Feature Pyramid Network Overview

what is Feature pyramid network in computer vision

Feature Pyramid Network (FPN) is a deep learning architecture used in computer vision for multi-scale feature extraction. It is commonly employed in object detection, instance segmentation, and other vision tasks to improve the ability of convolutional neural networks (CNNs) to detect objects at different scales.

Key Concepts of FPN

1. **Pyramidal Feature Representation**:
    - Traditional CNNs extract features at multiple levels but typically use only the final layer for detection. FPN constructs a feature pyramid to leverage information from multiple scales.
2. **Bottom-Up Pathway**:
    - A standard CNN (like ResNet) extracts features at different levels using a hierarchy of convolutional layers.
    - Lower layers capture fine details (high resolution, low semantics).
    - Higher layers capture abstract patterns (low resolution, high semantics).
3. **Top-Down Pathway**:
    - FPN introduces a top-down pathway that up-samples the higher-level feature maps and combines them with the corresponding lower-level feature maps.
4. **Lateral Connections**:
    - Skip connections are used to fuse high-resolution spatial information from earlier layers with semantically rich information from deeper layers.
5. **Multi-Scale Predictions**:
    - The final output is a feature pyramid where each level contains both low-level details and high-level semantics, making detection more effective across different object sizes.

Applications of FPN

- **Object Detection**:
    - Used in **Faster R-CNN with FPN**, RetinaNet, Mask R-CNN, etc.

- Enhances small object detection.
- **Instance Segmentation**:
  - Helps segment objects accurately across various scales.
- **Image Classification**:
  - Can improve classification by capturing multi-scale information.

Advantages of FPN

✅ Better small object detection.
✅ Efficient computation with feature reuse.
✅ Improved accuracy in object detection tasks.

Would you like a code example or more details on its implementation? 🚀

implementation of FPN give me step by step with MNIST

Here's a step-by-step implementation of a **Feature Pyramid Network (FPN)** using **PyTorch** with the MNIST dataset. We'll modify a simple CNN to include an FPN structure for multi-scale feature extraction.

Step 1: Install Dependencies

Ensure you have **PyTorch** installed. If not, install it using:

```bash
pip install torch torchvision matplotlib
```

Step 2: Import Required Libraries

```python
import torch
import torch.nn as nn
```

```python
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

Step 3: Load MNIST Dataset

Since MNIST images are **28×28 grayscale**, we will **pad them to 32×32** to work with standard CNN architectures like ResNet.

```python
# Data Preprocessing
transform = transforms.Compose([
    transforms.Pad(2),  # Make images 32x32
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # Normalize dataset
])

# Load MNIST
batch_size = 64
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
shuffle=False)
```

Step 4: Define a Backbone Network

We use a **ResNet-like CNN** as our feature extractor.

```python
```

```python
class BackboneCNN(nn.Module):
    def __init__(self):
        super(BackboneCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)  # 32x32
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1) # 16x16
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1) # 8x8
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, stride=2, padding=1) # 4x4

    def forward(self, x):
        c1 = self.conv1(x)  # Feature map 1: 32x32
        c2 = self.conv2(c1)  # Feature map 2: 16x16
        c3 = self.conv3(c2)  # Feature map 3: 8x8
        c4 = self.conv4(c3)  # Feature map 4: 4x4
        return [c1, c2, c3, c4]  # Return multi-scale features
```

Step 5: Define the FPN

The FPN will use **lateral connections** and **upsampling** to merge features from different scales.

python

```python
class FPN(nn.Module):
    def __init__(self, backbone):
        super(FPN, self).__init__()
        self.backbone = backbone  # Feature extractor

        # 1x1 Conv layers for lateral connections
        self.lateral4 = nn.Conv2d(512, 256, kernel_size=1)
        self.lateral3 = nn.Conv2d(256, 256, kernel_size=1)
        self.lateral2 = nn.Conv2d(128, 256, kernel_size=1)
        self.lateral1 = nn.Conv2d(64, 256, kernel_size=1)

        # Up-sampling layers
        self.upsample = nn.Upsample(scale_factor=2, mode="nearest")

    def forward(self, x):
        c1, c2, c3, c4 = self.backbone(x)  # Extract features
```

```
        # Top-down pathway
        p4 = self.lateral4(c4)  # 4x4
        p3 = self.upsample(p4) + self.lateral3(c3)  # 8x8
        p2 = self.upsample(p3) + self.lateral2(c2)  # 16x16
        p1 = self.upsample(p2) + self.lateral1(c1)  # 32x32

        return [p1, p2, p3, p4]  # Pyramid feature maps
```

Step 6: Define Classification Head

We'll use the highest-resolution **p1** (32x32) to classify digits.

```python
class ClassificationHead(nn.Module):
    def __init__(self):
        super(ClassificationHead, self).__init__()
        self.conv = nn.Conv2d(256, 128, kernel_size=3, padding=1)
        self.pool = nn.AdaptiveAvgPool2d(1)  # Global Average Pooling
        self.fc = nn.Linear(128, 10)  # 10 classes (digits)

    def forward(self, x):
        x = self.conv(x)
        x = self.pool(x).squeeze()  # Reduce to [batch, 128]
        x = self.fc(x)  # Fully Connected Layer
        return x
```

Step 7: Train the Model

Now, let's **combine** the backbone, FPN, and classification head and train the model.

```python
# Initialize components
backbone = BackboneCNN()
```

```
fpn = FPN(backbone)
classifier = ClassificationHead()

# Move to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
fpn.to(device)
classifier.to(device)

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(list(fpn.parameters()) + list(classifier.parameters()),
lr=0.001)
```

Training Loop

```python
num_epochs = 5

for epoch in range(num_epochs):
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in trainloader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()

        # Forward pass through FPN
        pyramid_features = fpn(images)
        outputs = classifier(pyramid_features[0])  # Use highest resolution p1

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = outputs.max(1)
        correct += predicted.eq(labels).sum().item()
        total += labels.size(0)
```

```
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(trainloader):.4f}, Accuracy: {100 * correct/total:.2f}%")

print("Training Complete!")
```

Step 8: Test the Model

```python
correct = 0
total = 0

with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)

        pyramid_features = fpn(images)
        outputs = classifier(pyramid_features[0])  # Use highest resolution p1

        _, predicted = outputs.max(1)
        correct += predicted.eq(labels).sum().item()
        total += labels.size(0)

print(f"Test Accuracy: {100 * correct/total:.2f}%")
```

Step 9: Visualize Feature Maps

```python
import numpy as np

def visualize_feature_maps(features, title):
    feature_map = features[0].cpu().detach().numpy()  # Extract first image feature
maps
    fig, axes = plt.subplots(1, 4, figsize=(20, 5))
```

```python
    for i, fmap in enumerate(feature_map[:4]):  # Show first 4 feature maps
        axes[i].imshow(fmap, cmap="viridis")
        axes[i].set_title(f"Feature Map {i+1}")
    plt.suptitle(title)
    plt.show()

# Get feature maps
images, _ = next(iter(testloader))
images = images.to(device)
pyramid_features = fpn(images)

# Visualize
visualize_feature_maps(pyramid_features, "FPN Feature Maps")
```

Final Thoughts

✅ **FPN enhances multi-scale feature extraction**, improving small object detection.
✅ **We successfully trained an FPN on MNIST** using a ResNet-like CNN.
✅ **This approach can be extended** to more complex datasets like COCO or Pascal VOC.

Would you like to try a different dataset or make modifications? 🚀

display a single image

You can display a single MNIST image using `matplotlib`. Here's how:

Code to Display a Single Image

```python
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
from torchvision.datasets import MNIST

# Load the MNIST dataset
transform = transforms.Compose([
    transforms.Pad(2),  # Make images 32x32
    transforms.ToTensor()
])
dataset = MNIST(root="./data", train=True, download=True, transform=transform)
```

```
# Get a single image and its label
image, label = dataset[0]  # First image in the dataset

# Convert tensor to numpy and display
plt.imshow(image.squeeze(), cmap="gray")
plt.title(f"Label: {label}")
plt.axis("off")
plt.show()
```

This will show the first image from the MNIST dataset along with its label. 🎨📸

with 5 images and shape

You can display **5 images** from the MNIST dataset along with their shapes using **matplotlib**. Here's the code:

Code to Display 5 Images with Their Shapes

python

```python
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
from torchvision.datasets import MNIST

# Load the MNIST dataset
transform = transforms.Compose([
    transforms.Pad(2),  # Make images 32x32
    transforms.ToTensor()
])
dataset = MNIST(root="./data", train=True, download=True, transform=transform)

# Display 5 images
fig, axes = plt.subplots(1, 5, figsize=(12, 4))

for i in range(5):
    image, label = dataset[i]
    axes[i].imshow(image.squeeze(), cmap="gray")
    axes[i].set_title(f"Label: {label}\nShape: {image.shape}")
```

```
    axes[i].axis("off")

plt.show()
```

Output Details

- Displays **5 MNIST images** from the dataset.
- Shows **labels** and **shapes** (should be **(1, 32, 32)** due to padding).
- Uses **matplotlib** for visualization.

Let me know if you need modifications! 🚀📸