Practical 3 - Transfer Learning

Team number: 21
Team members: Emil Alizada, Sofiene Boutaj, Felix Hsieh, Daniil Morozov

The problem statement:

Generating high quality labels for medical datasets is an expensive and time consuming task, especially on tasks such as segmentation where expert radiologists are required. Often we are faced with datasets that have scarce labels or none at all and must rely on self-supervised pre-training methods to increase our performance. We will continue to use the brain MRI dataset from the previous practicals. The Jupyter Notebook provided contains some preliminary code you can use and some function prototypes that you are expected to fill in. The deliverables for the submission an archive containing the code provided completed as well as a short report explaining your strategies and choices for each task in this practical.

Task 1: *We will concern ourselves with segmentation again and we will be using the medical imaging library MONAI https://monai.io/. In this part of the exercise we will examine how our models respond to the low label regime. After establishing a full-data baseline, we will investigate how our results change if we were to only have three segmented brains as labels.*

1a: *Set appropriate arguments for the UNet class and train using the full dataset. Reduce the train dataset down to only three samples using the provided code and rerun.*
What happens to our training dynamics? Do we converge faster or slower? Is there a better way to select a model than a set number of steps?

After experiments with different architectures of UNet and completing the whole assignment we decided to choose the following structure, which is similar to one we have in the 2nd part of the assignment. This was done in order to compare different learning strategies.

```
model_unet = UNet(
 spatial_dims=3,
 out_channels=4,
 in_channels=1,
 channels=(64, 128, 256, 512, 1024),
 strides=(2, 2, 2, 2),
 num_res_units=1)
```

At first, we trained the net using the full dataset (200 samples) for 500 iterations and obtained these results at test time:

Test loss : 0.2449
Test Mean Dice : 0.8632
Test Dice_CSF : 0.8345
Test Dice_WM : 0.8832
Test Dice_GM : 0.8718

Secondly, we reduced the dataset to only 3 samples (“low-data mode”) and trained the same model. We achieved the following performance on the test set:

Test loss : 0.5270
Test Mean Dice : 0.7375
Test Dice_CSF : 0.7001
Test Dice_WM : 0.7517
Test Dice_GM : 0.7608

As we can see, the results are much worse for low data. We can now analyze what happens to our training dynamics :

Step		Full Data	Low Data
50	Train loss	1.3721	1.2798
	val loss	0.8540	0.7516
100	Train loss	0.6355	0.5508
	val loss	0.4841	0.4809
200	Train loss	0.3521	0.2737
	val loss	0.3232	0.3981
300	Train loss	0.2942	0.1689
	val loss	0.2894	0.4228
450	Train loss	0.2365	0.0500
	val loss	0.2389	0.4894

Using full data we converge slower with train loss, as the dataset is larger than in law-data mode, but validation loss after a certain point starts to increase, as 3 samples are not enough to learn all important features for accurate segmentations.

We suggest using a smaller model for law data training instead of adjusting the number of steps. So we decided to improve the architecture of the net and adapt it for low-data mode. The new architecture was:

```
model_unet_lowdata = UNet(
 spatial_dims=3,
 out_channels=4,
 in_channels=1,
 channels=(8, 16, 32),
 strides=(2, 2),
 num_res_units=1)
```

With these model we achieved the following results, which are much better in comparison to previous architecture:

Test loss : 0.4242
Test Mean Dice : 0.7646
Test Dice_CSF : 0.7440
Test Dice_WM : 0.7888
Test Dice_GM : 0.7610

1b: *Implement early stopping and rerun the experiments. Try different patience values and comment on how the selected patience influences validation/test performance. How could we set a high patience but still select the optimal model? (Note: You do not need to implement this)*

For early stopping we tried the following values for patience: 1, 3, 5, 20

- Patience = 1 : stopped step 900

```
Test loss: 0.2169
Test Mean Dice: 0.8787
Test Dice_CSF: 0.8481
Test Dice_WM: 0.8966
Test Dice_GM: 0.8916
```

- Patience = 3: stopped step 900

```
Test loss: 0.2284
Test Mean Dice: 0.8746
Test Dice_CSF: 0.8476
Test Dice_WM: 0.8931
Test Dice_GM: 0.8831
```

- Patience = 5: stopped step 1050

```
Test loss: 0.2332
Test Mean Dice: 0.8733
Test Dice_CSF: 0.8468
Test Dice_WM: 0.8924
Test Dice_GM: 0.8808
```

- Patience = 20 : stopped step 1900

```
Test loss: 0.2640
Test Mean Dice: 0.8647
Test Dice_CSF: 0.8373
Test Dice_WM: 0.8822
Test Dice_GM: 0.8747
```

With those 3 different patience values the training always stops at a similar step number. The use of patience has a negative influence on the test loss because the loss decrease happens gradually without jitter and therefore patience does not really help to bridge the loss decrease gaps. The model is also further away from the optimal (lowest loss) state.

To still keep the optimal model, even with high patience values, we could save the model parameters each time a new lowest loss is reached. Those parameters can then be loaded to have the optimal model during testing.

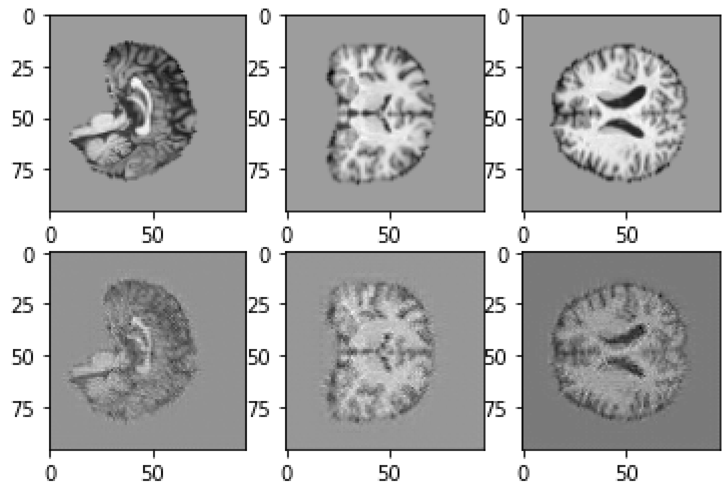
Task 2: *With such few samples our average Dice score has fallen by quite a bit. Lets try some unsupervised learning strategies to make use of our data that don't have labels.*

2a: *Train an Autoencoder*
Fill the appropriate arguments for UNet. Take care when setting the number of out channels. Visualize the reconstructed results of the autoencoder below the original images.

We used the same architecture for autoencoder as in the 1st part.

```
model_ae = UNet(
 spatial_dims=3,
 in_channels=1,
 out_channels=1,
 channels=(64, 128, 256, 512, 1024),
 strides=(2, 2, 2, 2),
 num_res_units=1)
```

After training the autoencoder we got the following reconstruction of the image:



We can observe that the reconstructed image is quite similar to the original one. The MAE score of reconstruction equals 0.0190.

2b: Transfer Learning
Initialize a new model and transfer the encoder weights from the trained autoencoder to the new model. Freeze the encoder weights and retrain. You should see a mean Dice score improvement between 2 and 3 points depending on your architecture

Without pre training we got the following results for low data :

Test loss : 0.5270
Test Mean Dice : 0.7375
Test Dice_CSF : 0.7001
Test Dice_WM : 0.7517
Test Dice_GM : 0.7608

Now, by transferring the encoder weights to the model and freezing these weights when training, we achieved the following results :

Test loss: 0.3608
Test Dice: 0.7939
Test Dice_CSF: 0.7767
Test Dice_WM: 0.8183
Test Dice_GM: 0.7868

We can then clearly see that the test loss is smaller and that the mean Dice score is higher. Indeed, the second model performs better because it uses the autoencoder weights that contain information about the features.

Task 2c: *Experiments*
When doing some form of pretraining there usually are quite a lot of decisions that need to be made. Take the time to explore the alternative options and report your findings. What happens if we leave the network fully trainable? What happens if we train our autoencoder to convergence? What about different architectures? Skip connections? Try some things out and report back!

1) Different architectures

We experimented with the number of layers (channels var) and number of channels in each layer. Here we show the test results for UNet with the following architecture:

```
model_ae = UNet(
 spatial_dims=3,
 in_channels=1,
 out_channels=4,
 channels=(16, 32, 64, 128, 256),
 strides=(2, 2, 2, 2),
 num_res_units=1)
```

Test loss: 0.4482
Test Dice: 0.7660
Test Dice_CSF: 0.7440
Test Dice_WM: 0.7847
Test Dice_GM: 0.7693

We can see that reducing the number of channels in each layer leads to worse results, though we don’t claim our architecture produces the best results.

Reducing the number of layers also leads to score decline, which can be explained with less features learned during encoder training.

```
model_ae = UNet(
 spatial_dims=3,
 in_channels=1,
 out_channels=4,
 channels=(64, 128, 256),
 strides=(2, 2),
 num_res_units=1)
```

Test loss: 0.6271
Test Dice: 0.7513
Test Dice_CSF: 0.7301
Test Dice_WM: 0.7687
Test Dice_GM: 0.7551

2) Skip connections

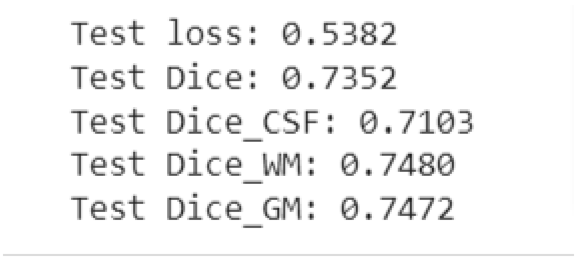
We tried setting 3 different values for the number of residual units: [0, 1, 2]. The highest segmentation results were obtained with only one skip connection in each layer. The use of residual units is beneficial as it simplifies training and speeds convergence, though setting the value of this parameter more than 1 led to worse performance.

3) Training autoencoder to convergence

Our architecture requires learning a lot of parameters, so we didn’t manage to train the autoencoder to full convergence. Nevertheless, the longer training (~650 steps) allowed us to achieve a slightly better score during test time (Dice(long train) - Dice(500 steps) = 0.007).

4) Unfreezing encoder part

We left the encoder section fully trainable and observed the following results:



```
Test loss: 0.5382
Test Dice: 0.7352
Test Dice_CSF: 0.7103
Test Dice_WM: 0.7480
Test Dice_GM: 0.7472
```

In comparison to the half-trainable network, the results are worse with approximately 0.6 dice points difference, but remained similar to the results with low-data training from 1st part. This can be explained by the fact that the network updates the learned features in the encoder part and partially forgets them, because it is trained with only 3 samples.

Conclusion:

- Supervised learning gives us the best results in comparison to other techniques;
- Low data training mode requires a simpler architecture of the net for achieving better

results, though it can not perform the same as the networked, trained with full data;
- Using an autoencoder we can learn the features from the data and therefore use its weights in our main task (ex. segmentation) with low data. This transfer learning approach allows us to achieve higher scores than general low data training.