Computer Vision Learning Path: Learn by Doing

Philosophy

You'll solve small, concrete problems that force you to learn concepts. Each problem introduces new fundamentals. You'll implement both classical and deep learning approaches to understand trade-offs.

Format: Problem statement → solve it → learn concepts in context → move to next problem

---

Week 1: Foundation Problems

Problem 1: Image Loading & Exploration (Day 1)

Goal: Understand images as numerical arrays

Task:

- Load a simple image (download any image from internet: cat, dog, landscape)
- Display its shape, data type, value range
- Visualize the image and individual color channels
- Convert to grayscale, HSV color space
- Display histograms of each channel

Concepts You'll Learn:

- Images as 3D arrays (Height × Width × Channels)
- Data types (uint8 vs float32) and value ranges
- Color spaces (RGB, Grayscale, HSV)
- How to manipulate arrays with NumPy

Tools:

- OpenCV or Pillow for loading
- NumPy for array operations
- Matplotlib for visualization

Deliverable: Script showing image properties and visualizations


python

```python
# Starter code structure:
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg')
# Explore: shape, dtype, min/max values
# Convert: grayscale, HSV
# Plot: original, channels, histogram
```

Time: 1-2 hours

Problem 2: Image Filtering & Edge Detection - Classical Approach (Day 2)

Goal: Understand convolution through manual filtering

Task:

- Apply Gaussian blur to image (classical filter)
- Manually create a simple edge detection kernel ([-1, 0, 1]) and understand convolution
- Use Canny edge detector (automated edge detection)
- Compare: manual kernel vs Canny results
- Experiment: how do kernel size and parameters affect output?

Concepts You'll Learn:

- Convolution operation (sliding kernel, dot product)
- How filters detect features (edges, blurs)
- Padding and stride effects
- Canny algorithm (Gaussian blur + gradient + thresholding)

Tools: OpenCV (cv2.filter2D, cv2.Canny, cv2.GaussianBlur)

Why This Matters: Understanding convolution manually makes deep learning convolutions intuitive later

Deliverable: Comparison of filtering techniques with visualizations

python

```python
# Starter structure:
import cv2
import numpy as np

img = cv2.imread('image.jpg', 0)  # grayscale

# 1. Gaussian blur
blurred = cv2.GaussianBlur(img, (5, 5), 0)

# 2. Manual edge kernel
kernel = np.array([[-1, 0, 1]])
edges_manual = cv2.filter2D(img, -1, kernel)

# 3. Canny edge detection
edges_canny = cv2.Canny(img, 100, 200)

# Visualize and compare
```

Time: 2-3 hours

Problem 3: Image Classification - Classical Approach (Day 2-3)

Goal: Classify images without deep learning

Task:

- Collect 2 classes of images (e.g., cats vs dogs, or any 2 categories - aim for ~20-30 images per class)
- Extract features using Histogram of Oriented Gradients (HOG)
- Train a simple classifier (SVM from scikit-learn)
- Test accuracy on validation set
- Visualize what features the classifier learned

Concepts You'll Learn:

- Feature engineering: Creating meaningful representations
- HOG features: Detecting shapes and edges
- Training/validation splits
- Classification accuracy
- How classical methods work end-to-end

Tools: scikit-image (HOG extraction), scikit-learn (SVM classifier)

Deliverable: Classifier that can distinguish 2 classes with accuracy metric


python

```
# Starter structure:
from skimage.feature import hog
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import os

# 1. Load images and extract HOG features
X_train, y_train = [], []
for img_path in train_images:
    img = cv2.imread(img_path, 0)
    features = hog(img, pixels_per_cell=(8, 8))
    X_train.append(features)
    y_train.append(label)

# 2. Train SVM
clf = SVC(kernel='rbf')
clf.fit(X_train, y_train)

# 3. Evaluate
predictions = clf.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

Time: 3-4 hours

Problem 4: Image Classification - Deep Learning Approach (Day 3-4)

Goal: Solve same problem as Problem 3 using neural network

Task:

- Use PyTorch to build simple CNN (2-3 conv layers)
- Train on same 2-class dataset
- Compare accuracy with classical approach
- Experiment: what happens if you change network architecture?
- Visualize what filters learned (what edges/patterns do early layers detect?)

Concepts You'll Learn:

- CNN architecture (conv layers, pooling, fully connected)
- Training loop (forward pass, loss calculation, backpropagation, weight updates)
- Overfitting vs good generalization
- How to visualize learned features
- Why deep learning often beats classical features

Tools: PyTorch, torchvision

Key Difference from Classical: Instead of hand-engineering HOG features, let network learn features automatically

Deliverable: Neural network classifier with comparison to classical method

python

```python
# Starter structure:
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset

class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.fc = nn.Linear(32 * 56 * 56, 2)  # 2 classes

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# Training loop:
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(10):
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Time: 4-5 hours

Problem 5: Image Preprocessing & Data Augmentation (Day 4)

Goal: Understand why preprocessing matters and how augmentation helps

Task:

- Take your 2-class dataset
- Train network WITHOUT preprocessing (raw images)
- Train network WITH preprocessing (normalization)
- Train network WITH data augmentation (random rotations, flips, crops)
- Compare accuracies
- Understand: why does preprocessing help? Why augmentation?

Concepts You'll Learn:

- Normalization: standardizing input ranges for neural networks
- Data augmentation: creating artificial variations to prevent overfitting
- Batch processing
- Why preprocessing/augmentation matters for model performance

Tools: PyTorch transforms, Albumentations (for augmentation)

Deliverable: Comparison showing impact of preprocessing and augmentation

python

```
# Starter structure:
from torchvision import transforms

# No preprocessing
transform_none = transforms.Compose([
    transforms.ToTensor(),
])

# With preprocessing
transform_prep = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5],
                         std=[0.5, 0.5, 0.5]),
])

# With augmentation
transform_aug = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5],
                         std=[0.5, 0.5, 0.5]),
])

# Train 3 models with different transforms, compare
```

Time: 2-3 hours

Problem 6: Transfer Learning (Day 5)

Goal: Use pretrained model instead of training from scratch

Task:

- Take your 2-class dataset (likely small)
- Train from scratch (like you did in Problem 4)
- Load pretrained ResNet18 from torchvision
- Freeze early layers, replace last layer for 2 classes
- Fine-tune on your dataset
- Compare: accuracy, training time, how much data you need

Concepts You'll Learn:

- Transfer learning: why pretrained models help
- Fine-tuning: adapting pretrained weights to your task
- What early vs late layers learn
- Trade-offs: training speed vs model size

Deliverable: Comparison showing transfer learning beats training from scratch on small data

python

```python
# Starter structure:
import torchvision.models as models

# Option 1: Train from scratch
model_scratch = models.resnet18(pretrained=False)
# ... train ...

# Option 2: Transfer learning
model_transfer = models.resnet18(pretrained=True)
# Freeze early layers
for param in model_transfer.parameters():
    param.requires_grad = False
# Replace last layer
model_transfer.fc = nn.Linear(512, 2)
# Fine-tune (only last layer learns)
# ... train ...
```

Time: 3-4 hours

End of Week 1: You've learned core concepts by solving real problems. You understand:

- How images work at numerical level
- Classical filtering and feature extraction
- Building neural networks from scratch
- Transfer learning
- Impact of preprocessing and augmentation

Week 2: Object Detection Problems

Problem 7: Object Detection - Bounding Box Basics (Day 1)

Goal: Understand bounding boxes and detection evaluation

Task:

- Collect images with 1-2 objects in each (e.g., photos of animals/objects)
- Manually annotate bounding boxes (create dataset: image → object coordinates)
- Use a pretrained YOLO model to detect objects
- Compare predicted boxes to ground truth boxes
- Calculate IoU (Intersection over Union) for each prediction
- Understand: what does IoU tell you about detection quality?

Concepts You'll Learn:

- Bounding box representation (x1, y1, x2, y2)

- IoU metric
- How to evaluate object detection
- Practical use of pretrained detection models

Tools: YOLOv8 (pretrained), OpenCV (drawing boxes), custom IoU calculation

Deliverable: Visualization showing predicted vs ground truth boxes with IoU scores

python

```python
# Starter structure:
from ultralytics import YOLO
import cv2

# Load pretrained YOLO
model = YOLO('yolov8n.pt')

# Run inference
results = model.predict('image.jpg')

# Extract predictions
for result in results:
    boxes = result.boxes
    for box in boxes:
        x1, y1, x2, y2 = box.xyxy[0]
        conf = box.conf[0]
        # Calculate IoU with ground truth
        # Visualize
```

Time: 2-3 hours

Problem 8: Object Detection - Classical Approach (Day 1-2)

Goal: Detect objects using classical methods before deep learning

Task:

- Use same images from Problem 7
- Detect objects using feature matching (ORB features + contour detection)
- Draw bounding boxes around detected objects
- Compare classical detection to YOLO
- Pros/cons of each approach?

Concepts You'll Learn:

- Classical object detection: feature matching + templates
- When classical methods fail
- Why deep learning is better for general object detection
- Trade-offs: speed vs accuracy

Tools: OpenCV (ORB, contours, template matching)

Deliverable: Side-by-side comparison of classical vs deep learning detection

python

```python
# Starter structure:
import cv2

img = cv2.imread('image.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# 1. Template matching approach
template = cv2.imread('template.jpg', 0)
result = cv2.matchTemplate(gray, template, cv2.TM_CCOEFF)
# Find matches

# 2. Contour-based approach
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
_, thresh = cv2.threshold(blurred, 127, 255, cv2.THRESH_BINARY)
contours, _ = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
# Get bounding boxes from contours
```

Time: 3-4 hours

Problem 9: Object Detection Fine-tuning (Day 2-3)

Goal: Train custom YOLO model on your own data

Task:

- Annotate 100-200 images with bounding boxes (use CVAT or Roboflow for easier labeling)
- Download YOLOv8 and fine-tune on your dataset
- Evaluate on validation set (mAP metric)
- Compare: pretrained YOLO vs your fine-tuned YOLO on your specific objects
- Understand: when does fine-tuning help? When is pretrained enough?

Concepts You'll Learn:

- How to prepare dataset for object detection (annotation format)
- Fine-tuning detection models
- mAP metric for object detection
- Train/validation/test splits
- When to fine-tune vs use pretrained

Tools: YOLOv8, Roboflow (optional, for annotation help)

Deliverable: Fine-tuned detection model with performance metrics

python

```python
# Starter structure:
from ultralytics import YOLO

# Load pretrained model
model = YOLO('yolov8n.yaml')  # nano model
# or YOLO('yolov8n.pt')  # pretrained weights

# Train on custom dataset
# Dataset structure: dataset/
#   images/
#     train/
#     val/
#   labels/
#     train/
#     val/

results = model.train(
    data='dataset.yaml',  # YAML file describing dataset
    epochs=50,
    imgsz=640,
    device=0,  # GPU
)

# Evaluate
metrics = model.val()
print(f"mAP: {metrics.box.map50}")
```

Time: 4-5 hours (annotation is slow, but important!)

Problem 10: Non-Maximum Suppression & Post-processing (Day 3-4)

Goal: Clean up raw detections into final predictions

Task:

- Take raw detections from YOLO (many overlapping boxes)
- Implement NMS (Non-Maximum Suppression) from scratch
- Understand how NMS threshold affects results
- Experiment: IoU threshold = 0.3 vs 0.5 vs 0.9, what changes?
- Understand: why is NMS necessary?

Concepts You'll Learn:

- NMS algorithm (keep best box, remove overlapping ones)

- How detection models output many redundant predictions
- Post-processing pipeline
- Confidence thresholding

Deliverable: Custom NMS implementation + visualization

python

```python
# Starter structure (implement NMS):
import numpy as np

def nms(boxes, scores, iou_threshold=0.5):
    """
    boxes: Nx4 array [[x1,y1,x2,y2], ...]
    scores: N array [confidence, ...]

    Returns: indices of boxes to keep
    """
    # 1. Sort by confidence (descending)
    sorted_indices = np.argsort(-scores)

    # 2. Keep track of boxes to keep
    keep = []

    # 3. For each box (in descending confidence order):
    #    - Add to keep list
    #    - Remove all other boxes that overlap > threshold

    return keep

# Test on your detections
```

Time: 2-3 hours

Problem 11: Evaluation Metrics & Debugging Detection (Day 4-5)

Goal: Understand detection evaluation deeply

Task:

- Take your fine-tuned detector from Problem 9
- Collect validation results (predicted boxes, ground truth boxes)
- Calculate mAP (mean Average Precision) manually
- Analyze failure cases: which images/objects does it fail on?
- Visualize precision-recall curve
- Confusion analysis: what does it confuse?

Concepts You'll Learn:

- mAP calculation in detail
- Precision vs recall trade-off
- How to debug detection failures
- When model needs more data vs different architecture

Deliverable: Detailed analysis of detection performance

python

```python
# Starter structure:
def calculate_iou(box1, box2):
    """Calculate IoU between two boxes"""
    intersection = ...
    union = ...
    return intersection / union

def calculate_map(predictions, ground_truth, iou_threshold=0.5):
    """Calculate mAP from predictions and ground truth"""
    # For each confidence threshold:
    # - Calculate precision: TP / (TP + FP)
    # - Calculate recall: TP / (TP + FN)
    # - Plot precision-recall curve
    # - Calculate area under curve = AP

    return map_value

# Analyze failures
# Visualize high-confidence wrong predictions
# Group failures by object class, image size, etc.
```

Time: 3-4 hours

Problem 12: Semantic Segmentation Basics (Day 5)

Goal: Go beyond bounding boxes to pixel-level classification

Task:

- Collect images where you want to segment regions (e.g., sky vs ground)
- Annotate pixel-level masks (simpler: create binary masks)
- Use pretrained U-Net or DeepLab for segmentation
- Understand IoU metric for segmentation (same concept as detection, but per-pixel)
- Compare: bounding box vs segmentation for your use case

Concepts You'll Learn:

- Pixel-level classification
- Segmentation architectures (encoder-decoder)
- Mask creation and visualization
- When you need segmentation vs detection

Tools: PyTorch, segmentation models from segmentation-models-pytorch

Deliverable: Segmentation model that identifies regions

python

```python
# Starter structure:
import segmentation_models_pytorch as smp

model = smp.UNet(
    encoder_name="resnet18",
    encoder_weights="imagenet",
    in_channels=3,
    classes=2,  # background + foreground
)

# Training similar to classification but:
# - Loss function: DiceLoss or CrossEntropyLoss
# - Metric: IoU per pixel
# - Output: mask (same size as input) with class per pixel
```

Time: 3-4 hours

Week 2+ Reflection & Next Steps

By end of Problem 12, you've learned:

- Both classical and deep learning approaches
- Image classification, object detection, semantic segmentation
- How to evaluate each
- When each is appropriate
- Real debugging and analysis

Concepts Learned Through Doing: ✓ Convolution, CNNs, neural networks ✓ Transfer learning ✓ Bounding boxes, IoU, NMS ✓ Detection evaluation (mAP) ✓ Data augmentation and preprocessing ✓ Classical vs deep learning trade-offs ✓ Feature extraction (HOG) ✓ Semantic segmentation

---

Problem Selection Strategy

Choose problems in order for first time through:

- Problems 1-6 build foundational concepts
- Problems 7-11 apply to object detection (your domain)

- Problem 12 shows beyond detection

Can branch out:

- Interested in faces? Do face classification then face detection
- Interested in video? Add motion detection to your images
- Interested in optimization? Measure inference speed across problems

---

Learning Resources by Problem

Problem	Key Tutorial/Doc
1-2	OpenCV documentation, NumPy basics
3	scikit-learn + scikit-image tutorials
4-6	PyTorch official tutorials
7-9	YOLOv8 documentation, Roboflow annotation
10	Write from scratch, understand algorithm
11	Papers With Code object detection evaluation
12	segmentation-models-pytorch docs

Practical Tips

For Each Problem:

1. Start with given starter code
2. Run it, understand output
3. Modify to understand parameters
4. Break it (intentionally) to learn what each part does
5. Compare results with different approaches

Debugging Strategy:

- Print intermediate shapes and values
- Visualize at every step
- Compare your results to known-good results

Data Gathering:

- Don't spend hours on perfect dataset
- 20-30 images per class enough for early problems
- Use whatever images you have (your photos, public datasets)

Annotation:

- Problems 1-6: No annotation needed (use ImageNet classes)
- Problems 7-9: Invest time here, annotation is important
- Use CVAT, Roboflow, or even manual Python script

Time Per Problem: 2-5 hours each, do 1-2 per day