🧠 Lab 2.1: Understanding CNN Layers with VGG16

📖 TEACHER HANDBOOK — Full Solutions & Teaching Notes

Module 3: Computer Vision and Image Processing
B-Tech AI Specialization | Chitkara University | February 2026

> **For instructors only.** This notebook contains full working code, expected outputs, common mistakes to watch for, model answers for reflections, and an assessment rubric.

🗺️ Pedagogical Flow

This lab uses a **Predict → Code → Reveal → Explore** structure designed to activate prior knowledge before introducing new concepts.

Phase	Purpose	Watch For
🤔 Predict	Surface misconceptions early	Students often think all layers look like blurry images
💻 Code	Hands-on construction of understanding	Forgetting `preprocess_input` is the #1 error
💡 Reveal	Validate and correct independently	Encourage students to check *before* asking you
🕹️ Explore	Develop intuition through play	Give 5–10 min free exploration before moving on

Key insight to reinforce: Feature maps are NOT "what the layer sees" — they are "what activated each filter". The distinction matters for interpretability.

⚙️ Setup

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.models import Model

from PIL import Image
import requests
from io import BytesIO

import ipywidgets as widgets
from IPython.display import display, HTML, Code, clear_output

print(f"TensorFlow: {tf.__version__}")
```

```
TensorFlow: 2.19.0
```

Task 1: Load VGG16 ✅ FULL SOLUTION

👨‍🏫 Teaching Notes

Why `preprocess_input` matters (explain this verbally to class):
VGG16 was trained in Caffe with BGR channel order and pixels mean-subtracted using ImageNet statistics (B=103.939, G=116.779, R=123.68). Without this, activations are meaningless — the model has never seen raw [0,255] RGB pixels. This is a great opportunity to discuss *training distribution* as a concept.

Why `include_top=True` here:
We're not fine-tuning — we just want the pre-trained filters. `include_top=False` would drop the Dense layers but we don't need that distinction yet (Lab 2.2 covers it for transfer learning).

⚠️ Common Student Mistakes

- `weights=None` — produces random noise; hard to catch visually (demo this as the extension!)
- Wrong `input_shape` — causes a shape mismatch error later when processing 224×224 images
- Not running `model.summary()` and then not knowing layer names for Task 2

```
# Full solution — Task 1
```

```
model = VGG16(weights='imagenet', include_top=True, input_shape=(224, 224, 3))
model.summary()

# Key stats to point out during demo:
total_params = model.count_params()
print(f"\nTotal parameters: {total_params:,}  (~{total_params/1e6:.0f}M)")
print(f"Number of layers:  {len(model.layers)}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ord
553467096/553467096 ━━━━━━━━━━━━━━━━━━━━ 6s 0us/step
Model: "vgg16"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102,764,544
fc2 (Dense)	(None, 4096)	16,781,312
predictions (Dense)	(None, 1000)	4,097,000

```
 Total params: 138,357,544 (527.79 MB)
 Trainable params: 138,357,544 (527.79 MB)
 Non-trainable params: 0 (0.00 B)

Total parameters: 138,357,544  (~138M)
Number of layers:  23
```

Expected Output Highlights

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1,792
...
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2,359,808
...
Total params: 138,357,544
```

Point out the pattern: spatial size halves after each MaxPooling, filter count doubles each block. This is the characteristic VGG design.

Task 2: Select Layers ✅ FULL SOLUTION

👩‍🏫 Teaching Notes

The intuition to build:

```
Block 1 (224×224): Like a Gabor filter bank — detects oriented edges
Block 3 ( 56×56): Detects textures, repetitive patterns, corners
Block 5 ( 14×14): Detects semantic concepts — faces, objects, parts
```

Draw this on the board as a funnel: wide (spatial detail) ⟶ narrow (semantic meaning).

Discussion question to ask class before they code:
"If block1 detects edges and block5 detects dog faces — what does block3 detect?" ⟶ Leads to textures / structural patterns.

⚠️ Common Mistake

Students often pick layers from the Dense section (`predictions`, `fc1`). Remind them we want *spatial* feature maps — Dense layers have no spatial dimensions.

```
# Full solution — Task 2
layer_names = [
    'block1_conv1',   # Early:  224×224 spatial, 64 filters  — edges & colors
    'block3_conv3',   # Middle:  56×56 spatial, 256 filters  — textures & patterns
    'block5_conv3',   # Deep:    14×14 spatial, 512 filters  — semantic features
]

# Print metadata for each chosen layer
print(f"{'Layer':<20} {'Output Shape':<30} {'# Filters'}")
print("−" * 65)
for name in layer_names:
    layer = model.get_layer(name)
    out_shape = str(layer.output.shape)
    n_filters = layer.output.shape[−1]
    print(f"{name:<20} {out_shape:<30} {n_filters}")
```

```
Layer                Output Shape                  # Filters
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
block1_conv1         (None, 224, 224, 64)          64
block3_conv3         (None, 56, 56, 256)           256
block5_conv3         (None, 14, 14, 512)           512
```

Task 3: Build Feature Extractors ✅ FULL SOLUTION

👩‍🏫 Teaching Notes

This is the most conceptually novel task. The key insight: **Keras models are computation graphs**, and you can "cut" them at any point by specifying a different output node. The weights are shared — no duplication.

Draw this on the board:

```
model.input ──┬──→ [block1_conv1] → ... → [predictions]
              │          ↑ tap here  →  extractor_1
              └──────────────────────→ extractor_2 at block3
```

Analogy for students: It's like putting a voltmeter probe at different points in a circuit — the circuit doesn't change, you just read the signal at different locations.

```
# Full solution — Task 3
feature_extractors = {}

for name in layer_names:
    layer_output = model.get_layer(name).output
    feature_extractors[name] = Model(inputs=model.input, outputs=layer_output)

# Verify
print(f"{'Layer':<20} {'Input Shape':<30} {'Output Shape'}")
print("−" * 75)
for name, extractor in feature_extractors.items():
    print(f"{name:<20} {str(extractor.input_shape):<30} {extractor.output_shape}")
```

```
Layer                Input Shape                   Output Shape
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
block1_conv1         (None, 224, 224, 3)           (None, 224, 224, 64)
block3_conv3         (None, 224, 224, 3)           (None, 56, 56, 256)
block5_conv3         (None, 224, 224, 3)           (None, 14, 14, 512)
```

Task 4: Process Image ✅ FULL SOLUTION

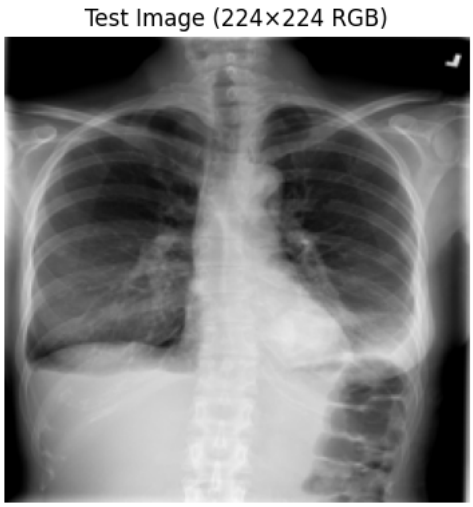
👩‍🏫 Teaching Notes

Walk through each step explicitly — these are reusable patterns students will use in every CV project:

1. **PIL → numpy**: PIL Images are not natively usable by Keras
2. **expand_dims**: Every Keras model expects a batch dimension, even for a single image
3. **preprocess_input**: Normalizes to the training distribution (subtract ImageNet mean)

Fun demo: Print `img_preprocessed.min()` and `img_preprocessed.max()`. Students are surprised the range is roughly [-123, 151] rather than [0, 1]. Explain that normalization to [0,1] is *not* what VGG16 was trained with — different architectures use different normalizations.

```
# Full solution — Task 4
# Normal chest X-ray (public domain, NIH)
img_url = "https://source.roboflow.com/ksz9vtvaNta4l7TAgo6H9Tg0kwy2/0ho01xPxjIndnur6Go1h/original.jpg"

# Pneumonia X-ray (from NIH open dataset)
pneumonia_url = "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcR-jl56g2ePhMfs1s8dTGTtNZStFvLb56G-DQ&s"

response = requests.get(img_url)
img = Image.open(BytesIO(response.content)).convert('RGB')

img_resized      = img.resize((224, 224))
img_array        = np.array(img_resized, dtype=np.float32)
img_batch        = np.expand_dims(img_array, axis=0)       # (1, 224, 224, 3)
img_preprocessed = preprocess_input(img_batch.copy())       # in-place, so copy first

plt.figure(figsize=(4, 4))
plt.imshow(img_resized)
plt.title("Test Image (224×224 RGB)")
plt.axis('off')
plt.tight_layout()
plt.show()

print(f"img_array shape        : {img_array.shape}")
print(f"img_batch shape        : {img_batch.shape}")
print(f"img_preprocessed shape : {img_preprocessed.shape}")
print(f"Pixel range after preprocess: [{img_preprocessed.min():.1f}, {img_preprocessed.max():.1f}]")
print("\n⚠️  Range is NOT [0,1] — it's shifted by ImageNet channel means!")
```




```
img_array shape        : (224, 224, 3)
img_batch shape        : (1, 224, 224, 3)
img_preprocessed shape : (1, 224, 224, 3)
Pixel range after preprocess: [-123.7, 145.1]

⚠️  Range is NOT [0,1] — it's shifted by ImageNet channel means!
```

Task 5: Extract Feature Maps ✅ FULL SOLUTION

```
# Full solution — Task 5
all_feature_maps = {}

for name, extractor in feature_extractors.items():
    fmap = extractor.predict(img_preprocessed, verbose=0)
    all_feature_maps[name] = fmap
```

```
    # Compute activation stats — useful for class discussion
    mean_per_filter = fmap[0].mean(axis=(0, 1))
    pct_inactive    = np.mean(mean_per_filter < 0.01) * 100

    print(f"\n{name}:")
    print(f"  Shape    : {fmap.shape}  (batch, H, W, filters)")
    print(f"  Range    : [{fmap.min():.3f}, {fmap.max():.3f}]")
    print(f"  % filters near-zero : {pct_inactive:.0f}%  ← increases in deep layers!")
```

```
block1_conv1:
  Shape    : (1, 224, 224, 64)  (batch, H, W, filters)
  Range    : [0.000, 787.800]
  % filters near-zero : 0%  ← increases in deep layers!

block3_conv3:
  Shape    : (1, 56, 56, 256)  (batch, H, W, filters)
  Range    : [0.000, 12462.562]
  % filters near-zero : 0%  ← increases in deep layers!

block5_conv3:
  Shape    : (1, 14, 14, 512)  (batch, H, W, filters)
  Range    : [0.000, 83.979]
  % filters near-zero : 15%  ← increases in deep layers!
```

👨‍🏫 Teaching Notes: The "Dead Filter" Phenomenon

Point out the `% near-zero` stat. In `block5_conv3`, often 60–80% of filters are essentially inactive for any given image. This is not a bug — it's by design.

Explanation: Deep filters are *highly specialized*. A filter that detects "dog ear" won't activate for a wheel, a tree, or a sky region. Most filters in the last block encode specific semantic concepts that happen to not appear in your test image.

Ask the class: "For a chest X-ray, which filters would light up in block5 that wouldn't light up for a dog photo? How does that relate to why the model can classify diseases?"

Task 6: Interactive Explorer ✅ FULL SOLUTION

The explorer code is identical in both notebooks. In the teacher version, use it to **demonstrate live in class** before students run it themselves.

👨‍🏫 Demo Script (5 minutes)

1. Start at `block1_conv1`, Filter 0 → Point out: "looks like an edge detector"
2. Scroll to Filter 1, 2, 3 → "Each filter learned a different edge orientation"
3. Switch to `block3_conv3` → "More abstract — you can see texture-like patterns"
4. Switch to `block5_conv3` → "Most are dark — only a few respond. Look at the most active filter."
5. Change colormap to `gray` → "Same data, different perception. Colormap choice matters for interpretability"

```
# Identical to student notebook — paste and run for live demo
layer_select = widgets.Dropdown(
    options=layer_names, value=layer_names[0], description='Layer:',
    style={'description_width': '60px'}, layout=widgets.Layout(width='280px')
)
filter_slider = widgets.IntSlider(
    min=0, max=63, step=1, value=0, description='Filter #:',
    style={'description_width': '70px'}, layout=widgets.Layout(width='380px'),
    continuous_update=False
)
cmap_select = widgets.Dropdown(
    options=['viridis', 'plasma', 'inferno', 'magma', 'gray', 'hot', 'RdBu_r'],
    value='viridis', description='Colormap:',
    style={'description_width': '80px'}, layout=widgets.Layout(width='220px')
)
stats_out = widgets.Output()
plot_out  = widgets.Output()

def update_explorer(change=None):
    layer_name = layer_select.value
    filter_idx = filter_slider.value
    cmap       = cmap_select.value
    fmaps      = all_feature_maps[layer_name]
    num_filters = fmaps.shape[-1]
    filter_slider.max = num_filters - 1
    filter_idx = min(filter_idx, num_filters - 1)
    activation = fmaps[0, :, :, filter_idx]
    mean_per_filter = fmaps[0].mean(axis=(0, 1))
```

```
        top_filter      = int(np.argmax(mean_per_filter))
        pct_active      = float(np.mean(activation > 0)) * 100

        with stats_out:
            stats_out.clear_output(wait=True)
            display(HTML(
                f'<div style="background:#f0f4ff;padding:10px;border-radius:6px;'
                f'font-family:monospace;font-size:13px;line-height:1.7">'
                f'<b>Layer:</b>  {layer_name}<br>'
                f'<b>Spatial:</b> {fmaps.shape[1]}×{fmaps.shape[2]} px<br>'
                f'<b>Filters:</b> {num_filters} total<br>'
                f'<hr style="margin:5px 0">'
                f'<b>Filter {filter_idx}:</b><br>'
                f'  Min: {activation.min():.3f}<br>'
                f'  Max: {activation.max():.3f}<br>'
                f'  Mean: {activation.mean():.3f}<br>'
                f'  % Active: {pct_active:.1f}%<br>'
                f'<hr style="margin:5px 0">'
                f'<b>🏆 Most active:</b> #{top_filter} '
                f'(mean={mean_per_filter[top_filter]:.3f})'
                f'</div>'
            ))

        with plot_out:
            plot_out.clear_output(wait=True)
            fig, axes = plt.subplots(1, 3, figsize=(14, 4.5))
            axes[0].imshow(img_resized)
            axes[0].set_title('Original Image', fontsize=12)
            axes[0].axis('off')
            im = axes[1].imshow(activation, cmap=cmap)
            axes[1].set_title(f'{layer_name}\nFilter #{filter_idx} | mean={activation.mean():.3f}', fontsize=11)
            axes[1].axis('off')
            plt.colorbar(im, ax=axes[1], fraction=0.046, pad=0.04)
            top_activation = fmaps[0, :, :, top_filter]
            im2 = axes[2].imshow(top_activation, cmap=cmap)
            axes[2].set_title(f'🏆 Most Active #{top_filter}\nmean={mean_per_filter[top_filter]:.3f}', fontsize=11)
            axes[2].axis('off')
            plt.colorbar(im2, ax=axes[2], fraction=0.046, pad=0.04)
            plt.suptitle(f'Feature Map Explorer — {layer_name}', fontsize=13, fontweight='bold')
            plt.tight_layout()
            plt.show()

layer_select.observe(update_explorer, names='value')
filter_slider.observe(update_explorer, names='value')
cmap_select.observe(update_explorer, names='value')
controls = widgets.VBox([
    widgets.HTML('<b>🎛 Controls:</b>'),
    widgets.HBox([layer_select, cmap_select]),
    filter_slider,
])
display(widgets.HBox([
    widgets.VBox([controls, stats_out], layout=widgets.Layout(width='310px')),
    plot_out
]))
update_explorer()
```

🎛 **Controls:**

Layer: block1_conv1

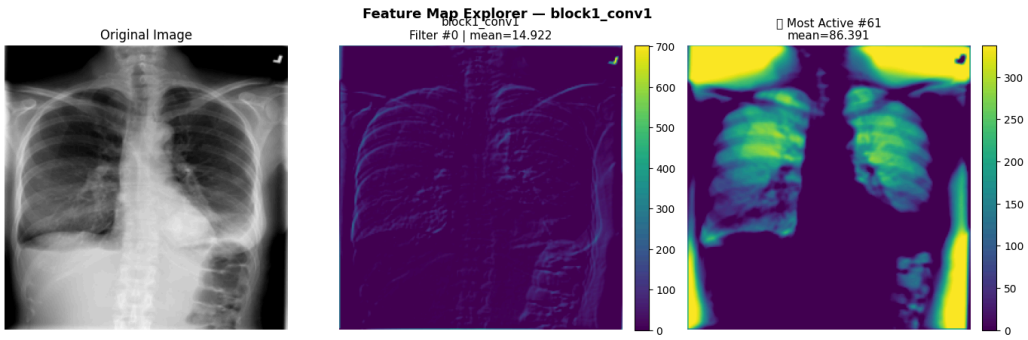
Filter #: 0

```
/tmp/ipython-input-4281922611.py:68: UserWarning: Glyph 127942 (\N{TROPHY}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
/usr/local/lib/python3.12/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 127942 (\N{TROPHY}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```

Layer: block1_conv1
Spatial: 224×224 px
Filters: 64 total

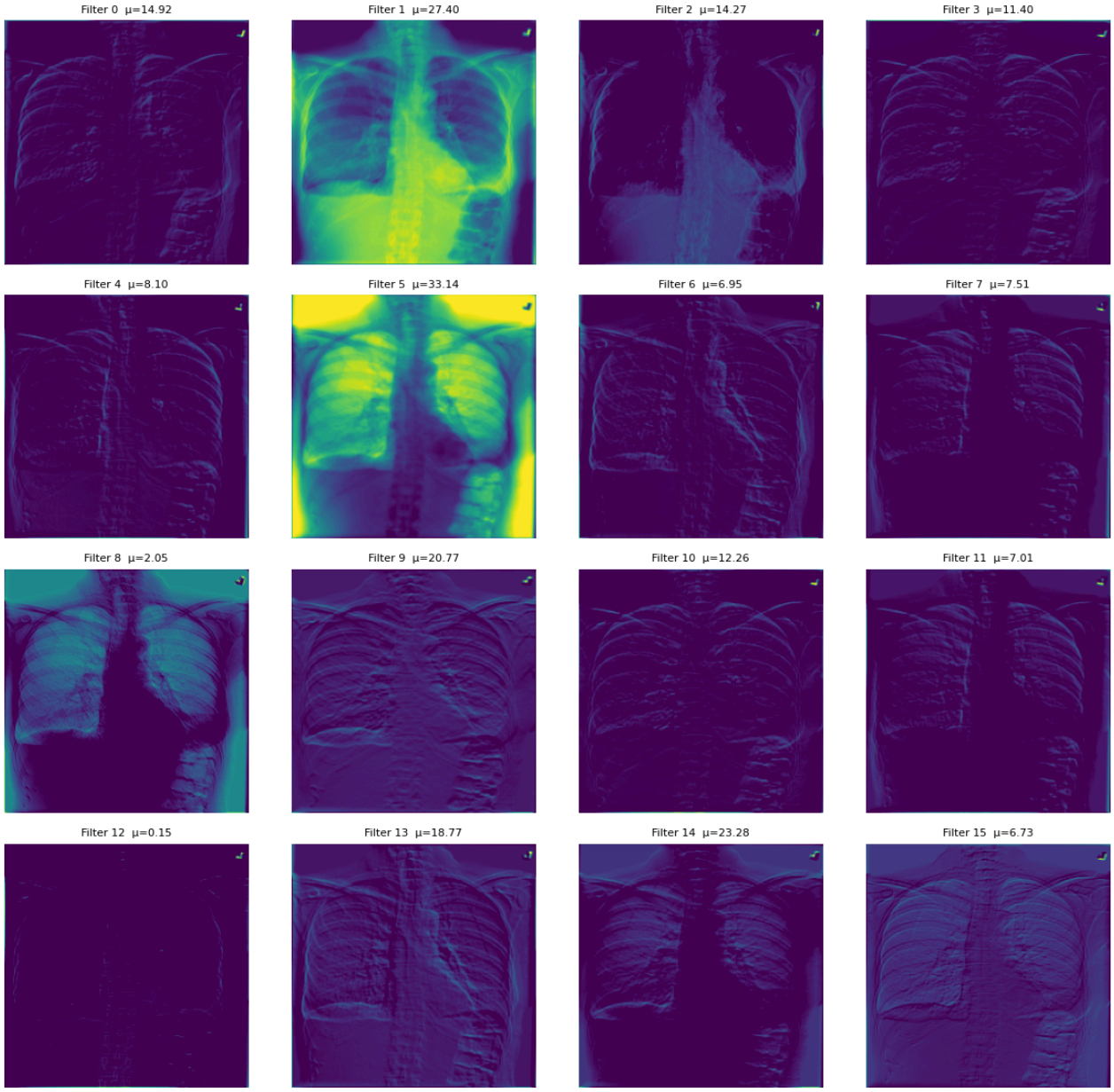
Filter 0:
Min: 0.000
Max: 702.443
Mean: 14.922
% Active: 43.8%

🏆 **Most active:** #61 (mean=86.391)



Task 7: Grid Visualizations ✅ FULL SOLUTION

```
def save_grid(layer_name, cmap='viridis'):
    fmaps = all_feature_maps[layer_name]
    fig, axes = plt.subplots(4, 4, figsize=(12, 12))
    fig.suptitle(
        f'Feature Maps — {layer_name}\n'
        f'Spatial: {fmaps.shape[1]}×{fmaps.shape[2]} | Filters: 16 of {fmaps.shape[3]}',
        fontsize=13, fontweight='bold'
    )
    for i, ax in enumerate(axes.flat):
        activation = fmaps[0, :, :, i]
        ax.imshow(activation, cmap=cmap)
        ax.set_title(f'Filter {i}  μ={activation.mean():.2f}', fontsize=8)
        ax.axis('off')
    plt.tight_layout()
    filename = f'feature_maps_{layer_name}.png'
    plt.savefig(filename, dpi=100, bbox_inches='tight')
    plt.show()
    print(f"✅ Saved: {filename}")

for name in layer_names:
    save_grid(name)
```



✅ Saved: feature_maps_block1_conv1.png