.gabor` but also introduces a full network example, showcasing a Gabor wavelet processing pipeline entirely reliant on class `torch.nn.Module`. Experimental evaluations demonstrate a significant performance boost in features and similarities computation with `pytorch_gabor` on CUDA, especially evident with large datasets compared to running on CPU or using `bob.ip.gabor`.

In conclusion, our work not only contributes to the field by providing a more versatile and efficient Gabor wavelet processing package but also opens avenues for further exploration in parallelization and deep learning integration. The demonstrated speed improvements underscore the practical applicability of our `pytorch_gabor` package in real-world scenarios.

11.2 Future work

Possible future work for this package includes:

Learnable Parameters in Gabor Filters: The current implementation of the `pytorch_gabor` package relies on manually set parameters for Gabor filters. To enhance adaptability and performance, a potential avenue for future work involves introducing learnable parameters. Nevertheless, allowing each wavelet to learn and update the specific value of each entry might lead our `GaborFilterSpatial` and `GaborFilterFrequency` to behave like regular neural networks. A more meaningful approach is to enable the parameter set Γ, to undergo learning and updating. However, this introduces a challenge due to the involvement of non-differentiable elements such as `num_of_directions` and `num_of_scales`, which are integer-valued. That requires the exploration of alternative optimization strategies or innovative parameterization schemes compatible with gradient-based optimization methods.

Integration into the Idiap Ecosystem: The successful integration of the `pytorch_gabor` package into the broader Idiap ecosystem is a key aspect of future development. This integration goes beyond the package's functionality and involves smooth inclusion into existing workflows and systems. Specifically, efforts should be directed toward integrating the package into the Idiap continuous integration (CI) system. This ensures ongoing compatibility testing, including unit tests, integration tests, and performance tests. Additionally, collaborative initiatives with other projects and researchers within Idiap can foster knowledge exchange, improvements, and a more interconnected research environment.

List of Figures

List of Tables

List of Listings

Bibliography

Anjos, A., El-Shafey, L., Wallace, R., Günther, M., McCool, C., and Marcel, S. (2012). Bob: a free signal processing and machine learning toolbox for researchers. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 1449–1452.

Anjos, A., Günther, M., de Freitas Pereira, T., Korshunov, P., Mohammadi, A., and Marcel, S. (2017). Continuously reproducing toolchains in pattern recognition and machine learning experiments. In *Thirty-fourth International Conference on Machine Learning*.

Buhmann and Lange (1989). Distortion invariant object recognition by matching hierarchically labeled graphs. In *International 1989 Joint Conference on Neural Networks*, pages 155–159. IEEE.

Daugman, J. G. (1985). Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *JOSA A*, 2(7):1160–1169.

Günther, M. (2012). *Statistical Gabor graph based techniques for the detection, recognition, classification, and visualization of human faces*. Shaker.

Günther, M., Haufe, D., and Würtz, R. P. (2012). Face recognition with disparity corrected Gabor phase differences. In *Artificial Neural Networks and Machine Learning–ICANN 2012: 22nd International Conference on Artificial Neural Networks, Lausanne, Switzerland, September 11-14, 2012, Proceedings, Part I 22*, pages 411–418. Springer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Lades, M., Vorbruggen, J. C., Buhmann, J., Lange, J., Von Der Malsburg, C., Wurtz, R. P., and Konen, W. (1993). Distortion invariant object recognition in the dynamic link architecture. *IEEE Transactions on computers*, 42(3):300–311.

Luan, S., Chen, C., Zhang, B., Han, J., and Liu, J. (2018). Gabor convolutional networks. *IEEE Transactions on Image Processing*, 27(9):4357–4366.

Schneider, H. J., Kosilek, R. P., Günther, M., Roemmler, J., Stalla, G. K., Sievers, C., Reincke, M., Schopohl, J., and Würtz, R. P. (2011). A novel approach to the detection of acromegaly: accuracy of diagnosis by automatic face classification. *The Journal of Clinical Endocrinology & Metabolism*, 96(7):2074–2080.

Wiskott, L., Fellous, J.-M., Kruger, N., and Malsburg, C. (1996). Face recognition by elastic bunch graph matching. *TR96-08, Institut für Neuroinformatik, Ruhr-Universität Bochum*.

Zhang, W., Shan, S., Gao, W., Chen, X., and Zhang, H. (2005). Local Gabor binary pattern histogram sequence: A novel non-statistical model for face representation and recognition. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 1, pages 786–791. IEEE.