lesson

March 27, 2025

[ ]:
```python
# Phase Correlation Manifold for AI-Generated Image Detection

import numpy as np
import matplotlib.pyplot as plt
from scipy import fftpack
from urllib.request import urlopen
from PIL import Image
import io
import cv2

# Helper function to display images side by side
def plot_images(images, titles, figsize=(15, 5)):
    fig, axes = plt.subplots(1, len(images), figsize=figsize)
    for i, (img, title) in enumerate(zip(images, titles)):
        if len(img.shape) == 2:  # Grayscale
            axes[i].imshow(img, cmap='gray')
        else:  # Color
            axes[i].imshow(img)
        axes[i].set_title(title)
        axes[i].axis('off')
    plt.tight_layout()
    plt.show()
```

[ ]:
```python
# Function to load an image from local file and convert to grayscale
def load_image_from_file(filepath, size=None):
    img = Image.open(filepath)
    if size:
        img = img.resize(size)
    img_array = np.array(img)
    if len(img_array.shape) == 3:  # Color image
        gray_img = cv2.cvtColor(img_array, cv2.COLOR_RGB2GRAY)
        return img_array, gray_img
    else:  # Already grayscale
        return img_array, img_array

# Load a natural image and an AI-generated image from local files
natural_image_path = "starry_night.jpg"
ai_image_path = "ai_image.png"
```

```python
# Load the images
natural_img_color, natural_img = load_image_from_file(natural_image_path,␣
↪size=(512, 512))
ai_img_color, ai_img = load_image_from_file(ai_image_path, size=(512, 512))

# Display the images
plot_images([natural_img_color, ai_img_color],
            ['Natural Image (Starry Night)', 'AI-Generated Image'])
```

[1]:
```python
# What is the Fourier Transform?
# The Fourier Transform converts an image from the spatial domain (pixels)
# to the frequency domain (wave patterns)

def compute_and_display_fourier(img, title):
    # Compute the 2D FFT
    f_transform = fftpack.fft2(img)

    # Shift the zero-frequency component to the center
    f_transform_shifted = fftpack.fftshift(f_transform)

    # Compute the logarithm of the magnitude spectrum (amplitude)
    magnitude_spectrum = np.log(np.abs(f_transform_shifted) + 1)

    # Normalize for better visualization
    magnitude_spectrum_norm = magnitude_spectrum / np.max(magnitude_spectrum)

    # Display original and magnitude spectrum
    plot_images([img, magnitude_spectrum_norm],
                [title, f'Magnitude Spectrum - {title}'])

    return f_transform_shifted

# Compute and display Fourier transform for both images
natural_fourier = compute_and_display_fourier(natural_img, 'Natural Image')
ai_fourier = compute_and_display_fourier(ai_img, 'AI-Generated Image')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 25
     22     return f_transform_shifted
     24 # Compute and display Fourier transform for both images
---> 25 natural_fourier = compute_and_display_fourier(natural_img, 'Natural␣
↪Image')
     26 ai_fourier = compute_and_display_fourier(ai_img, 'AI-Generated Image')
```

```
NameError: name 'natural_img' is not defined
```

[ ]:
```python
# Extracting Amplitude and Phase Components
# Any Fourier transform has two components:
# 1. Amplitude (how strong each frequency component is)
# 2. Phase (where each frequency component is positioned)

def display_amplitude_phase(f_transform, title):
    # Compute amplitude (magnitude)
    amplitude = np.abs(f_transform)
    log_amplitude = np.log(amplitude + 1)
    norm_amplitude = log_amplitude / np.max(log_amplitude)

    # Compute phase
    phase = np.angle(f_transform)

    # Display
    plot_images([norm_amplitude, phase],
                [f'Amplitude - {title}', f'Phase - {title}'])

# Display amplitude and phase for both images
display_amplitude_phase(natural_fourier, 'Natural Image')
display_amplitude_phase(ai_fourier, 'AI-Generated Image')

print("Notice how amplitude tells us how much of each frequency exists,")
print("while phase tells us where those frequencies are positioned in the image.
 ↪")
```

[ ]:
```python
# Phase is actually more important than amplitude for image structure!
# Let's demonstrate this by swapping phase and amplitude between our images

def swap_phase_amplitude(f1, f2):
    # Get magnitude and phase from both Fourier transforms
    mag1, phase1 = np.abs(f1), np.angle(f1)
    mag2, phase2 = np.abs(f2), np.angle(f2)

    # Create new Fourier transforms with swapped components
    # Image 1 magnitude + Image 2 phase
    new_f1 = mag1 * np.exp(1j * phase2)
    # Image 2 magnitude + Image 1 phase
    new_f2 = mag2 * np.exp(1j * phase1)

    # Convert back to spatial domain
    new_img1 = np.real(fftpack.ifft2(fftpack.ifftshift(new_f1)))
    new_img2 = np.real(fftpack.ifft2(fftpack.ifftshift(new_f2)))
```

```
    # Normalize for display
    new_img1 = (new_img1 - np.min(new_img1)) / (np.max(new_img1) - np.
 ↪min(new_img1))
    new_img2 = (new_img2 - np.min(new_img2)) / (np.max(new_img2) - np.
 ↪min(new_img2))

    return new_img1, new_img2

# Swap phase and amplitude between natural and AI images
natural_mag_ai_phase, ai_mag_natural_phase =␣
 ↪swap_phase_amplitude(natural_fourier, ai_fourier)

# Display the original and swapped images
plot_images([natural_img/255, ai_img/255, natural_mag_ai_phase,␣
 ↪ai_mag_natural_phase],
            ['Original Natural Image', 'Original AI Image',
             'Natural Amplitude + AI Phase', 'AI Amplitude + Natural Phase'])

print("Notice how the swapped images look more like the image that contributed␣
 ↪the PHASE!")
print("This shows that phase contains most of the structural information.")
```

[ ]:
```
# Current AI detection methods often focus on power spectrum analysis
# Natural images follow a characteristic 1/f^2 power falloff

def compute_radial_power_spectrum(img):
    # Compute the 2D FFT
    f_transform = fftpack.fft2(img)

    # Shift the zero-frequency component to the center
    f_transform_shifted = fftpack.fftshift(f_transform)

    # Compute power spectrum (squared magnitude)
    power_spectrum = np.abs(f_transform_shifted)**2

    # Calculate radial average (azimuthal average)
    rows, cols = power_spectrum.shape
    center_row, center_col = rows // 2, cols // 2

    # Create a grid of coordinates relative to the center
    y, x = np.indices(power_spectrum.shape)
    r = np.sqrt((x - center_col)**2 + (y - center_row)**2)
    r = r.astype(int)

    # Compute the azimuthally averaged radial profile
    tbin = np.bincount(r.ravel(), weights=power_spectrum.ravel())
    nr = np.bincount(r.ravel())
```

```
    radial_prof = tbin / nr

    # Remove the DC component (zero frequency)
    frequencies = np.arange(1, len(radial_prof))
    radial_prof = radial_prof[1:]

    return frequencies, radial_prof

# Compute power spectra
nat_freq, nat_power = compute_radial_power_spectrum(natural_img)
ai_freq, ai_power = compute_radial_power_spectrum(ai_img)

# Plot power spectra on log-log scale
plt.figure(figsize=(10, 6))
plt.loglog(nat_freq, nat_power, label='Natural Image')
plt.loglog(ai_freq, ai_power, label='AI-Generated Image')

# Add reference line with 1/f^2 slope
ref_freq = np.logspace(0, np.log10(min(len(nat_freq), len(ai_freq))), 100)
ref_power = ref_freq**(-2) * nat_power[0]  # Scale to match the data
plt.loglog(ref_freq, ref_power, 'k--', label='1/f² Reference')

plt.xlabel('Spatial Frequency')
plt.ylabel('Power')
plt.title('Power Spectrum Analysis')
plt.legend()
plt.grid(True, which="both", ls="-", alpha=0.2)
plt.show()

print("Notice how natural images follow a characteristic 1/f² power falloff.")
print("AI-generated images often show deviations from this pattern, especially␣
 ↪at high frequencies.")
```

[ ]:
```
# Phase Correlation is a technique that measures how well aligned two images are
# It's based on the phase component of the Fourier transform

def phase_correlation(img1, img2):
    # Compute Fourier transforms
    f1 = fftpack.fft2(img1)
    f2 = fftpack.fft2(img2)

    # Calculate cross-power spectrum
    cross_power = f1 * np.conj(f2)

    # Normalize to get only the phase information
    cross_power_norm = cross_power / np.abs(cross_power)
```

```
    # Inverse FFT to get the correlation surface
    correlation = np.abs(fftpack.ifft2(cross_power_norm))

    # Shift to center
    correlation = fftpack.fftshift(correlation)

    return correlation

# For demonstration, let's create a slightly shifted version of the natural␣
 ↪image
shifted_img = np.roll(natural_img, shift=(20, 30), axis=(0, 1))

# Compute phase correlation
correlation = phase_correlation(natural_img, shifted_img)

# Display
plt.figure(figsize=(10, 6))
plt.imshow(correlation, cmap='viridis')
plt.colorbar()
plt.title('Phase Correlation')
plt.show()

print("The bright spot indicates the shift between the two images.")
print("Phase correlation measures alignment using only phase information, not␣
 ↪amplitude.")
```

[ ]:
```
# For our novel method, we're interested in phase correlations across scales
# Let's demonstrate by creating a multi-scale representation

def create_gaussian_pyramid(img, levels=3):
    """Create a Gaussian pyramid by repeatedly blurring and downsampling."""
    pyramid = [img.copy()]
    for i in range(levels-1):
        # Blur and downsample
        img = cv2.pyrDown(img)
        pyramid.append(img)
    return pyramid

# Create Gaussian pyramids
natural_pyramid = create_gaussian_pyramid(natural_img)
ai_pyramid = create_gaussian_pyramid(ai_img)

# Display pyramids
natural_pyramid_display = [p for p in natural_pyramid]
ai_pyramid_display = [p for p in ai_pyramid]

plot_images(natural_pyramid_display,
```

```
            [f'Natural Image - Level {i}' for i in range(len(natural_pyramid))])
plot_images(ai_pyramid_display,
            [f'AI Image - Level {i}' for i in range(len(ai_pyramid))])

print("These Gaussian pyramids represent the images at different scales.")
print("Our Phase Correlation Manifold analyzes how phase components relate␣
↪across these scales.")
```

[ ]:
```
# The Phase Correlation Manifold (PCM) method analyzes phase relationships
# across different scales and frequencies in an image

def compute_phase_correlations(pyramid):
    """Compute phase correlations between adjacent scales."""
    correlations = []
    for i in range(len(pyramid)-1):
        # Resize the smaller image to match the larger one
        small_resized = cv2.resize(pyramid[i+1],
                                   (pyramid[i].shape[1], pyramid[i].shape[0]),
                                   interpolation=cv2.INTER_LINEAR)

        # Compute phase correlation
        corr = phase_correlation(pyramid[i], small_resized)
        correlations.append(corr)

    return correlations

# Compute phase correlations for both images
natural_correlations = compute_phase_correlations(natural_pyramid)
ai_correlations = compute_phase_correlations(ai_pyramid)

# Display correlations
plot_images(natural_correlations,
            [f'Natural Image - Correlation {i}/{i+1}' for i in␣
↪range(len(natural_correlations))])
plot_images(ai_correlations,
            [f'AI Image - Correlation {i}/{i+1}' for i in␣
↪range(ai_correlations)])

print("These correlation patterns show how well phase information aligns across␣
↪scales.")
print("Natural images and AI-generated images show different correlation␣
↪patterns.")
```

[ ]:
```
# In the full PCM method, we create higher-dimensional phase correlation tensors
# These capture complex relationships between phase components

def compute_phase_correlation_tensor(img, block_size=8, stride=4):
```

```python
    """Compute a simplified version of the phase correlation tensor."""
    h, w = img.shape
    tensor = []

    # Extract blocks and compute correlations
    for i in range(0, h-block_size, stride):
        for j in range(0, w-block_size, stride):
            # Extract a block
            block = img[i:i+block_size, j:j+block_size]

            # Create a small pyramid for this block
            block_pyramid = create_gaussian_pyramid(block, levels=2)

            # Compute FFT for each level
            ffts = [fftpack.fft2(level) for level in block_pyramid]

            # Extract phase components
            phases = [np.angle(fft) for fft in ffts]

            # Create a simple correlation measure
            # (In the full method, this would be more sophisticated)
            correlation = np.corrcoef(phases[0].flatten(), phases[1].
 ↪flatten())[0, 1]

            tensor.append(correlation)

    return np.array(tensor)

# Compute simplified phase correlation tensors
natural_tensor = compute_phase_correlation_tensor(natural_img)
ai_tensor = compute_phase_correlation_tensor(ai_img)

# Plot histograms of the correlation values
plt.figure(figsize=(10, 6))
plt.hist(natural_tensor, bins=30, alpha=0.7, label='Natural Image')
plt.hist(ai_tensor, bins=30, alpha=0.7, label='AI-Generated Image')
plt.xlabel('Phase Correlation Value')
plt.ylabel('Frequency')
plt.title('Distribution of Phase Correlation Values')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print("The distributions of phase correlation values differ between natural and␣
 ↪AI-generated images.")
print("This is the core insight of the Phase Correlation Manifold method.")
```

[ ]:

```python
# The full PCM method maps these correlation tensors to a lower-dimensional␣
↪manifold
# We'll simulate a simplified version of this process

from sklearn.manifold import TSNE
import numpy as np

# Function to extract more sophisticated features for manifold mapping
def extract_pcm_features(img, block_size=8, stride=4, levels=3):
    """Extract simplified PCM features from an image."""
    h, w = img.shape
    features = []

    # Extract blocks and compute features
    for i in range(0, h-block_size*2, stride):
        for j in range(0, w-block_size*2, stride):
            # Extract overlapping blocks
            block1 = img[i:i+block_size, j:j+block_size]
            block2 = img[i+block_size:i+2*block_size, j:j+block_size]
            block3 = img[i:i+block_size, j+block_size:j+2*block_size]

            # Compute FFTs
            fft1 = fftpack.fft2(block1)
            fft2 = fftpack.fft2(block2)
            fft3 = fftpack.fft2(block3)

            # Extract phase components
            phase1 = np.angle(fft1)
            phase2 = np.angle(fft2)
            phase3 = np.angle(fft3)

            # Compute correlations between phases
            corr12 = np.corrcoef(phase1.flatten(), phase2.flatten())[0, 1]
            corr13 = np.corrcoef(phase1.flatten(), phase3.flatten())[0, 1]
            corr23 = np.corrcoef(phase2.flatten(), phase3.flatten())[0, 1]

            # Add to feature vector
            features.append([corr12, corr13, corr23])

    return np.array(features)

# For demonstration, let's create a synthetic dataset of feature vectors
np.random.seed(42)  # For reproducibility

# Generate synthetic features for "natural" images
# (in reality, these would be computed from a dataset of natural images)
n_natural = 500
```

```
natural_features = np.random.normal(loc=0.6, scale=0.15, size=(n_natural, 3))

# Generate synthetic features for "AI" images
# (in reality, these would be computed from AI-generated images)
n_ai = 500
ai_features = np.random.normal(loc=0.3, scale=0.15, size=(n_ai, 3))

# Combine features and create labels
all_features = np.vstack([natural_features, ai_features])
labels = np.array(['Natural']*n_natural + ['AI']*n_ai)

# Map to a 2D manifold using t-SNE
tsne = TSNE(n_components=2, random_state=42)
embedded = tsne.fit_transform(all_features)

# Split the embedded points by class
natural_points = embedded[:n_natural]
ai_points = embedded[n_natural:]

# Plot the manifold
plt.figure(figsize=(10, 8))
plt.scatter(natural_points[:, 0], natural_points[:, 1], color='blue', alpha=0.
↪6, label='Natural Images')
plt.scatter(ai_points[:, 0], ai_points[:, 1], color='red', alpha=0.6,␣
↪label='AI-Generated Images')
plt.title('Simplified Phase Correlation Manifold (Synthetic Data)')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print("In the full PCM method, we map the high-dimensional phase correlation␣
↪tensors")
print("to a lower-dimensional manifold where natural and AI-generated images␣
↪form distinct clusters.")
```

[ ]:

```
# The full PCM method would use topological data analysis techniques
# like persistent homology to analyze the manifold structure
# This is a simplified visualization of what that might look like

from scipy.spatial import distance
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import minimum_spanning_tree

def visualize_simplified_topology(points, labels, threshold=2.0):
    """Create a simplified visualization of topological structure."""
```

```
    # Compute distance matrix
    dist_matrix = distance.pdist(points)
    dist_matrix = distance.squareform(dist_matrix)

    # Threshold the distances to create a graph
    graph = dist_matrix.copy()
    graph[graph > threshold] = 0

    # Compute minimum spanning tree for visualization
    mst = minimum_spanning_tree(csr_matrix(graph)).toarray()

    # Plot points and connections
    plt.figure(figsize=(12, 10))

    # Plot the points
    natural_idx = labels == 'Natural'
    ai_idx = labels == 'AI'
    plt.scatter(points[natural_idx, 0], points[natural_idx, 1],
                color='blue', alpha=0.6, label='Natural Images')
    plt.scatter(points[ai_idx, 0], points[ai_idx, 1],
                color='red', alpha=0.6, label='AI-Generated Images')

    # Plot the connections
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            if mst[i, j] > 0 or mst[j, i] > 0:
                plt.plot([points[i, 0], points[j, 0]],
                         [points[i, 1], points[j, 1]],
                         'k-', alpha=0.2)

    plt.title('Simplified Topological Structure')
    plt.xlabel('Dimension 1')
    plt.ylabel('Dimension 2')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Visualize the simplified topology
visualize_simplified_topology(embedded, labels, threshold=5.0)

print("The full PCM method analyzes the topological structure of the phase␣
 ↪correlation manifold.")
print("This captures higher-order relationships between phase components that␣
 ↪simple statistical")
print("measures miss, revealing fundamental differences between natural and␣
 ↪AI-generated images.")
```

[ ]:

```python
# Here's a simplified outline of the complete Phase Correlation Manifold method

def pcm_method_outline(img):
    """Outline of the complete PCM method."""
    # 1. Compute the 2D Fourier transform
    f_transform = fftpack.fft2(img)

    # 2. Extract the phase component
    phase = np.angle(f_transform)

    # 3. Create multi-scale representation using Gaussian pyramid
    pyramid = create_gaussian_pyramid(img)

    # 4. Compute phase correlation tensor across scales and spatial locations
    # (This would be much more sophisticated in the full method)
    tensor = compute_phase_correlation_tensor(img)

    # 5. Map to a lower-dimensional manifold
    # (In practice, this would use features from many images)
    manifold = tsne.fit_transform(tensor.reshape(-1, 1))

    # 6. Analyze topological properties of the manifold
    # (This would use persistent homology and other TDA techniques)

    # 7. Extract topological features for classification

    # 8. Compare to reference distributions for natural and AI-generated images

    # Return a detection score
    return 0.5  # Placeholder

print("The complete Phase Correlation Manifold method would:")
print("1. Extract phase components from the image's Fourier transform")
print("2. Analyze phase correlations across different scales and frequencies")
print("3. Map these correlations to a lower-dimensional manifold")
print("4. Apply topological data analysis to the manifold")
print("5. Extract features that distinguish natural from AI-generated images")
print("")
print("This approach captures fundamental differences in how phase components")
print("relate to each other in natural versus AI-generated images,")
print("providing a robust method for detection that's grounded in the physics")
print("of natural image formation.")
```