Computer Vision 2: Exercise Sheet 4

Summary:

1. Using validation data and shuffling training data
2. Saving and loading parameters
3. Using pre-trained networks

1 Validation data and data shuffling

Validation data is used to monitor the generalization capability of a neural network. While training the network (using training data), the performance on validation data is periodically checked. If the validation loss starts to increase, this may indicate that the network has been overfit, for example.

Shuffling training data breaks correlations between batches and helps avoid overfitting. As validation data is only used to evaluate a model, it is not necessary to shuffle it.

No backpropagation is needed for the validation data. Validation data therefore requires less memory as no gradient information has to be stored. Therefore we can also use a larger batch size.

Load the training and validation sets of the MNIST dataset available in Moodle and create data loaders for the training and validation data as follows.

```
train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle=True)
valid_ds = TensorDataset(x_valid, y_valid)
valid_dl = DataLoader(valid_ds, batch_size=bs * 2)
```

Use the same CNN architecture, loss function, and optimizer as in the previous exercises. Create a training loop that includes a validation step after each epoch. The following code provides a rough outline that you can adapt, see also the remarks after the code block.

```
for epoch in range(epochs):
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        opt.step()
        opt.zero_grad()

    model.eval()
    with torch.no_grad():
        valid_loss = sum(loss_func(model(xb), yb) for xb, yb in valid_dl)

    print(epoch, valid_loss / len(valid_dl))
```

- `model` refers to your CNN module instance.

- `model.train()` and `model.eval()` are methods that should always be called before the training or validation/test phases, respectively. The short explanation for why this is good to do is that some layers such as batch normalization or dropout may operate differently when training versus evaluating.

- `with torch.no_grad()` creates a block where for all code inside the block, gradient tracking and therefore automatic differentiation for all tensors is disabled. This is equivalent to calling `.required_grad_(False)` on all the tensors manually.

Now, the printed validation loss may be monitored. For example, if the validation loss starts to increase, most likely the network is overfitting.

2 Saving and loading parameters

This section is based on the tutorial `https://pytorch.org/tutorials/beginner/saving_loading_models.html#what-is-a-state-dict`. A summary is given below, but you can read the complete tutorial for more in-depth knowledge.

Learnable parameters (i.e., weights and biases) of an `torch.nn.Module` model are contained in the model's parameters accessed with `model.parameters()`. A `state_dict` is a Python dictionary object that maps each layer to its parameter tensor. Only layers with learnable parameters (convolutional layers, linear layers, etc.) have entries in the model's `state_dict`. Optimizer objects also have a `state_dict`, which contains information about the optimizer's state and hyperparameters.

To see what is stored in a `state_dict`, run the following code. Once again, `model` refers to your CNN module instance, and `optimizer` to an instance of the optimizer object, such as `torch.optim.SGD`.

```
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])
```

2.1 Basic saving and loading

With `state_dict`, saving and loading model parameters is easy. A common PyTorch convention is to save models using either a .pt or .pth file extension.

Saving:

```
torch.save(model.state_dict(), PATH)
```

Here `PATH` is a string giving the path and file name where to save the dictionary, for example `PATH = 'my_model_epoch00.pt'`.

Loading (assumes you have an instance of your model class constructed):

```
model.load_state_dict(torch.load(PATH))
model.eval()
```

Notice that the `load_state_dict()` function takes a dictionary object, NOT a path to a saved object.

2.2 Saving and loading a checkpoint to resume training from

If you want to save a so-called checkpoint during training, you need to store the optimizer state and loss as well. These can be saved in a dictionary format as well, for example:

```
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ... # here other properties if you need
            }, PATH)
```

Then the checkpoint may be loaded and training can be resumed, or evaluation may be started. For example, using the same format as saving above, and assuming you have created a model and optimizer instance:

```
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

model.eval()
# - or -
model.train()
```

Using checkpoints. Work with your code from Section 1, and complete the following tasks.

1. Modify your training loop so it saves a checkpoint after each epoch.
2. Run your training for a couple of epochs, and store the checkpoints.
3. Modify your code so that instead of starting from scratch, it loads a given checkpoint and resumes training from there.

3 Pre-trained networks and transfer learning

Transfer learning is the process of applying a neural network learned for one task in a source domain as a starting point to solve another task in a target domain. It is a powerful tool, as parameters of networks trained on large datasets can be used to bootstrap learning on other problems. At its simplest, transfer learning just means initializing the parameters of your network by loading them from a pre-trained network with the `state_dict` mechanism introduced earlier. A concise summary of some aspects of transfer learning may be found in the tutorial `https://cs231n.github.io/transfer-learning/`.

The parameters are then *fine-tuned* by using data from the target domain. Sometimes, all parameters are fine-tuned, and sometimes a part of the parameters are kept fixed to their pre-trained values. For example, it is common to only fine-tune or re-train one or a few of the last layers of a CNN. Generally, the more training data is available in the target domain, the more parameters can be fine-tuned successfully.

3.1 Model zoo

PyTorch includes a “model zoo” with many pre-trained models available. For example:

```
import torchvision.models as models
vgg16 = models.vgg16(pretrained=True)
```

This returns the VGG 16-layer model (configuration “D”) from the paper Very Deep Convolutional Networks For Large-Scale Image Recognition. A comprehensive list of all available models is found at `https://pytorch.org/docs/stable/torchvision/models.html`.

Important: All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3-by-H-by-W), where H and W are expected to be at least 224. The images have to be loaded in to a range of $[0, 1]$ and then normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`. You can use the following transform to normalize:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

3.2 Using a pre-trained model as a feature extractor

A typical use case is to apply a pre-trained CNN as a fixed feature extractor. All convolutional layers are fixed, and we train one or several newly-initialized fully connected layers. This is conceptually illustrated in Figure 1.

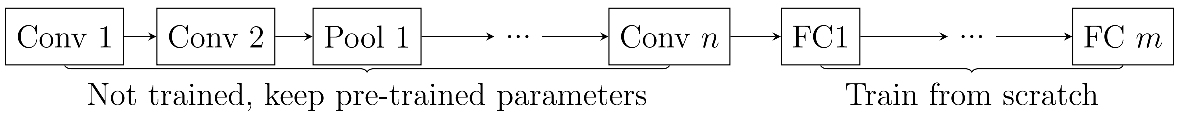

Figure 1: Using a CNN as a fixed feature extractor and train a classifier on top.

Setting this up in PyTorch for one fully connected layer and with `num_out_classes` output classes works as follows for the VGG16 example from above.

```
for param in vgg16.parameters():
    param.requires_grad = False

# newly constructed modules have requires_grad=True by default
num_ftrs = vgg16.fc.in_features
vgg16.fc = nn.Linear(num_ftrs, num_out_classes)
```

Note that `vgg16.fc` is a submodule that consists of the fully connected layers of the network. It is replaced by a newly created linear layer.

The newly added fully connected (linear) layer can be trained as usual. No gradients are calculated for the convolutional layers.