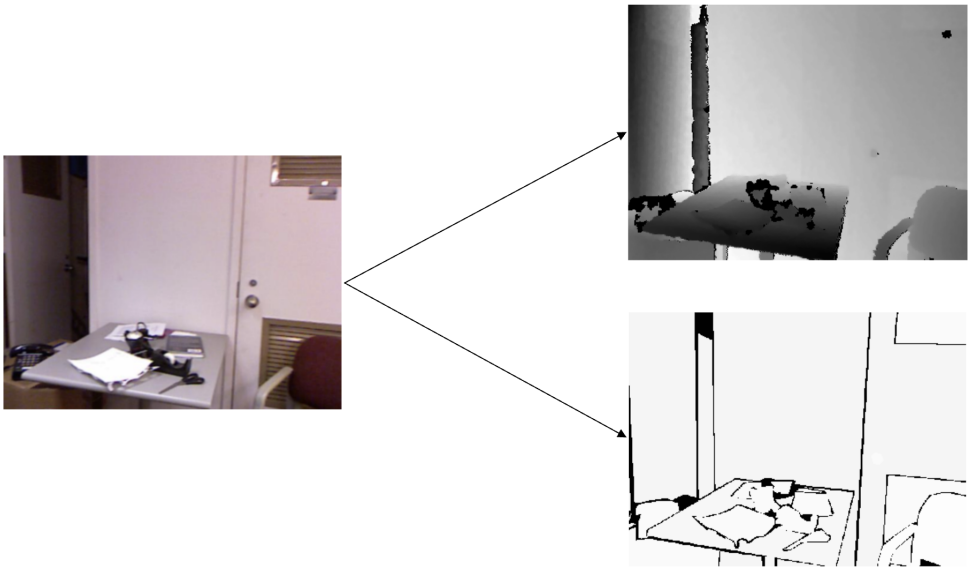
AI VIET NAM – AIO COURSE 2023

Project: Multi-Task Learning

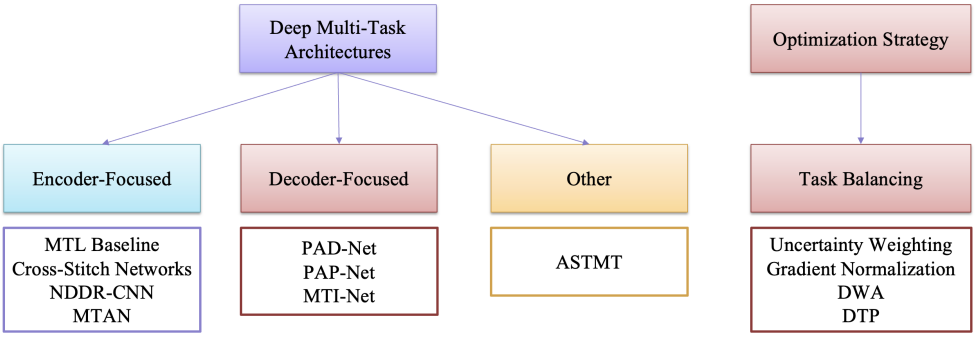
Quoc-Thai Nguyen và Quang-Vinh Dinh

Ngày 30 tháng 4 năm 2024

Phần I. Giới thiệu



Hình 1: Ví dụ về mô hình học đa tác vụ cho bài toán image semantic segmentation và depth-image prediction



Hình 2: Các phương pháp huấn luyện mô hình học đa tác vụ.

Mô hình học đa tác vụ (Multi-Task Learning) là phương pháp huấn luyện cho một mô hình nhưng có thể giải quyết cho nhiều bài toán khác nhau. Ví dụ, chúng ta huấn luyện một mô hình với đầu vào là một hình ảnh và đầu ra giải quyết cho hai bài toán khác nhau như: semantic segmentation và depth-image prediction được mô tả như Hình 1.

Các phương pháp huấn luyện các mô hình MTL có thể được chia thành 2 nhóm:

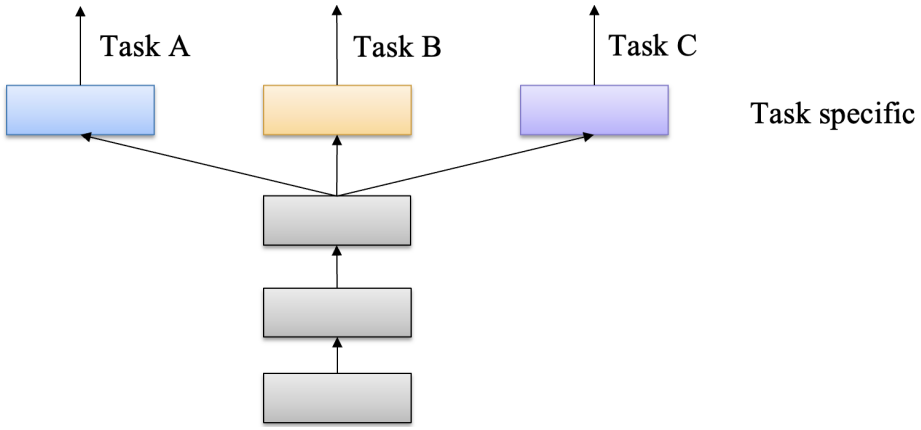
1. Deep Multi-Task Architectures. Bao gồm các phương pháp tập trung vào xây dựng các mô hình chung để giải quyết các bài toán khác nhau. Trong đó gồm 2 thành phần chính: thành phần thứ nhất bao gồm các layer chung hoặc chia sẻ trọng số để học các đặc trưng thường gọi là shared-encoder; thành phần thứ hai bao gồm các layer riêng để học các đặc trưng để tối ưu cho từng bài toán riêng thường được gọi là task-specific layer.

2. Optimization Strategy. Với mỗi bài toán sẽ có hàm loss đánh giá khác nhau. Trong phần này sẽ tập trung xây dựng các thuật toán tối ưu trong quá trình huấn luyện và cập nhật trọng số. Điển hình trong đó là các phương pháp cân bằng trọng số khi tổng hợp hàm loss, như gradient normalization, uncertainy weighting,...

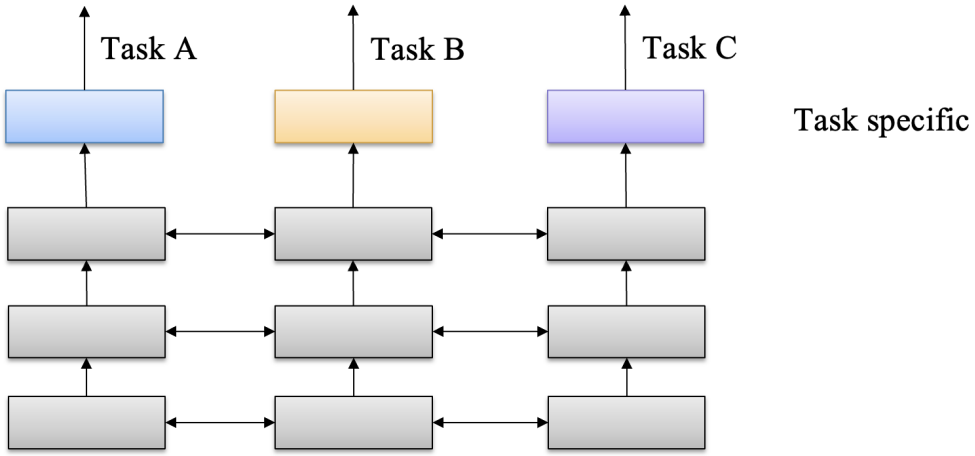
Trong phần này để bước đầu hiểu rõ về các phương pháp huấn luyện mô hình MTL, chúng ta sẽ triển khai mô hình MTL cơ bản tập trung vào tinh chỉnh phần encoder là hard parameter sharing model.

Phần II. Multi-Task Learning for Computer Vision

Hai phương pháp chính bao gồm: hard parameter sharing được mô tả trong hình 3 và soft parameter sharing được mô tả trong hình 4.



Hình 3: Hard parameter sharing.



Hình 4: Soft parameter sharing.

Hard parameter sharing bao gồm các layer dùng chung cho tất cả các task, shared layers. Sau các shared layers bao gồm các task-specific layers cho các task khác nhau.

Soft parameter sharing bao gồm các layer dùng riêng cho các task khác nhau. Tuy nhiên, thay vì ở các layer đầu tiên của các mô hình sẽ hoạt động riêng lẽ cho các task thì sẽ có các kết nốt với đến task khác. Các kết nối có thể là tuyến tính hoặc sử dụng các hàm kích hoạt để kết nối các layer khác nhau.

Trong phần này chúng ta sẽ huấn luyện hard parameter sharing model trên bộ dữ liệu NYUD-V2 cho hai bài toán semantic segmentation và depth-image prediction. Bộ dữ liệu sau khi thực hiện các bước tiền xử lý như chuyển sang tensor,... có thể được tải về tại đây.

1.1. Dataset

```
import os
import torch
import fnmatch
import numpy as np
from torch.utils.data import DataLoader

class NYUv2(torch.utils.data.dataset.Dataset):
    def __init__(self, root, train=True):
        self.train = train
        self.root = os.path.expanduser(root)

        # read the data file
        if train:
            self.data_path = root + '/train'
        else:
            self.data_path = root + '/val'

        # calculate data length
        self.data_len = len(fnmatch.filter(os.listdir(self.data_path + '/image'), '*.npy'))

    def __getitem__(self, index):
        # load data from the pre-processed npy files
        image = torch.from_numpy(np.moveaxis(np.load(self.data_path + '/image/{:d}.npy'.format(index)), -1, 0))
        semantic = torch.from_numpy(np.load(self.data_path + '/label/{:d}.npy'.format(index)))
        depth = torch.from_numpy(np.moveaxis(np.load(self.data_path + '/depth/{:d}.npy'.format(index)), -1, 0))

        return {
            'image': image.float(),
            'semantic': semantic.float(),
            'depth': depth.float()
        }

    def __len__(self):
        return self.data_len

data_path = './data/NYUDv2'
train_ds = NYUv2(root=data_path, train=True)
val_ds = NYUv2(root=data_path, train=False)

batch_size = 4
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=batch_size, shuffle=False)
```

1.2. Model

Trong phần này chúng ta xây dựng mô hình hard parameter sharing được mô tả như hình 3. Trong đó, các layer sẽ bao gồm các lớp convolution, batch normalization, activation,...

```
import torch.nn as nn
import torch.nn.functional as F

class HardParameterSharingModel(nn.Module):
    def __init__(self):
        super(HardParameterSharingModel, self).__init__()
        # initialise network parameters
```

```
        filter = [64, 128, 256, 512, 512]

        self.class_nb = 13

        # define encoder decoder layers
        self.encoder_block = nn.ModuleList([self.conv_layer([3, filter[0]])])
        self.decoder_block = nn.ModuleList([self.conv_layer([filter[0], filter[0]])])
        for i in range(4):
            self.encoder_block.append(self.conv_layer([filter[i], filter[i + 1]]))
            self.decoder_block.append(self.conv_layer([filter[i + 1], filter[i]]))

        # define convolution layer
        self.conv_block_enc = nn.ModuleList([self.conv_layer([filter[0], filter[0]])])
        self.conv_block_dec = nn.ModuleList([self.conv_layer([filter[0], filter[0]])])
        for i in range(4):
            if i == 0:
                self.conv_block_enc.append(self.conv_layer([filter[i + 1], filter[i + 1]]))
                self.conv_block_dec.append(self.conv_layer([filter[i], filter[i]]))
            else:
                self.conv_block_enc.append(nn.Sequential(self.conv_layer([filter[i + 1], filter[i + 1]]),
                                                         self.conv_layer([filter[i + 1], filter[i + 1]])))
                self.conv_block_dec.append(nn.Sequential(self.conv_layer([filter[i], filter[i]]),
                                                         self.conv_layer([filter[i], filter[i]])))

        # define task specific layers
        self.pred_task1 = nn.Sequential(nn.Conv2d(in_channels=filter[0], out_channels=filter[0], kernel_size=3, padding=1),
                                        nn.Conv2d(in_channels=filter[0], out_channels=self.class_nb, kernel_size=1, padding=0))
        self.pred_task2 = nn.Sequential(nn.Conv2d(in_channels=filter[0], out_channels=filter[0], kernel_size=3, padding=1),
                                        nn.Conv2d(in_channels=filter[0], out_channels=1, kernel_size=1, padding=0))

        # define pooling and unpooling functions
        self.down_sampling = nn.MaxPool2d(kernel_size=2, stride=2, return_indices=True)
        self.up_sampling = nn.MaxUnpool2d(kernel_size=2, stride=2)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.xavier_normal_(m.weight)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight)
                nn.init.constant_(m.bias, 0)

    # define convolutional block
    def conv_layer(self, channel):
        conv_block = nn.Sequential(
                nn.Conv2d(in_channels=channel[0], out_channels=channel[1], kernel_size=3, padding=1),
```

```
                nn.BatchNorm2d(num_features=channel[1]),
                nn.ReLU(inplace=True)
            )
        return conv_block

    def forward(self, x):
        g_encoder, g_decoder, g_maxpool, g_upsampl, indices = ([0] * 5 for _ in range(5))
        for i in range(5):
            g_encoder[i], g_decoder[-i - 1] = ([0] * 2 for _ in range(2))

        # global shared encoder-decoder network
        for i in range(5):
            if i == 0:
                g_encoder[i][0] = self.encoder_block[i](x)
                g_encoder[i][1] = self.conv_block_enc[i](g_encoder[i][0])
                g_maxpool[i], indices[i] = self.down_sampling(g_encoder[i][1])
            else:
                g_encoder[i][0] = self.encoder_block[i](g_maxpool[i - 1])
                g_encoder[i][1] = self.conv_block_enc[i](g_encoder[i][0])
                g_maxpool[i], indices[i] = self.down_sampling(g_encoder[i][1])

        for i in range(5):
            if i == 0:
                g_upsampl[i] = self.up_sampling(g_maxpool[-1], indices[-i - 1])
                g_decoder[i][0] = self.decoder_block[-i - 1](g_upsampl[i])
                g_decoder[i][1] = self.conv_block_dec[-i - 1](g_decoder[i][0])
            else:
                g_upsampl[i] = self.up_sampling(g_decoder[i - 1][-1], indices[-i - 1])
                g_decoder[i][0] = self.decoder_block[-i - 1](g_upsampl[i])
                g_decoder[i][1] = self.conv_block_dec[-i - 1](g_decoder[i][0])

        # define task prediction layers
        t1_pred = F.log_softmax(self.pred_task1(g_decoder[i][1]), dim=1)
        t2_pred = self.pred_task2(g_decoder[i][1])

        return {
            'semantic': t1_pred,
            'depth': t2_pred
        }
```

1.3. Loss

Trong phần này chúng ta xây dựng hai hàm loss khác nhau cho bài toán semantic segmentation là pixel-wise cross entropy và depth-image prediction là L1.

```
def compute_loss(x_pred, x_output, task_type):
    device = x_pred.device

    # binary mark to mask out undefined pixel space
    binary_mask = (torch.sum(x_output, dim=1) != 0).float().unsqueeze(1).to(device)

    if task_type == 'semantic':
        # semantic loss: depth-wise cross entropy
        loss = F.nll_loss(x_pred, x_output, ignore_index=-1)

    if task_type == 'depth':
        # depth loss: l1 norm
        loss = torch.sum(torch.abs(x_pred - x_output) * binary_mask) / torch.nonzero(binary_mask, as_tuple=False).size(0)

    return loss
```

1.4. Training

```python
from tqdm import tqdm

def train_epoch(train_loader, model, device, optimizer):
    # iteration for all batches
    model.train()
    losses = {'semantic': [], 'depth': [], 'total': []}
    for i, batch in tqdm(enumerate(train_loader)):
        images = batch['image'].to(device)
        semantic = batch['semantic'].long().to(device)
        depth = batch['depth'].to(device)

        output = model(images)

        optimizer.zero_grad()
        train_loss = {
            'semantic': compute_loss(output['semantic'], semantic, 'semantic'),
            'depth': compute_loss(output['depth'], depth, 'depth')
        }

        loss = train_loss['semantic'] + train_loss['depth']

        loss.backward()
        optimizer.step()

        losses['semantic'].append(train_loss['semantic'].item())
        losses['depth'].append(train_loss['depth'].item())
        losses['total'].append(loss.item())

    avg_losses = {task: sum(task_loss)/len(task_loss) for task, task_loss in losses.
    items()}

    return avg_losses


def evaluation_epoch(val_loader, model, device):
    # iteration for all batches
    model.eval()
    losses = {'semantic': [], 'depth': [], 'total':[]}
    with torch.no_grad():
        for i, batch in tqdm(enumerate(val_loader)):
            images = batch['image'].to(device)
            semantic = batch['semantic'].long().to(device)
            depth = batch['depth'].to(device)

            output = model(images)

            train_loss = {
                'semantic': compute_loss(output['semantic'], semantic, 'semantic'),
                'depth': compute_loss(output['depth'], depth, 'depth')
            }

            loss = train_loss['semantic'] + train_loss['depth']

            losses['semantic'].append(train_loss['semantic'].item())
            losses['depth'].append(train_loss['depth'].item())
            losses['total'].append(loss.item())

    avg_losses = {task: sum(task_loss)/len(task_loss) for task, task_loss in losses.
```

```
    items()}
    return avg_losses

def train(train_loader, val_loader, model, device, optimizer, epochs):
    best_loss = 100.
    for epoch in range(epochs):
        train_loss = train_epoch(train_loader, model, device, optimizer)
        val_loss = train_epoch(train_loader, model, device, optimizer)
        scheduler.step()
        if val_loss['total'] < best_loss:
            best_loss = val_loss['total']
            torch.save(model.state_dict(), './model/
    soft_parameter_sharing_model_weights.pth')
            print(f"Model save: ./model/soft_parameter_sharing_model_weights.pth")
        print('Epoch: {:04d} | Train: Semantic Loss {:.4f} - Depth Loss {:.4f} - Total
     Loss {:.4f} ||'
            'Eval: Semantic Loss {:.4f} - Depth Loss {:.4f} - Total Loss {:.4f} '
            .format(epoch+1, train_loss['semantic'], train_loss['depth'], train_loss['
    total'],
                    val_loss['semantic'], val_loss['depth'], val_loss['total']))
    returnn model

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = HardParameterSharingModel()
model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)

epochs = 10
model = train(train_loader, val_loader, model, device, optimizer, epochs)
```

1.5. Inference

```
model_path = './model/hard_parameter_sharing_model_weights.pth'
model = HardParameterSharingModel()
model.load_state_dict(torch.load(model_path))
model.eval()
model.to(device)

test_sample = next(iter(val_ds))
test_sample = {task: test_sample[task].unsqueeze(0).to(device) for task in test_sample
    .keys()}

with torch.no_grad():
    output = model(test_sample['image'])
# output: output['depth'], output['sematic']
```

Phần 3. Câu hỏi trắc nghiệm

Câu hỏi 1 Học đa tác vụ (Multi-Task Learning) là gì?
a) Học cách thực hiện nhiều tác vụ khác nhau một cách độc lập
b) Học cách thực hiện nhiều tác vụ cùng một lúc
c) Học cách thực hiện một tác vụ duy nhất
d) Học cách thực hiện nhiều tác vụ theo thứ tự

Câu hỏi 2 Lợi ích chính của học đa tác vụ là gì?
a) Tăng độ chính xác của mỗi tác vụ
b) Tăng khả năng chuyển giao kiến thức giữa các tác vụ
c) Cả 2 đáp án đều đúng
d) Cả 2 đáp án đều sai

Câu hỏi 3 Phương pháp nào sau đây không phải là phương pháp học đa tác vụ?
a) Chia sẻ tham số
b) Chia sẻ biểu diễn
c) Chia sẻ dữ liệu
d) Học tuần tự

Câu hỏi 4 Mô hình MTL nào sau đây tập trung vào tinh chỉnh khối encoder?
a) PAD-Net
b) PAP-Net
c) MTI-Net
d) Cross-Stitch Networks

Câu hỏi 5 Mô hình MTL nào sau đây tập trung vào tinh chỉnh khối decoder?
a) PAD-Net
b) MTAN
c) NDDR-CNN
d) Cross-Stitch Networks

Câu hỏi 6 Hàm loss để huấn luyện cho bài toán semantic segmentation là?
a) Pixel-wise Cross Entropy
b) Huber Loss
c) Recontruction Loss
d) Contrastive Loss

Câu hỏi 7 Hàm loss để huấn luyện cho bài toán depth-image prediction là?
a) Pixel-wise Cross Entropy
b) Huber Loss

c) L1
d) Contrastive Loss

Câu hỏi 8 Bộ dữ liệu sử dụng cho phần thực nghiệm là?
a) NYUD-V2
b) CIFAR10
c) CIFAR100
d) MNIST

Câu hỏi 9 Bộ dữ liệu NYUD-V2 bao nhiêu sample?
a) 499
b) 1119
c) 1449
d) 1999

Câu hỏi 10 Độ đo để đánh giá cho bài toán semantic segmentation là?
a) IoU
b) F-Score
c) Recall
d) Precision

- Hết -