When Multiplication Surpasses Convolution: Computational Efficiency and Optimizations

Lapo Chiostrini, Lorenzo Cappetti

07/03/2025

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduction

CNN

- **Convolutional Neural Networks (CNNs)** are neural networks designed to process images and structured data.
- They use convolutional filters to automatically extract relevant features.
- **Disadvantages:**
  - High computational cost
  - Not easily interpretable
  - Large number of parameters.

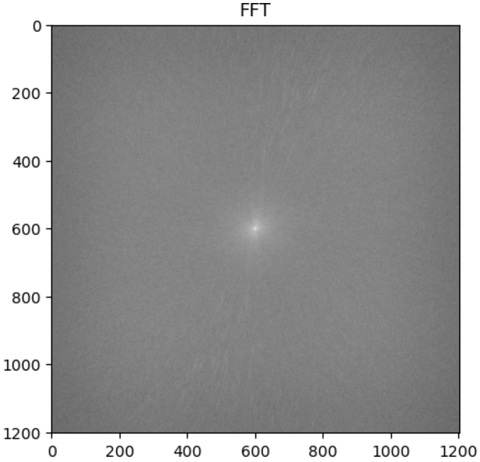
Goal

Why Use the Frequency Domain?

- According to the **Convolution Theorem**, we can perform a convolution in the spatial domain by using element-wise multiplication in the frequency domain.
- For this reason, we will use the **Fourier Transform** on the image and convolutional filters.
- **Advantages:**
  - Potential reduction in computational cost
  - Possibility of compression and noise reduction, useful for image processing applications.

Why Transforms?

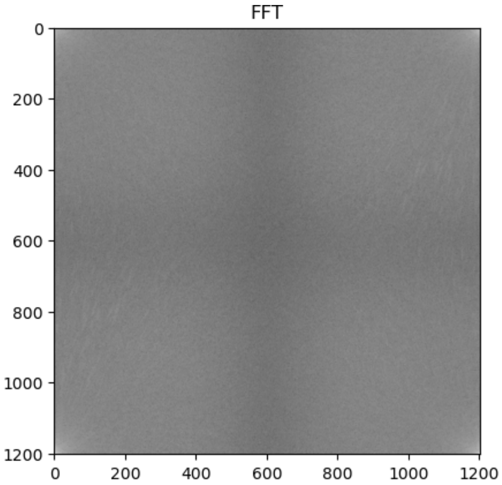
(a) Original Image



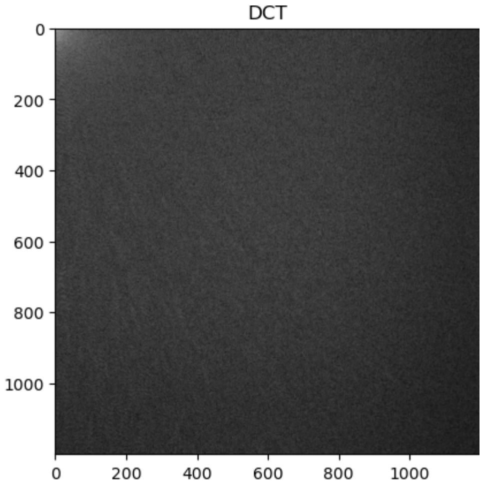
(b) Transformed Image

Why FFT?

Energy Compaction vs Accuracy



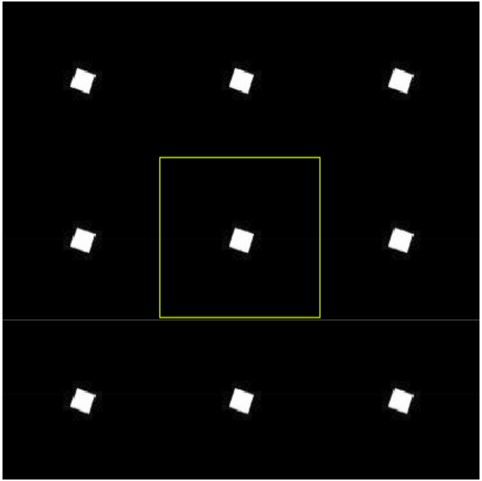
(a) Image Transformed with FFT



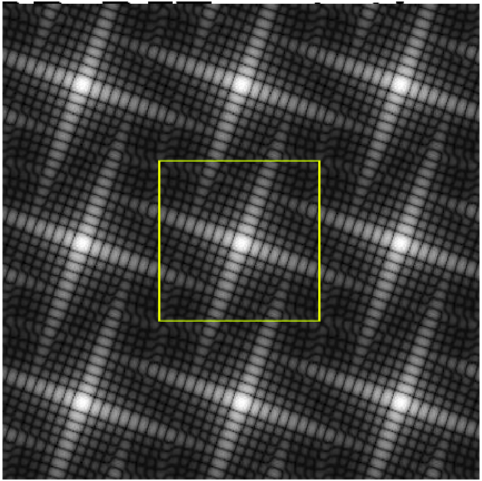
(b) Image Transformed with DCT

FFT Problems

Need for Padding and the Centering Dilemma



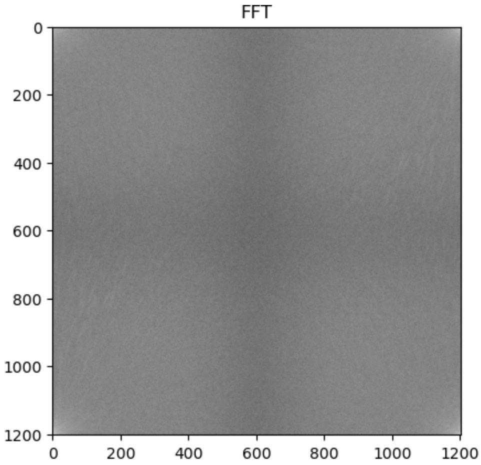
(a) Original Image



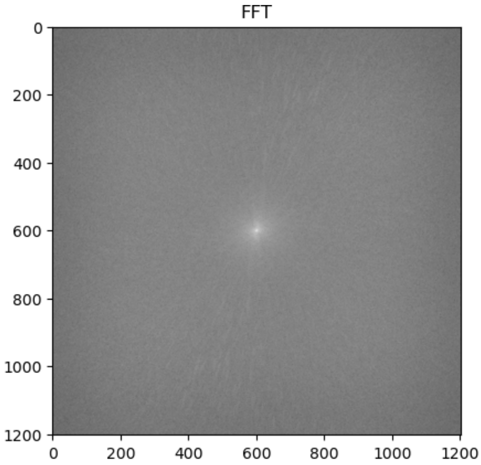
(b) Transformed Image

FFT Problems

Need for Padding and the Centering Dilemma



(a) FFT without Shift



(b) FFT with Shift

Convolution Theorem

Feasibility and Theoretical Overview

Convolution in the spatial domain can be transformed into a **multiplication** in the frequency domain:

$$(f * g)(x) = \int f(t)g(x - t)dt$$

Applying the Fourier Transform, we obtain:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

In terms of the inverse transform:

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$$

This allows replacing convolution with element-wise multiplication, reducing computational cost.

Computational Complexity Analysis

The cost to perform a matrix multiplication is:

$$O(N^3)$$

- Where N is the dimension of the matrix, assuming it is square.

The cost of convolutions is:

$$(A - K + 1)^2 \times O(K^3)$$

- A is the dimension of the image (assuming it is square, thus $A \times A$).
- K is the dimension of the convolution kernel (also square, $K \times K$).

From a theoretical standpoint, convolution should be **worse** than multiplication, especially when using large kernels.

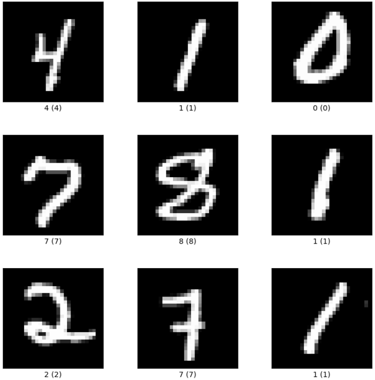
What Do We Expect?

Based on what was mentioned earlier, we expect that:

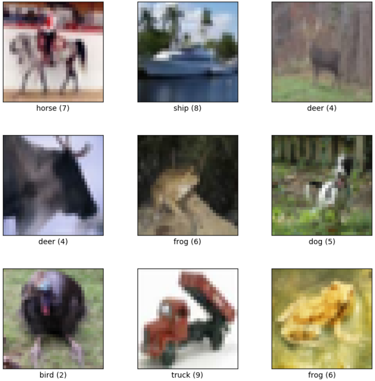
- our model will be **faster**, especially as the filter size increases
- both models will yield **similar accuracy**

Datasets Used

MNIST vs CIFAR-10



(a) MNIST Example



(b) CIFAR-10 Example

How Did We Create the New Datasets?

UNIVERSITÀ DEGLI STUDI FIRENZE

```python
import torch
from torchvision import datasets, transforms
import torchvision.transforms.functional as TF
import torch.nn.functional as F

import os


# Definiamo la trasformazione
transform = transforms.Compose([
    transforms.ToTensor(),
])

# Carichiamo il dataset con la nuova trasformazione
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

# Convertiamo le immagini in tensori
train_images = torch.stack([img for img, _ in trainset])
test_images = torch.stack([img for img, _ in testset])

# Salviamo il dataset trasformato
torch.save({
    'train_images': train_images,
    'train_labels': torch.tensor(trainset.targets),
    'test_images': test_images,
    'test_labels': torch.tensor(testset.targets),
}, './dataset/cifar10_RGB.pt')

print("Dataset in formato RGB salvato su disco.")
```

Figure: CIFAR-10 RGB Base

```python
import torch
from torchvision import datasets, transforms
import torch.nn.functional as F
import os

class FFTTransform:
    def __call__(self, img):
        img_tensor = transforms.functional.to_tensor(img)
        padded_img = F.pad(img_tensor, (0, 6, 0, 6))
        fft_result = torch.fft.fft2(padded_img)
        fft_magnitude = torch.abs(fft_result)
        fft_magnitude = fft_magnitude / fft_magnitude.max()
        return fft_magnitude

transform = transforms.Compose([
    FFTTransform(),
])

trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

train_images = torch.stack([img for img, _ in trainset])
test_images = torch.stack([img for img, _ in testset])

torch.save({
    'train_images': train_images,
    'train_labels': torch.tensor(trainset.targets),
    'test_images': test_images,
    'test_labels': torch.tensor(testset.targets),
}, './dataset/transformed_cifar10_RGB.pt')

print("Dataset saved to disk.")
```

Figure: CIFAR-10 RGB Transformed

Our Models vs Classic CNN

```python
class ClassicCNN(nn.Module):
    def __init__(self):
        super(ClassicCNN, self).__init__()
        print("Modello= Baseline Minst")
        print(f"Kernel: {kernel_size}, Conv1: {conv1_channels}, Conv2: {conv2_channels}, FC1: {fc1_size}")
        self.conv1 = nn.Conv2d(1, conv1_channels, kernel_size, padding=2)
        self.conv2 = nn.Conv2d(conv1_channels, conv2_channels, kernel_size, padding=2)

        with torch.no_grad():
            dummy_input = torch.zeros(1, 1, 28, 28)
            dummy_output = self.conv2(self.conv1(dummy_input))
            self.flatten_dim = dummy_output.numel()

        self.fc1 = nn.Linear(self.flatten_dim, fc1_size)
        self.fc2 = nn.Linear(fc1_size, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        return x

return ClassicCNN()
```

Figure: Classic CNN Model for MNIST

```python
class NostroMinst(nn.Module):
    def __init__(self,conv1_channels, conv2_channels, fc1_size, kernel_size):
        super(NostroMinst, self).__init__()
        print("Modello= Nostro Minst")
        print(f"Kernel: {kernel_size}, Conv1: {conv1_channels}, Conv2: {conv2_channels}, FC1: {fc1_size}")

        self.conv1 = FrequencyConv(1, conv1_channels, kernel_size)
        self.conv2 = FrequencyConv(conv1_channels, conv2_channels, kernel_size)
        self.fc1 = None
        self.fc2 = nn.Linear(fc1_size, 10)
        self.flatten_dim=None
        self.fc1_size = fc1_size

    def forward(self, x):
        x = self.conv1(x)
        x = torch.abs(x)
        x = self.conv2(x)
        x = torch.abs(x)
        if self.flatten_dim is None:
            print("Dimensione dopo convoluzione:", x.shape)
            self.flatten_dim = x.shape[1] * x.shape[2] * x.shape[3]  # Calcoliamo la dimensione corretta
            self.fc1 = nn.Linear(self.flatten_dim, self.fc1_size).to(x.device)  # Inizializziamo `fc1` dinamicamente

        x = x.reshape(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

Figure: Our Model for MNIST

Our Models vs Classic CNN

Our modified layer

```python
class FrequencyConv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size):
        super(FrequencyConv, self).__init__()
        self.kernel_size = kernel_size
        self.out_channels = out_channels
        self.in_channels = in_channels
        self.weights = nn.Parameter(torch.randn(1, out_channels, in_channels, kernel_size, kernel_size, dtype=torch.cfloat))

    def forward(self, x):
        batch_size, _, H, W = x.shape
        kernel_padded = torch.zeros((batch_size, self.out_channels, self.in_channels, H, W), dtype=torch.cfloat, device=x.device)
        kernel_padded[:, :, :, :self.kernel_size, :self.kernel_size] = self.weights
        kernel_freq = to_frequency_domain(kernel_padded)

        out_freq = x.unsqueeze(1) * kernel_freq
        return out_freq.sum(dim=2)
```

Figure: Implementation for our layer

Experimental Setup

Kernel Configuration

For the experiments, we varied:

- **the number of kernels**: 8 for the first layer and 16 for the second, or 16 for the first and 32 for the second.
- **the kernel sizes**: 3 and 7.

For each model, we performed 4 trainings with 15 epochs each, repeated 5 times, and then computed the average of the results. In total, we ran 6x4x5 = 120 training processes. To ”help” our model we didn’t transform the datasets at runtime but we created transform datasets to load at runtime.
We ran our experimets on a pc with this specifications:
GPU AMD Radeon(TM) Graphics; scheda video NVIDIA GeForce RTX 4060 Ti

Raw Results

MNIST								
	Baseline				Our			
Kernel	3	3	7	7	3	3	7	7
N. Kernel	8	16	8	16	8	16	8	16
Time (s)	42.70	69.90	38.10	51.20	61.35	212.12	61.73	217.04
Accuracy (%)	98.19	98.41	98.66	98.75	86.73	86.33	85.78	86.49

UNIVERSITÀ DEGLI STUDI FIRENZE

Raw Results

CIFAR-10 RGB								
	Baseline				Our			
Kernel	3	3	7	7	3	3	7	7
N. Kernel	8	16	8	16	8	16	8	16
Time (s)	27	57.93	24.38	44.88	158.55	580.64	159.44	583.98
Accuracy (%)	56.41	57.06	55.06	54.50	47.06	48.39	47.73	48.30

CIFAR-10 HSV								
	Baseline				Our			
Kernel	3	3	7	7	3	3	7	7
N. Kernel	8	16	8	16	8	16	8	16
Time (s)	26.65	57.91	24.60	44.89	158.57	578.91	159.25	584.66
Accuracy (%)	50.00	51.86	52.09	52.67	46.96	46.68	47.49	46.52

Results Interpretation

Some Strange results

1. In our model, the computational time **does not change** when the kernel size is altered.
2. The computational time is consistently **higher** in our model.
3. The accuracy is **lower** compared to the baseline.
4. The accuracy does not significantly improve with larger kernels.

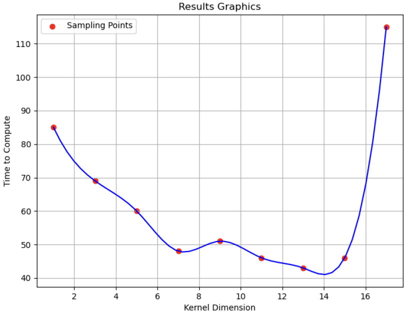
Results Interpretation

Some Reasonable Explanations

1. This is due to the use of padding to match the dimensions.
2. This is probably because of the overhead involved in instantiating all the kernels and transforming them. The process becomes especially heavy as the number of kernels increases. Based on the results, we can assume that the time complexity is quadratic in relation to the number of kernels.
3. The lower accuracy is probably caused by rounding errors introduced when using the FFT.

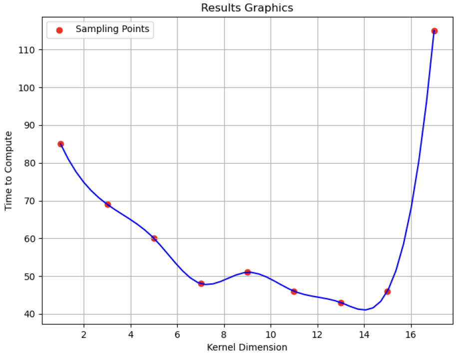
A Parcicular Strange result

We observed that the computational time in the baseline is consistently lower for a kernel size of 7 compared to a kernel size of 3. This was surprising to us, as we expected the second term of the convolution complexity to dominate. To investigate further, we conducted tests with different numbers and obtained the following results:



Considerations

This is likely due to the fact that the second term becomes larger but is repeated fewer times, making it more efficient up to a certain threshold, which in our case was found to be 13. It's also important to consider the time required for weight updates and their instantiation.



Paper Introduction

Introduction and Specs

- The paper we found is **”Learning Convolutional Neural Networks in the Frequency Domain”** by **H. Pan, Y. Chen, X. Niu, W. Zhou, and D. Li**.
- This paper proposes **CEMNet (Complex Element-wise Multiplication Network)**, a neural network that replaces convolution with element-wise multiplication in the frequency domain, using the Fourier Transform to reduce computational complexity.
- The paper uses the following training specifications:
  - **Number of epochs**: 800, **Batch size**: 100
  - **Learning rate**:
    - Initial: 0.004
    - Minimum: 0.0000001

Risultati paper

Modello	Forward Ops	Backward Ops	Errore Test
		MNIST	
CNN (LeNet-5)	$\approx 692K$	$\approx 692K$	0.72%
CEMNet	$\approx 368K$	$\approx 481K$	**0.67%**
		CIFAR-10	
Small CNN	$\approx 68.10M$	$\approx 68.10M$	21.07%
Small CEMNet	$\approx 31.33M$	$\approx 50.17M$	22.40%
Large CNN	$\approx 275.45M$	$\approx 275.45M$	**11.30%**
Large CEMNet	$\approx 124.39M$	$\approx 194.28M$	21.63%

Table: Confronto tra CNN e CEMNet su MNIST e CIFAR-10

Conclusions

UNIVERSITÀ DEGLI STUDI FIRENZE

- We have analyzed the use of Fourier transforms to improve the efficiency of convolutions in convolutional neural networks.
- Our method showed a significant theoretical computational saving, but the practical implementation revealed some limitations.
- The results show that:
  - The training time for the FFT-based model is longer compared to standard convolutions, due to preprocessing operations.
  - The accuracy achieved with the frequency-based method is lower than that of the traditional model.
  - The expected computational advantage may be better appreciated on larger datasets and with deeper networks.
- Possible future improvements include optimizations in the FFT implementation and exploring strategies to preserve information during the transformation.

When Multiplication Surpasses Convolution: Computational Efficiency and Optimizations

Thank you for listening!

Any questions?

UNIVERSITÀ DEGLI STUDI FIRENZE

Bibliografy

- ”Learning Convolutional Neural Networks in the Frequency Domain”, by H. Pan, Y. Chen, X. Niu, W. Zhou, and D. Li *Reference Paper*, `http://arxiv.org/abs/2204.06718v10`
- Repository github, `https://github.com/Cappetti99/FML-laboratory.git`
- Wikipedia, *Convolution Theorem*, `https://it.wikipedia.org/wiki/Teorema_di_convoluzione`
- Wikipedia, *Strassen Algoritm*, `https://it.wikipedia.org/wiki/Algoritmo_di_Strassen`