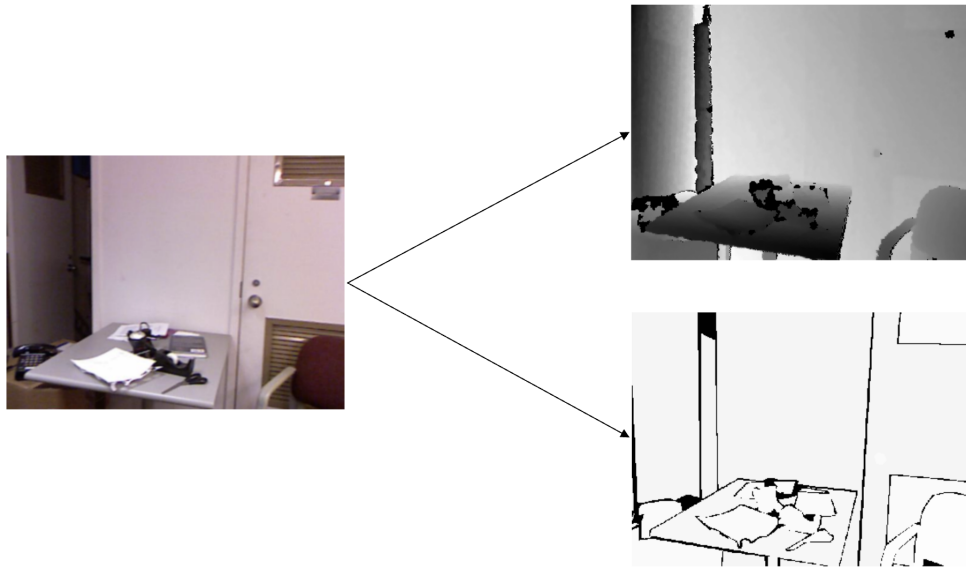


# Exercise: Multi-Task Learning

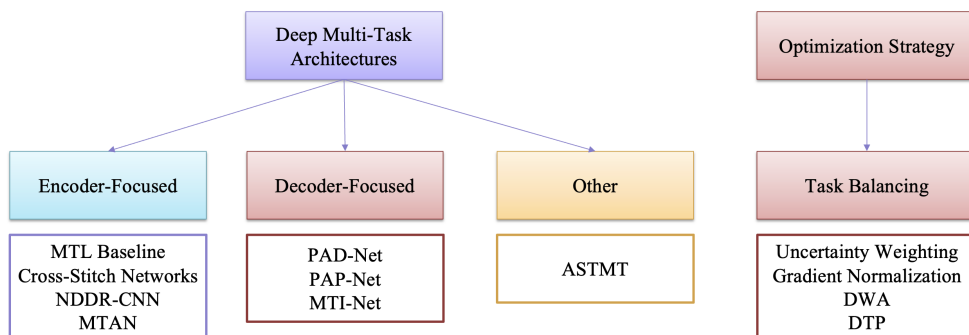
Quoc-Thai Nguyen và Quang-Vinh Dinh

Ngày 29 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.

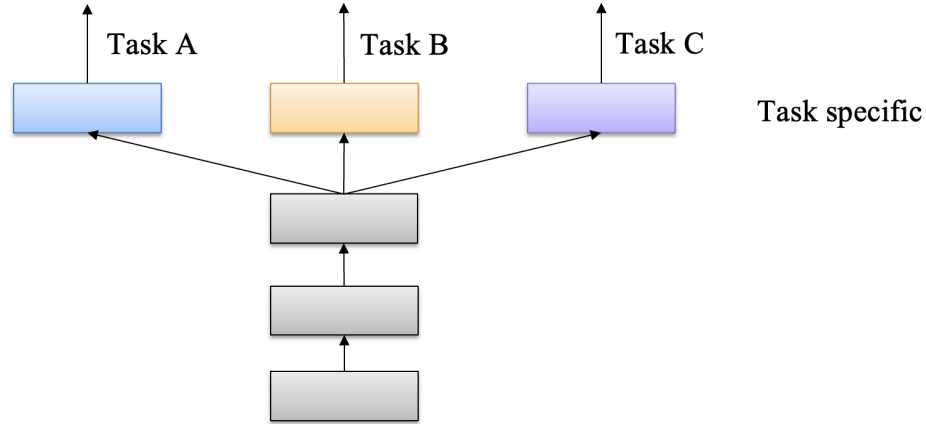
Các phương pháp huấn luyện các mô hình MTL có thể được chia thành 2 nhóm:

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, uncertainty weighting,...

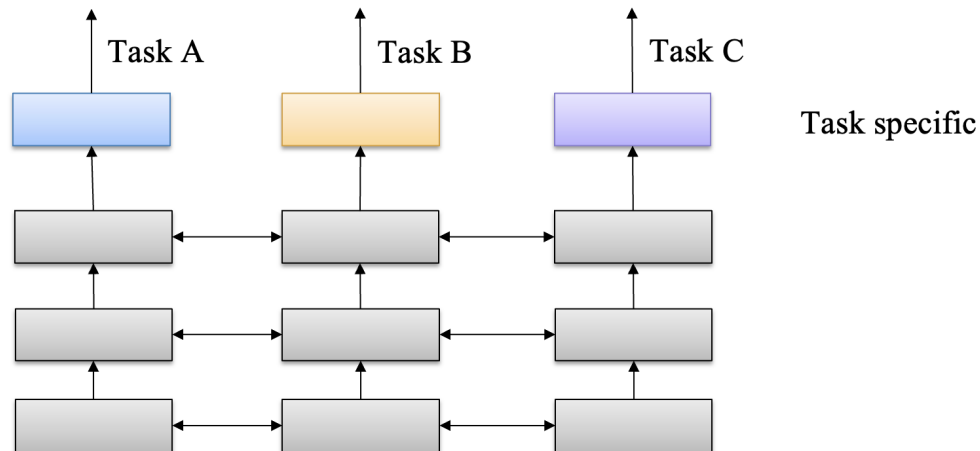
Trong phần này để bước đầu hiểu rõ về các phương pháp huấn luyện mô hình MTL, chúng ta sẽ triển khai mô hình MTL cơ bản tập trung vào tính chỉnh phần encoder là hard parameter sharing model.

## 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 lẻ cho các task thì sẽ có các kết nối 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

```

1 import os
2 import torch
3 import fnmatch
4 import numpy as np
5 from torch.utils.data import DataLoader
6
7 class NYUv2(torch.utils.data.dataset.Dataset):
8     def __init__(self, root, train=True):
9         self.train = train
10        self.root = os.path.expanduser(root)
11
12        # read the data file
13        if train:
14            self.data_path = root + '/train'
15        else:
16            self.data_path = root + '/val'
17
18        # calculate data length
19        self.data_len = len(fnmatch.filter(os.listdir(self.data_path + '/image'), '*.
    npy'))
20
21    def __getitem__(self, index):
22        # load data from the pre-processed npy files
23        image = torch.from_numpy(np.moveaxis(np.load(self.data_path + '/image/{:d}.np
    y'.format(index)), -1, 0))
24        semantic = torch.from_numpy(np.load(self.data_path + '/label/{:d}.np
    y'.format(index)))
25        depth = torch.from_numpy(np.moveaxis(np.load(self.data_path + '/depth/{:d}.np
    y'.format(index)), -1, 0))
26
27        return {
28            'image': image.float(),
29            'semantic': semantic.float(),
30            'depth': depth.float()
31        }
32
33    def __len__(self):
34        return self.data_len
35
36 data_path = './data/NYUDv2'
37 train_ds = NYUv2(root=data_path, train=True)
38 val_ds = NYUv2(root=data_path, train=False)
39
40 batch_size = 4
41 train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
42 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,...

```

1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 class HardParameterSharingModel(nn.Module):
5     def __init__(self):
6         super(HardParameterSharingModel, self).__init__()
7         # initialise network parameters

```

```

8         filter = [64, 128, 256, 512, 512]
9
10        self.class_nb = 13
11
12        # define encoder decoder layers
13        self.encoder_block = nn.ModuleList([self.conv_layer([3, filter[0]])])
14        self.decoder_block = nn.ModuleList([self.conv_layer([filter[0], filter[0]])])
15        for i in range(4):
16            self.encoder_block.append(self.conv_layer([filter[i], filter[i + 1]]))
17            self.decoder_block.append(self.conv_layer([filter[i + 1], filter[i]]))
18
19        # define convolution layer
20        self.conv_block_enc = nn.ModuleList([self.conv_layer([filter[0], filter[0]])])
21        self.conv_block_dec = nn.ModuleList([self.conv_layer([filter[0], filter[0]])])
22        for i in range(4):
23            if i == 0:
24                self.conv_block_enc.append(self.conv_layer([filter[i + 1], filter[i +
1]]))
25                self.conv_block_dec.append(self.conv_layer([filter[i], filter[i]]))
26            else:
27                self.conv_block_enc.append(nn.Sequential(self.conv_layer([filter[i +
1], filter[i + 1]]),
28                                                         self.conv_layer([filter[i +
1], filter[i + 1]])))
29                self.conv_block_dec.append(nn.Sequential(self.conv_layer([filter[i],
filter[i]]),
30                                                         self.conv_layer([filter[i],
filter[i]])))
31
32        # define task specific layers
33        self.pred_task1 = nn.Sequential(nn.Conv2d(in_channels=filter[0], out_channels=
filter[0], kernel_size=3, padding=1),
34                                       nn.Conv2d(in_channels=filter[0], out_channels=
self.class_nb, kernel_size=1, padding=0))
35        self.pred_task2 = nn.Sequential(nn.Conv2d(in_channels=filter[0], out_channels=
filter[0], kernel_size=3, padding=1),
36                                       nn.Conv2d(in_channels=filter[0], out_channels
=1, kernel_size=1, padding=0))
37
38        # define pooling and unpooling functions
39        self.down_sampling = nn.MaxPool2d(kernel_size=2, stride=2, return_indices=True
)
40        self.up_sampling = nn.MaxUnpool2d(kernel_size=2, stride=2)
41
42        for m in self.modules():
43            if isinstance(m, nn.Conv2d):
44                nn.init.xavier_normal_(m.weight)
45                nn.init.constant_(m.bias, 0)
46            elif isinstance(m, nn.BatchNorm2d):
47                nn.init.constant_(m.weight, 1)
48                nn.init.constant_(m.bias, 0)
49            elif isinstance(m, nn.Linear):
50                nn.init.xavier_normal_(m.weight)
51                nn.init.constant_(m.bias, 0)
52
53        # define convolutional block
54        def conv_layer(self, channel):
55            conv_block = nn.Sequential(
56                nn.Conv2d(in_channels=channel[0], out_channels=channel[1], kernel_size
=3, padding=1),

```

```

57         nn.BatchNorm2d(num_features=channel[1]),
58         nn.ReLU(inplace=True)
59     )
60     return conv_block
61
62     def forward(self, x):
63         g_encoder, g_decoder, g_maxpool, g_upsampl, indices = ([0] * 5 for _ in range
64 (5))
65         for i in range(5):
66             g_encoder[i], g_decoder[-i - 1] = ([0] * 2 for _ in range(2))
67
68         # global shared encoder-decoder network
69         for i in range(5):
70             if i == 0:
71                 g_encoder[i][0] = self.encoder_block[i](x)
72                 g_encoder[i][1] = self.conv_block_enc[i](g_encoder[i][0])
73                 g_maxpool[i], indices[i] = self.down_sampling(g_encoder[i][1])
74             else:
75                 g_encoder[i][0] = self.encoder_block[i](g_maxpool[i - 1])
76                 g_encoder[i][1] = self.conv_block_enc[i](g_encoder[i][0])
77                 g_maxpool[i], indices[i] = self.down_sampling(g_encoder[i][1])
78
79         for i in range(5):
80             if i == 0:
81                 g_upsampl[i] = self.up_sampling(g_maxpool[-1], indices[-i - 1])
82                 g_decoder[i][0] = self.decoder_block[-i - 1](g_upsampl[i])
83                 g_decoder[i][1] = self.conv_block_dec[-i - 1](g_decoder[i][0])
84             else:
85                 g_upsampl[i] = self.up_sampling(g_decoder[i - 1][-1], indices[-i - 1])
86                 g_decoder[i][0] = self.decoder_block[-i - 1](g_upsampl[i])
87                 g_decoder[i][1] = self.conv_block_dec[-i - 1](g_decoder[i][0])
88
89         # define task prediction layers
90         t1_pred = F.log_softmax(self.pred_task1(g_decoder[i][1]), dim=1)
91         t2_pred = self.pred_task2(g_decoder[i][1])
92
93         return {
94             'semantic': t1_pred,
95             'depth': t2_pred
96         }

```

### 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.

```

1 def compute_loss(x_pred, x_output, task_type):
2     device = x_pred.device
3
4     # binary mark to mask out undefined pixel space
5     binary_mask = (torch.sum(x_output, dim=1) != 0).float().unsqueeze(1).to(device)
6
7     if task_type == 'semantic':
8         # semantic loss: depth-wise cross entropy
9         loss = F.nll_loss(x_pred, x_output, ignore_index=-1)
10
11     if task_type == 'depth':
12         # depth loss: l1 norm
13         loss = torch.sum(torch.abs(x_pred - x_output) * binary_mask) / torch.nonzero(
14             binary_mask, as_tuple=False).size(0)
15
16     return loss

```

## 1.4. Training

```

1 from tqdm import tqdm
2
3 def train_epoch(train_loader, model, device, optimizer):
4     # iteration for all batches
5     model.train()
6     losses = {'semantic': [], 'depth': [], 'total': []}
7     for i, batch in tqdm(enumerate(train_loader)):
8         images = batch['image'].to(device)
9         semantic = batch['semantic'].long().to(device)
10        depth = batch['depth'].to(device)
11
12        output = model(images)
13
14        optimizer.zero_grad()
15        train_loss = {
16            'semantic': compute_loss(output['semantic'], semantic, 'semantic'),
17            'depth': compute_loss(output['depth'], depth, 'depth')
18        }
19
20        loss = train_loss['semantic'] + train_loss['depth']
21
22        loss.backward()
23        optimizer.step()
24
25        losses['semantic'].append(train_loss['semantic'].item())
26        losses['depth'].append(train_loss['depth'].item())
27        losses['total'].append(loss.item())
28
29    avg_losses = {task: sum(task_loss)/len(task_loss) for task, task_loss in losses.
30                  items()}
31    return avg_losses
32
33
34 def evaluation_epoch(val_loader, model, device):
35     # iteration for all batches
36     model.eval()
37     losses = {'semantic': [], 'depth': [], 'total': []}
38     with torch.no_grad():
39         for i, batch in tqdm(enumerate(val_loader)):
40             images = batch['image'].to(device)
41             semantic = batch['semantic'].long().to(device)
42             depth = batch['depth'].to(device)
43
44             output = model(images)
45
46             train_loss = {
47                 'semantic': compute_loss(output['semantic'], semantic, 'semantic'),
48                 'depth': compute_loss(output['depth'], depth, 'depth')
49             }
50
51             loss = train_loss['semantic'] + train_loss['depth']
52
53             losses['semantic'].append(train_loss['semantic'].item())
54             losses['depth'].append(train_loss['depth'].item())
55             losses['total'].append(loss.item())
56
57    avg_losses = {task: sum(task_loss)/len(task_loss) for task, task_loss in losses.

```

```

    items()
58     return avg_losses
59
60 def train(train_loader, val_loader, model, device, optimizer, epochs):
61     best_loss = 100.
62     for epoch in range(epochs):
63         train_loss = train_epoch(train_loader, model, device, optimizer)
64         val_loss = train_epoch(val_loader, model, device, optimizer)
65         scheduler.step()
66         if val_loss['total'] < best_loss:
67             best_loss = val_loss['total']
68             torch.save(model.state_dict(), './model/
soft_parameter_sharing_model_weights.pth')
69             print(f"Model save: ./model/soft_parameter_sharing_model_weights.pth")
70             print('Epoch: {:04d} | Train: Semantic Loss {:.4f} - Depth Loss {:.4f} - Total
Loss {:.4f} ||'
71                   'Eval: Semantic Loss {:.4f} - Depth Loss {:.4f} - Total Loss {:.4f} '
72                   '.format(epoch+1, train_loss['semantic'], train_loss['depth'], train_loss['
total'],
73                             val_loss['semantic'], val_loss['depth'], val_loss['total']))
74     return model
75
76 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
77 model = HardParameterSharingModel()
78 model.to(device)
79 optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
80 scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)
81
82 epochs = 10
83 model = train(train_loader, val_loader, model, device, optimizer, epochs)

```

## 1.5. Inference

```

1 model_path = './model/hard_parameter_sharing_model_weights.pth'
2 model = HardParameterSharingModel()
3 model.load_state_dict(torch.load(model_path))
4 model.eval()
5 model.to(device)
6
7 test_sample = next(iter(val_ds))
8 test_sample = {task: test_sample[task].unsqueeze(0).to(device) for task in test_sample
.keys()}
9
10 with torch.no_grad():
11     output = model(test_sample['image'])
12 # output: output['depth'], output['semantic']

```



## 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) MTL-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) Reconstruction 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 -**