University of Oxford

Honour School of Mathematics
and Computer Science

Computer Science Project (Part C)

Applications of Reinforcement Learning to Medical Image Segmentation

Author:
Edoardo Pirovano

Supervisor:
Dr. Irina Voiculescu

May 2017

Abstract

This project investigates the application of reinforcement learning techniques to the segmentation of medical images. In particular, we present a **novel approach** that is based on learning how to grow a selection with regions obtained from an image partition forest (IPF) based on various attributes of the regions. Our algorithm will be almost automatic, although we discuss why it cannot be classed as entirely automatic.

We then proceed to quantitatively evaluate this method against two datasets of manually segmented femurs in MRI scans of knees and compare its performance to that of state-of-the-art algorithms. We also discuss advantages that our approach provides, such as achieving results in a much shorter time and with significantly less expensive data labelling.

Contents

1 Introduction **7**

2 Background **9**
2.1 Reinforcement Learning 9
2.2 Windowing 10
2.3 Image Partition Forests 11
2.4 OxMedIS 11
2.5 Evaluation Measures 11

3 Related Work **15**
3.1 Knee Segmentation 15
3.2 Reinforcement Learning-based Segmentation 16

4 Design and Implementation **17**
4.1 Design Overview 17
4.2 Implementation 17
4.2.1 Input and Preprocessing 17
4.2.2 Segmentation and Learning 19
4.2.3 Postprocessing and Outputting 20
4.3 Adapting to Other Tissues 20

5 Materials **21**
5.1 Botnar Research Centre Data 21
5.2 SKI10 Grand Challenge Data 23
5.3 X-Ray Data 25

6 Parameter Selection **27**
6.1 Choice of Inclusion Criteria 27
6.2 Choice of ε and Number of Training Runs 29

7 Evaluation **31**
7.1 Results on Botnar Data 31
7.2 Results on SKI10 Data 35
7.3 Investigation Into Other Data Types 37

8 Critical Analysis **39**
8.1 Paucity of Data 39
8.2 Extent of Automation 39
8.3 Brief Comparison to SKI10 Competitors 41

9 Conclusion **43**
9.1 Discussion of Main Advantages 43
9.2 Future Work 43

10 Acknowledgements **45**

Bibliography **46**

Appendix A Data Parameters **49**
A.1 Botnar Research Centre Data 49
A.2 SKI10 Grand Challenge Data 49

Appendix B Code **50**
B.1 build.sbt 50
B.2 rl 50
B.2.1 Policy.scala 50
B.3 parser 52
B.3.1 Parser.scala 52
B.3.2 IPF.scala 53
B.3.3 MFS.scala 57
B.4 image 58
B.4.1 Raw.scala 58
B.4.2 WindowedImage.scala 61
B.5 segmentation 63
B.5.1 RegionInfo.scala 63
B.5.2 SegmentationResult.scala 63
B.5.3 Selection.scala 65
B.5.4 RLSegmentation.scala 67

List of Figures

2.1 Interaction between an agent and its environment 9
2.2 Example of a windowing function 10
2.3 Example of the first few layers of an IPF 12
(a) Leaf Layer 12
(b) First Branch Layer 12
(c) Second Branch Layer 12
(d) Third Branch Layer 12
2.4 Two example segmentations with the same DSC 14
(a) First Segmentation 14
(b) Second Segmentation 14
3.1 Example of a selection created from thresholding an image 15
(a) Original Image 15
(b) Threshold-Based Selection 15
4.1 The control flow during a run of our algorithm 18
4.2 Example of a selection before and after applying morphological closing 20
(a) Original Image 20
(b) Selection Before Closing 20
(c) Selection After Closing 20
5.1 An example slice from the Botnar data-set 22
5.2 An example slice from the SKI10 data-set 24
5.3 The three X-Ray images used 26
(a) Training Image X 26
(b) Training Image Y 26
(c) Evaluation Image Z 26
6.1 An image and its corresponding gradient images 27
(a) Original Image 27
(b) X Gradient 27
(c) Y Gradient 27
7.1 Improvement in segmentation during training runs 32
7.2 Example of results for one of the images in the Botnar data-set . . 34
(a) Image 34
(b) Gold Standard 34
(c) Before Training 34
(d) After Training 34
7.3 Example of results for one of the images in the SKI10 data-set . . . 36
(a) Image 36
(b) Gold Standard 36
(c) Before Training 36
(d) After Training 36
7.4 Results of the evaluation image for our X-Ray data 38
(a) Image 38
(b) Gold Standard 38
(c) Before Training 38
(d) After Training 38

List of Tables

6.1 Results for our experiment to determine optimal inclusion criteria . 28
6.2 Results of varying ε and the number of training runs 30
7.1 Results for the Botnar data-set 31
(a) Full Results 31
(b) Mean Results 31
7.2 Part of the policy learned by our agent 32
7.3 Results for the SKI10 data-set 35
(a) Full Results 35
(b) Mean Results 35
7.4 Results for our X-Ray data (images X-Z). 37
8.1 Results for teams entering the SKI10 challenge 42
8.2 Results for the SKI10 data-set in terms of AvgD and RMSD 42
(a) Full Results 42
(b) Mean Results 42
A.1 Details of the images used from the Botnar data 49
A.2 Details of the images used from the SKI10 data 49

1 Introduction

The problem of *segmentation* is that of splitting an image into meaningful regions. In particular, in the context of medical images, this usually means separating out organs or bone from the surrounding tissues. Closely related is the problem of *feature identification* which centres around identifying these regions. In this project, we will focus on the former problem.

Segmentation is an important problem for many clinical applications. For instance, knowledge of the volumes of tumours can be used by doctors to guide decisions on treatment and asses the success of such treatments. Additionally, algorithms have been developed to convert scans labelled with locations of organs into 3D meshes [1], allowing doctors to directly visualize a patient's organs or bone.

The segmentation of femurs, which we will focus on in this project, has applications to *patient-specific instruments* (PSI), which take into account the shape of an individual patient's knee in knee-replacement surgery, with the aim of improving postoperative alignment and reducing surgery time [2]. PSI has seen a large increase in popularity in recent years [3].

Manual segmentation of scans by radiologists is very time-consuming, so much research attention has been dedicated to developing algorithms to perform segmentation. These can roughly be divided into two types: *Semi-automatic* algorithms rely on some user input, whereas *automatic* ones do not. The algorithm we present here is almost fully automatic, although we will see in Section 8.2 why it cannot be fully classed as this.

The rest of this report is organised into eight main sections. Section 2 gives some background on the ideas underlying our approach. Section 3 presents some related work. Section 4 describes the design and implementation of our method. Section 5 describes the data-sets we will use for training and evaluation of our algorithm. Section 6 describes a couple of experiments used to decide some important parameters for our algorithm. Section 7 contains an evaluation of the effectiveness of our method on segmenting femurs in scans of knees. Section 8 gives an analysis of the successes and shortcomings of our approach. Finally, Section 9 gives some concluding remarks and points out some areas for further work.

2 Background

2.1 Reinforcement Learning

Reinforcement learning [4] is the problem of learning what decisions (*actions*) to take in a given situation (*state*) to maximise a numerical reward function by a process of trial-and-error. Depending on the problem and its formulation, the action taken by the learner (*agent*) may affect not only the immediate reward but also subsequent rewards. This interaction between an agent and its *environment* can be summarised by Figure 2.1.

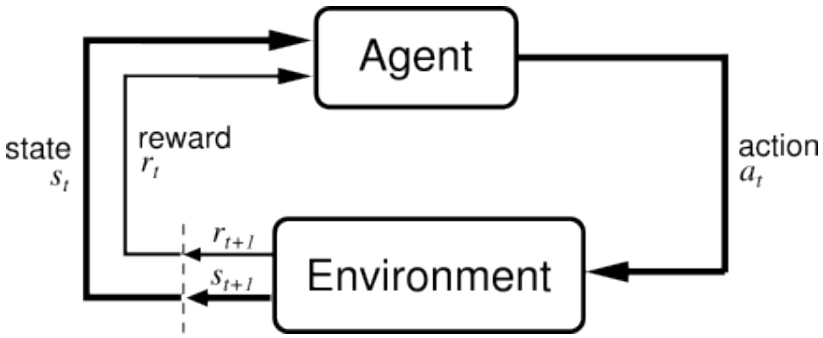


Figure 2.1: *A diagram showing the interaction between an agent and its environment. Figure obtained from [4].*

While reinforcement learning can be classified as machine learning in the sense that it "gives computers the ability to learn without being explicitly programmed,"[1] it is very different from other types of machine learning.

Most of machine learning relies on statistical pattern recognition while considering the entirety of a large set of data at once. **In contrast, in reinforcement learning, all the learning occurs from interacting with the environment and trying different actions.**

As a result of this, reinforcement learning can sometimes yield methods that train more quickly and with significantly less training data (as will be the case in this project). This is particularly desirable in medical imaging where obtaining training data is time-consuming and expensive.

One important aspect of reinforcement learning is finding a balance between *exploration* (making different choices in order to see how well they work) and *exploitation* (making what we believe to be the best choice in order to maximise reward). One common way of achieving this is by an ε-greedy policy, under which the agent makes the choice of action it believes to be best with probability $1-\varepsilon$ for some, usually small, parameter $0 < \varepsilon < 1$ and otherwise chooses an action uniformly at random. The parameter ε can be fixed (this will be the case in this project, as discussed in Section 6.2) or can be made to vary according to some heuristic (for example, it can decrease as time goes on and our agent has more knowledge).

[1] The definition of machine learning given by Arthur Samuel, one of the pioneers of the field, in 1959.

Reinforcement learning has been successfully applied in many areas including telecommunications [5], elevator control [6], and games such as Backgammon [7] and Go [8]. Applying the same techniques to medical imaging is in some respect more challenging: we have access only to a limited amount of training data and no way of generating more since we cannot simulate the environment of the agent. There have nonetheless been some successful applications, which are discussed in Section 3.2.

2.2 Windowing

The unit recorded by a CT scanner is known as the *Hounsfield Unit* (HU). HU values have a very large range, so it is usually impractical to work on them directly. In order to focus on a smaller sub-section of the values that we are interested in, we use a *windowing* for the scan.

A windowing is composed of a *window centre* and a *window width*, which define the centre and size of the range of HU values we are interested in. Together, these define a piece-wise linear function that re-scales the intensity of each pixel into a value within our desired range (which throughout this project will be 0 to 255). An example of this can be seen in Figure 2.2.

MRI scans use different techniques, depending on the type of the scan and its parameters. Nevertheless, they generate a wide range of intensities, which need to be windowed in a similar manner.

While techniques exist for automatically choosing windowings for scans, for the purposes of this project we have selected suitable windowings manually as this allowed us to obtain better results. Once the work in this project is integrated into a larger medical image segmentation system, automated windowing will be looked at as a separate task.

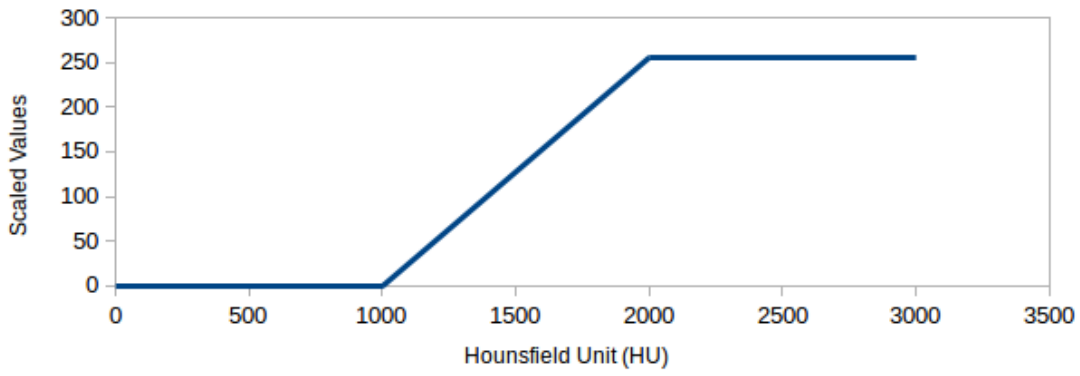


Figure 2.2: *An example of how a windowing with a centre of 1500 and a width of 1000 would re-scale HU values into our desired range of 0 to 255.*

2.3 Image Partition Forests

Another key concept used throughout this project is that of *Image Partition Forests* (IPFs). A full presentation of IPFs and how they are constructed is beyond the scope of this report and can be found in [9]. We however give a brief summary here since they are so central to the project.

An IPF (see Figure 2.3 for an example), is a forest of a chosen fixed depth constructed from a given image. Nodes in the leaf layer correspond to individual pixels in the image and nodes in higher up layers represent regions corresponding to the union of **connected** nodes below them. Notice that (by the fact the IPF is a forest) each layer thus corresponds to a "mosaic" that splits the image into a number of disjoint regions, with higher up layers giving a coarser mosaic.

Although this need not strictly be the case from the definition, the idea of IPFs is usually to group together pixels that have similar intensities and will likely be part of the same region in order to aid segmentation.

2.4 OxMedIS

The IPFs used in this project are constructed using OxMedIS,[2] a software tool for segmenting medical images using methods discussed in [9]. The first step in constructing an IPF is smoothing the data with several passes of anisotropic diffusion filter [10], which reduces the effect of noise in the image on the final result.

The segmentation then starts from the leaf layer (which is just every pixel in the image), repeatedly sets every pixel in each region to the maximum intensity in that region and then performs a watershed [11] on the transformed image to get the next layer up. This method can be used on either windowed or unwindowed data, but in this project we use the windowed data since this yields better results.

2.5 Evaluation Measures

There are many different quantitative measures to evaluate the success of a segmentation [12]. For our purposes, it will suffice to consider just a few of these (throughout, let MS be a set of pixels denoting a machine segmentation and GT another set of pixels denoting a gold standard for the scan):

- **DSC:** We define *Dice's Similarity Coefficient* by:

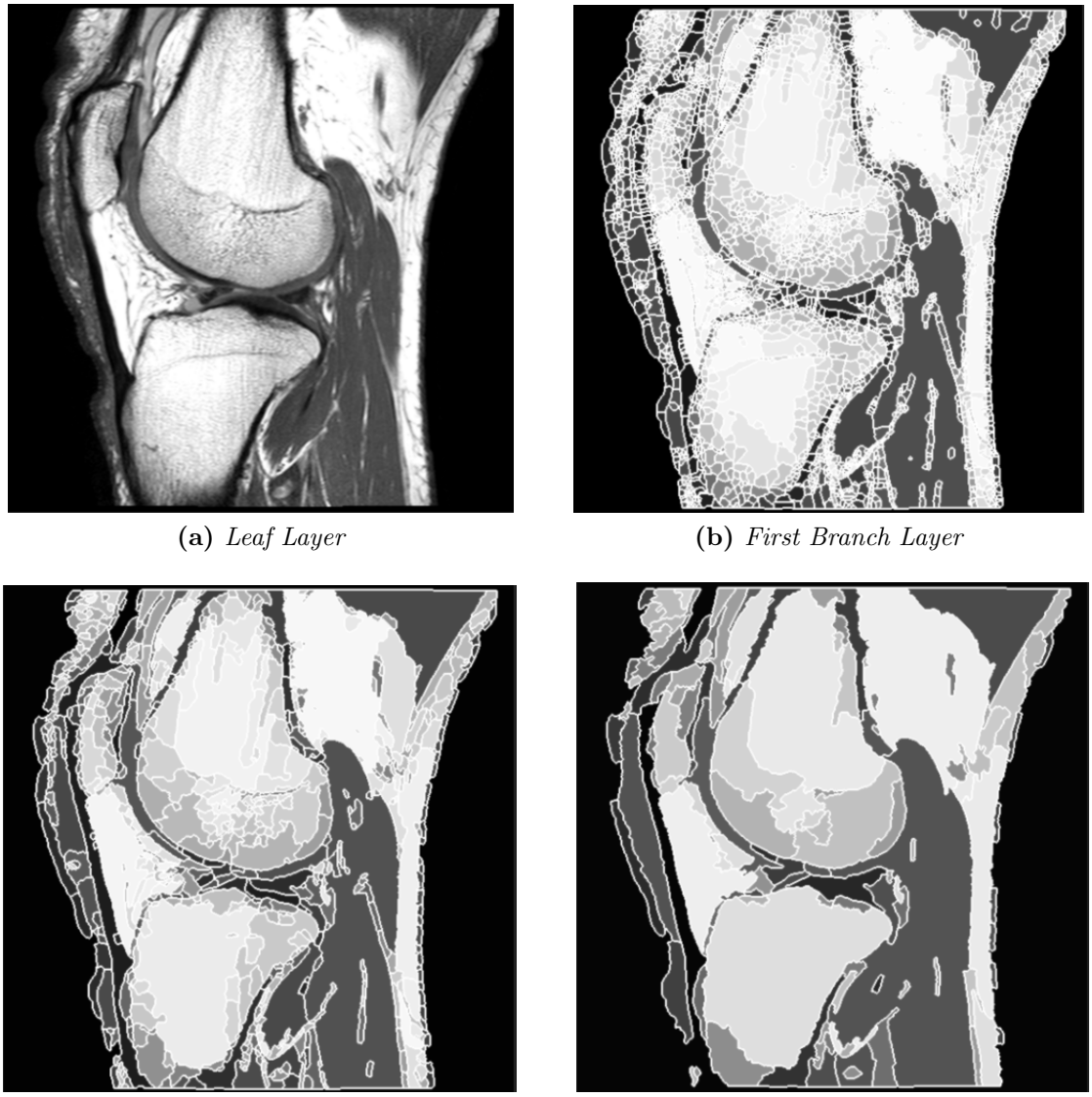
$$\mathrm{DSC} = \frac{2\,|MS \cap GT|}{|MS| + |GT|}.$$

  Intuitively, this gives a value between 0 and 1 that is closer to 1 the more similar MS is to GT.

- **TPVF and FPVF:** Additionally, let I denote the whole image. Then, the *true positive* is given by:

$$\mathrm{TPVF} = \frac{|MS \cap GT|}{|GT|}.$$

[2] `https://www.cs.ox.ac.uk/projects/OxMedIS/`



(a) *Leaf Layer*

(b) *First Branch Layer*

(c) *Second Branch Layer*

(d) *Third Branch Layer*

Figure 2.3: *Example of the first few layers of an IPF (layers above the third branch layer are not used in this project since the selections they give are often too coarse to be useful).*

Similarly, the *false positive* is given by:

$$\text{FPVF} = \frac{|MS \setminus GT|}{|I| - |GT|}.$$

While slightly less immediately intuitive, together these give us more information than DSC since they tell whether we identified too much or too little.

- **AvgD:** First, for a voxel x and a set of voxels A, let us define:

$$d(x, A) = \min_{y \in A} d(x, y),$$

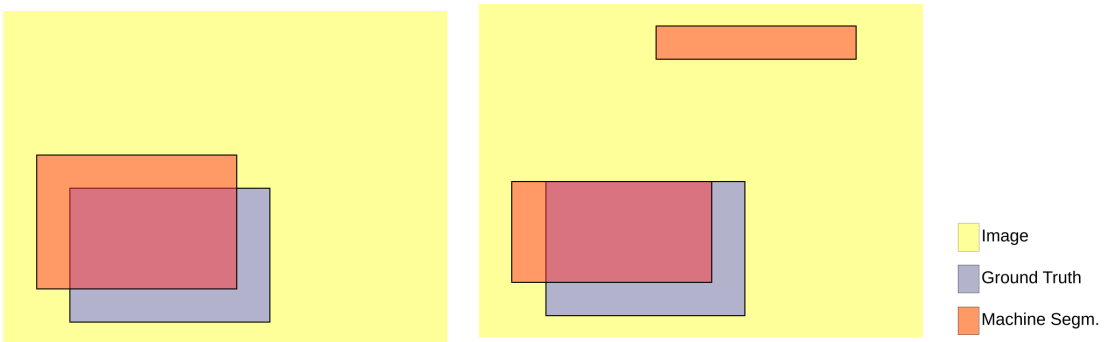
where $d(x, y)$ is the Euclidean distance between between two voxels. Let us also define B_A to be the boundary of a set of voxels A. Now, we can define the *average symmetric surface distance* by:

$$\text{AvgD} = \frac{1}{|B_{MS}| + |B_{GT}|} \left(\sum_{x \in B_{MS}} d(x, B_{GT}) + \sum_{y \in B_{GT}} d(y, B_{MS}) \right).$$

- **RMSD:** Similarly, we can define the *root mean square symmetric surface distance* by:

$$\text{RMSD} = \sqrt{\frac{1}{|B_{MS}| + |B_{GT}|} \left(\sum_{x \in B_{MS}} d^2(x, B_{GT}) + \sum_{y \in B_{GT}} d^2(y, B_{MS}) \right)}.$$

Notice that while DSC, TPVF and FPVF are ratios between 0 and 1, AvgD and RMSD are physical distances. There are many more measures we do not use in this report, and much discussion is currently on-going in the literature on which measures are most appropriate for which studies [13, 14, 15]. DSC is by far the most frequently encountered in literature, but it has significant shortcomings. For example, in Figure 2.4 we see two segmentations where the first is significantly more useful but both have the same DSC.



(a) *First Segmentation* **(b)** *Second Segmentation*

Figure 2.4: *Two segmentations that would be scored the same by DSC and other evaluation measures based on overlap. Figure obtained from [12].*

3 Related Work

In this section, we give a brief summary of work that has already been performed in the area of segmentation of knees and more generally segmentation based on reinforcement learning.

3.1 Knee Segmentation

Algorithms exist to segment the various parts of a knee: femur, tibia and cartilage. We will focus on the problem of segmenting the femur. Current state-of-the-art knee segmentation algorithms rely on a variety of different methods (and, often, combinations of several of these), including [9]:

- **Thresholding:** Selecting precisely those pixels with a certain range of intensity values. An example of this can be seen in Figure 3.1.
- **Region Growing:** Growing out a selection, starting from either a single pixel or a group of pixels and adding either a pixel or a group of pixels at a time, based on a set of criteria.
- **Deformable Models:** Starting from a model of the feature we wish to identify (for example, the typical shape of a femur), and then trying to find a suitable place in our image for it.
- **Clustering:** Attempting to collect pixels into groups that are similar based on certain criteria.
- **Atlas-based:** Constructing a probabilistic map of certain positions being certain features (based on training data) and then using this to guide segmentation of new images. Crucially, this relies on an aligning step that re-orients and re-scales the images so that relative positions make sense.

(a) *Original Image* **(b)** *Threshold-Based Selection*

Figure 3.1: *Example of a selection created from thresholding an image. Obtained from Wikimedia Commons (where it was released into public domain).*

As described in Section 4, our algorithm is based on a region growing method (in some sense combined with the clustering given by an IPF), in which we use reinforcement learning to learn the criteria for how to expand our region.

We will now briefly outline the results obtained by a few state-of-the-art algorithms to give us something to compare our own results with. It is worth noting that comparing results across different studies is somewhat challenging because different studies will use different sets of data and evaluation measures. One of the few publicly available data-sets with evaluation benchmarks is the one from the SKI10 grand challenge, which we will also use and describe in more detail in Section 5.2. The data-set is fairly challenging to segment in that the images have a low contrast.

One method [16] that works primarily by L_0 gradient minimization combined with several preprocessing and postprocesing steps achieves an average DSC of 0.949 ± 0.015 on a small subset of the SKI10 grand challenge data-set.

Another method [17] that relies on a 3D active shape model initialised using an atlas achieves a similar average DSC of 0.952 ± 0.072 and AvgD of 0.16mm, though the data-set used to evaluate this method was likely less challenging.

Another paper [18] using an atlas followed by region adjustment achieves a DSC of 0.717 ± 0.080 on the SKI10 data-set, which while significantly lower than that achieved by others [16] is still interesting because its evaluation was carried out on the whole data-set rather than a selected subset of it.

One paper [19] uses a method based on ray-casting which relies on the decomposition of the MRI images into multiple surface layers to localize the boundaries of the bones to achieve a DSC of 0.94 ± 0.05 and an AvgD of 0.19 ± 0.02mm on a set of 141 MRI scans (again, likely from a less challenging data-set than the SKI10 one).

3.2 Reinforcement Learning-based Segmentation

There have been a number of previous attempts to use reinforcement learning in the context of image segmentation. One paper [20] tackles the problem of identifying the prostate in an ultrasound image. This works by splitting the image into various sub-images and then learning a greyscale value threshold for each sub-image to determine which pixels to include. Another paper [21] uses a similar method on CT images of various different body parts.

Yet another paper [22] takes a different approach. Instead of learning how to segment an image, it instead learns which from a library of standard segmentation algorithms to use and which parameters to initialize it with.

Here, we have taken a novel approach: we use a user-provided seed (initial position in the image that is part of our desired region) and then grow our selection from there adding small regions from the lowest branch layer of the IPF, with the agent learning which regions should be included and which should not. We describe our approach in more detail in the next section (although some details of how the algorithm is configured are deferred till Section 6).

4 Design and Implementation

In this section, we will outline the design and implementation of a novel algorithm for tissue segmentation based on reinforcement learning techniques. We omit code from here, though the full code for our Scala implementation can be found in Appendix B and we will occasionally refer to files there in our description.

It is worth noting that while the examples we will see in our evaluation later on are 2D images, the implementation fully supports 3D volumes as well.

4.1 Design Overview

The basic control flow of our program in a run is shown in Figure 4.1. Essentially, the image and IPF are read in and parsed, along with a user-provided seed point in the image that is known to be part of the area we wish to segment.

This seed is then used to pick the corresponding region in the second or third branch layer of the IPF. Then, our agent keeps considering regions in the first branch layer that are adjacent to our selection and deciding whether to include them or not until there are no adjacent regions that we have not already excluded. This uses an ε-greedy policy for training runs, with a configurable parameter ε, and simply makes the choice it believes to be best in evaluation runs.

Once our selection is complete, if we are performing a training run we go back over the decisions that we made and update our policy based on whether they were correct or not. Finally, we morphologically close our result and then output it, along with various scores.

4.2 Implementation

We now proceed to describe each of these stages in more detail, along with descriptions of where to find the corresponding code should the reader wish to look in detail at this.

4.2.1 Input and Preprocessing

If we exclude the seed points and windowings (which are programmed into constants in the code), there are three inputs to the program: the image to be segmented, its IPF, and the gold standard to give a score for our result and, in the case of training runs, train our policy.

In many cases, the reading of the input image is handled by the ImageJ library [23], although this struggled with MetaImage and RAW files, so a parser for these was implemented in *Raw.scala*.

As mentioned in Section 2.4, the IPFs are generated using OxMedIS. The output from OxMedIS is parsed by our code in *IPF.scala*. The parser discards some of the information we are not interested in but stores the information we do want such as the hierarchy and adjacency graph [24].

The gold standard can also be an image, with white pixels representing the pixels in the selection and black pixels representing those not. Additionally, there is limited support for *multi-feature selection* files exported by OxMedIS. As their

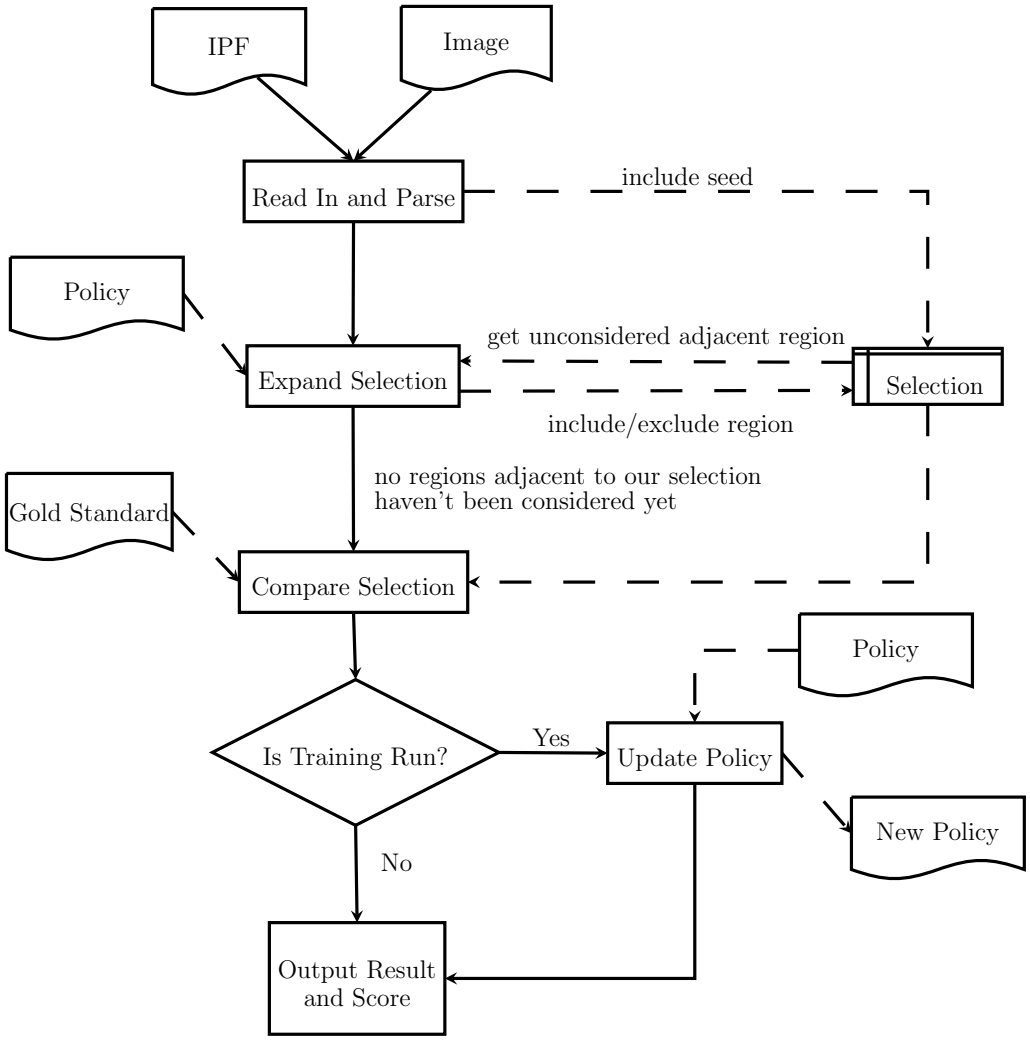


Figure 4.1: *The control flow during a run of our algorithm. Note for simplicity of the diagram we represent the policy as an immutable file that is input and output each run, but in practice it is stored in memory and mutated in-between runs. Similarly, the image, IPF and gold standard are stored in memory between runs for the same image rather than read in and parsed again each run.*

name suggests, these files can define multiple different feature selections. However, for simplicity we implemented support only for those that define a single selection. Parsing of these selections is implemented in *MFS.scala*.

Before we begin to segment the image, we first window it and compute the gradients at every point (once again using code from the ImageJ library) to be used later. This is implemented in *WindowedImage.scala*.

4.2.2 Segmentation and Learning

We begin our segmentation by selecting a seed point. Then, we find the corresponding region in either the third or second branch layer of the IPF. We use the third branch layer for evaluation runs, since in all the images encountered in this project this never resulted in over-selection but did select a suitably large portion of the region we wished to segment in order to give a good seed. However, we use the second layer in training runs since this results in more regions being considered later on and thus improves the amount of learning we can obtain from an image.

The selection is kept track of in *Selection.scala*. The main loop in *RLSegmentation.scala* then repeatedly considers regions (in the **first** branch layer of the IPF - which is the smallest regions that are not single pixels) adjacent to our selection for inclusion using the policy in *Policy.scala* until we have considered and excluded every region adjacent to our selection.

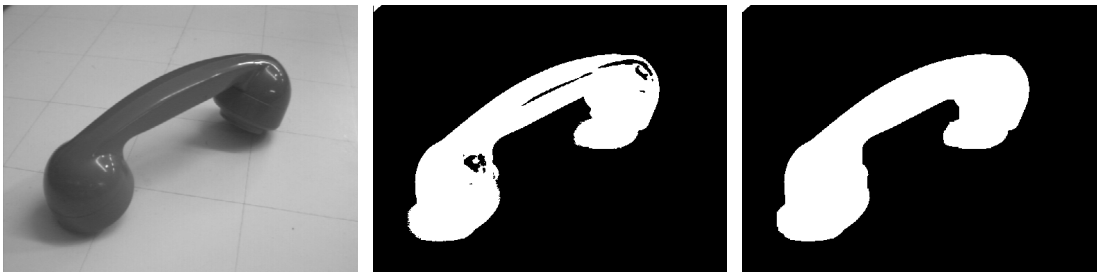
In training runs, we use an ε-greedy policy (with a configurable parameter ε) in order to encourage exploration. In evaluation runs, we always make what we believe to be the best choice. When we have no information about whether to include a region or not, the agent's default behaviour is to not include it. This usually results in better segmentation since if it was desirable to include the region it is likely that regions surrounding it will be included anyway and a post-processing step of morphologically closing the result (discussed in Section 4.2.3) will add it to the selection.

While the seed region is taken in one of the higher up layers, it is important to note that we then only use the first branch layer of the IPF to grow our selection. We make this choice since using any layer other than this to expand our selection with could make it impossible to achieve high segmentation accuracies on images where the higher up layers of the IPF contain regions that are only partly in the area we wish to select. While this could in theory also be an issue with the first branch layer, it is not in practice.

This problem could be avoided by giving the agent a third choice of action (in addition to including or excluding the region) when it is considering a region not in the first branch layer that allows it to split the region into its children and consider these separately. This possibility is left for future work, as discussed in Section 9.2.

Which information about the region is used by the agent when deciding whether or not to include it can be adjusted, and it is possible that different types of images will work better with different choices of information. We discuss various possibilities and evaluate their success in Section 6.1. For training runs, this information is cached across runs so we avoid recomputing it.

Once the segmentation process is complete, we go back over all our decisions and



(a) *Original Image* **(b)** *Selection Before Closing* **(c)** *Selection After Closing*

Figure 4.2: *Example of a selection before and after applying morphological closing. Figure obtained (with the author's permission) from [25].*

update the policy, giving a reward for each decision made. The reward is based on the number of voxels in the region that were correctly classified (each correct voxel gets a score of +1, and each incorrect one a score of -1). Thus, correct decisions get an overall positive reward, and incorrect ones a negative one.

4.2.3 Postprocessing and Outputting

We apply a further step of morphologically closing the segmentation result. This adds to our selection any regions that are surrounded by selected regions, thus ensuring the selection does not contain any holes (which can occur due to, for example, noise in the image). An example of this can be seen in Figure 4.2. Morphological closing is implemented in *SegmentationResult.scala* using the MorphoLibJ library [26].

Having done this, we compute DSC, TPVF and FPVF scores for the result and output them (this is done in *RLSegmentation.scala*). We also save a black and white image of the selection, with white representing voxels that our algorithm believes to be part of the selection. This saving is again handled by ImageJ.

4.3 Adapting to Other Tissues

This algorithm has been designed with the segmentation of femurs from MRI scans of knees in mind, as this is what our evaluation (in Section 7) focussed on. Nonetheless, it is likely that the same algorithm could be used to identify other features with only minor modifications to the code.

In particular, the criteria considered by our agent for deciding whether to include or exclude regions are hard-coded (in *RLSegmentation.scala*) as the maximum gradient and average intensity of the region since (as discussed in Section 6.1) these gave the best results for segmenting femurs. However, other criteria might work better for different tissues so an adaptation would likely require changing these.

5 Materials

In this section, we describe the data-sets we had access to, which will be used both for some experiments to select a suitable configuration for our algorithm (in Section 6) and for our evaluation (in Section 7).

5.1 Botnar Research Centre Data

The first data-set we used was obtained from the Botnar Research Centre (part of the Nuffield Department of Orthopaedics, Rheumatology and Musculoskeletal Sciences at Oxford). As described in [27, 28], the data consists of 18 knee MRI sequences taken with a Philips Achieva TX 3.0T MRI system, with a T1W contrast enhanced Turbo Spin Echo (TSE) sequence, with no fat suppression. The scans are saggital, meaning that the slices are taken in the plane that divides the body into right and left sides. The pixel spacing was 0.3mm by 0.3mm, and the slice thickness was 2.5mm (hence, the data is anisotropic[3]).

The scans were taken at the Botnar Research Centre in the Nuffield Orthopaedic Centre using the standard sequence employed in clinical practice within the UK's National Health Service. All scans were right knees of females with an average age of 49 (range: 31-63 years). Patients had been diagnosed with either a meniscal tear or patello-femoral arthritis. All scans were anonymised and were subject to ethical approval for use in research.

Of the 17 scans, we used a total of 10 and took a slice near the middle of the femur (see Figure 5.1 for an example) for each one. These 10 slices were then split into 5 training images and 5 evaluation images. The evaluation images were chosen to be those where the IPF gave a poor segmentation of the knee, since these highlighted improvements resulting from the learning.

Some of the patients had undergone surgery (for example, patient 2 had an implant attached to their femur). In these cases, the slices chosen from their scans were ones where there was no obvious resection or implants.

For all the chosen images, gold standards were made manually (by the author) and these were used for either training or evaluation. In Appendix A.1, we outline which slices were used and the windowing and seed point chosen for each.

[3]In this context *anisotropic* means not equally sized in all dimensions. In particular, in most medical images the distance between two slices of a scan is not equal to (usually larger than) the distance between the pixels within the slices.

Figure 5.1: *An example slice from the Botnar data-set. In particular, this is slice 11 of image 3, which we used as our second evaluation image.*

5.2 SKI10 Grand Challenge Data

The second set of data was obtained from the SKI10 grand challenge, a world-wide competition in knee segmentation [29] associated with MICCAI, a top international imaging conference. The data-set is designed to present difficulties for many algorithms. As described in the paper setting out the competition, the data consists of 150 MRI scans of knees, which were acquired from a large number of different centres (over 80), using various machines and settings. The images come with gold standard segmentations made manually by experts at Biomet, Inc.

It is worth noting that all the images were used for surgery planning of partial or complete knee replacement and thus the segmentation accuracy is usually much higher in the part of the knee relevant to the surgery and less so in the rest.

All images were acquired in the sagittal plane (as with the Botnar data) with a pixel spacing of 0.4mm by 0.4mm and a slice distance of 1mm. No contrast agents were used which, as we will see in our evaluation, makes automatic segmentation much more difficult. Field strength was 1.5T in about 90% of the cases, the rest was acquired mostly at 3T, with some images acquired at 1T. The employed MRI sequences show a huge variety: the vast majority of images used T1-weighting, but some were also acquired with T2-weighting.

As with the Botnar data, we selected 10 individual slices near the middle of the femur (see Figure 5.2 for an example) from 10 different images. The scans we used and accompanying parameters are described in Appendix A.2.

The selected scans were chosen to be the ones with slightly better contrast than the others, although as we see in our evaluation they still presented more difficulty than the Botnar data.

Figure 5.2: *An example slice from the SKI10 data-set. In particular, this is slice 83 of image 52, which we used as our third evaluation image. Notice the lower resolution (the image is 301x347 pixels rather than 512x512) and poorer contrast between the bone and surrounding tissue compared to the example image from the Botnar data-set (Figure 5.1).*

5.3 X-Ray Data

Scans (whether MRI or CT) are expensive, take a long time and require sophisticated devices. In contrast, X-Ray machines are ubiquitous and acquire data significantly faster. While some conditions require detailed investigations with 3D scans to diagnose, others can be diagnosed from X-Rays. Therefore, it would be desirable for our method to also be effective on X-Rays.

For a small test on X-Rays we acquired three similar X-Ray images of a knee, shown in Figure 5.3. The images were provided already windowed, so selecting windoweings was not necessary. Unlike the Botnar and SKI10 scans, these images are coronal, meaning that the plane they are taken in corresponds to that dividing the back and front of the patient's body. These were manually segmented by the author to produce gold standards used for training and evaluation.

While this may appear to be a very limited amount of data, we will see in Section 7.3 that it will be enough to show that our method does not immediately transfer to X-Ray images due to the lower contrast between bone and the surrounding tissue in them compared to MRI scans.

(a) *Training Image X* **(b)** *Training Image Y* **(c)** *Evaluation Image Z*

Figure 5.3: *The three X-Ray images used.*

6 Parameter Selection

6.1 Choice of Inclusion Criteria

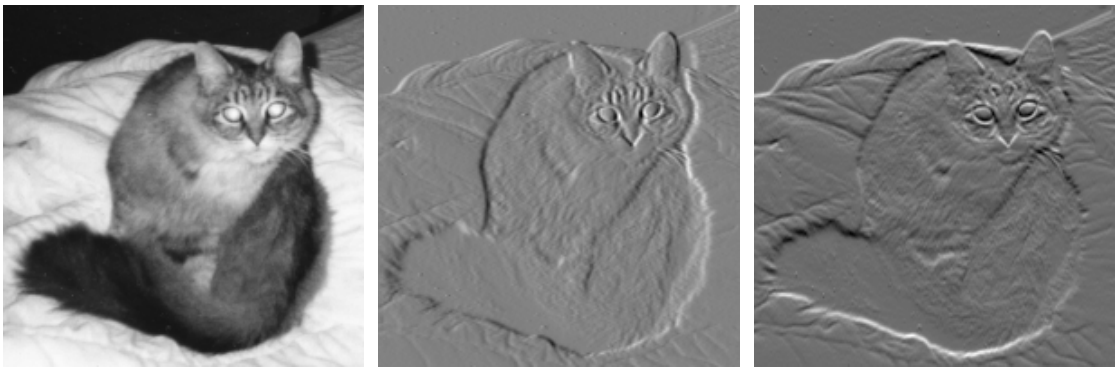
As discussed in Section 4, the segmentation program can be configured to present various different criteria to the agent when deciding whether or not to include a region into the segmentation. In particular, we will consider criteria based on two important characteristics of pixels:

- **Intensity:** The brightness of the pixel in the image, after our windowing has been applied. This will be in a range of 0 to 255.
- **Gradient:** For each pixel we can compute gradients by taking the derivative of the intensity values at that point (this is the change in intensity between the pixels either side divided by two), again after windowing, in each direction (as illustrated in Figure 6.1). We reduce these two (or three when working on voxels in 3D scans rather than single slices) components to one value by taking the largest. This will be in a range of 0 to 127.

We then have a choice of how to go from these values for individual pixels to an aggregated value for the entire region in the IPF. For this we consider taking the minimum, average or maximum for intensity and the average or maximum for gradient (minimum gradient is not a very useful measure since it is expected that within a region there will be areas with very similar intensity values making this always close to 0). Another interesting possibility could be considering the gradient only on the boundary of the region, but we leave this for future work.

When deciding how much information to present the agent, there is a certain trade-off. Presenting more information could allow the agent to make better decisions. However, it also increases the size of the state space, thus requiring a larger training data-set to encounter a large enough portion of the states to make the policy learnt effective.

Given the amount of training data we had available, presenting a pair of the values discussed above struck a good balance. This gives an upper bound on the



(a) *Original Image* **(b)** *X Gradient* **(c)** *Y Gradient*

Figure 6.1: *An image and its corresponding gradient images. Obtained from Wikimedia Commons (where it was released into public domain).*

	Min Intensity	Max Intensity	Avg Grad	Max Grad
Avg Intensity	0.707	0.751	0.743	**0.971**
Min Intensity	-	0.720	0.731	0.940
Max Intensity	-	-	0.738	0.952
Avg Grad	-	-	-	0.837
Max Grad	-	-	-	-

Table 6.1: *Mean DSC of a segmentation of the evaluation set after training on the training set, according to different choices of pairs of inclusion criteria.*

number of different states of $256^2 = 65536$. Because of dependencies between the different values, this upper bound is only theoretically possible for some pairs of values (for instance, minimum intensity and maximum intensity) and even then is extremely unlikely to occur in practice when working on medical images.

Then, it remained to decide what two values to consider. For this, we designed a small experiment. We took a set of 10 slices of scans with their corresponding gold standards from the Botnar Research Centre data (as described in Section 5.1) and split them into two equally sized sets: a training set, and an evaluation set.

We trained our agent on the training set (running it 40 times with $\varepsilon = 1/10$ on each image, as justified in the next section) and then ran it on our evaluation set and calculated the mean DSC of the results (as described in Section 2.5). We repeated this for a variety of different inclusion criteria. Our results are recorded in Table 6.1.

It is worth noting a few limitations of this experiment. Firstly, a different choice of evaluation measure might have given a different result. As such, if we were particularly interested in getting the best results under a certain measure (for example, if we wanted to keep false positive to a minimum), it would be beneficial to repeat this experiment with that measure. We chose to consider DSC since it is the most widely used in the literature.

Secondly, the results might have changed if we had used a different data-set. In particular, different inclusion criteria may be more or less useful depending on the parameters the scans were taken with or the feature that we are trying to segment (for instance, if we were trying to segment organs in a CT scan instead of bones in an MRI scan). Additionally, different sized data-sets might give different results: one choice of inclusion criteria might perform better with a smaller data-set (like ours) while another might do better when there is more data available to train on.

Nonetheless, we conclude that in our particular scenario the optimal choice of criteria is average intensity and maximum gradient. This choice makes sense because a high average intensity would tell us that the region is primarily bone and thus should be included, while a high maximum gradient would tell us that the region includes the boundary between the bone and the surrounding soft tissue and thus should not be included.

From this point in the report onwards, we fix our inclusion criteria as average intensity and maximum gradient.

6.2 Choice of ε and Number of Training Runs

It is worth giving a justification of our choice of ε and the number of training runs per image, which are (along with the choice of what criteria consider when deciding which regions to include/exclude) the other ways the algorithm can be configured.

Recall from Section 2.1 that the parameter $0 < \varepsilon < 1$ controls how much an agent following an ε-greedy policy will attempt actions it does not believe to be optimal: the agent makes the choice of action it believe to be best with probability $1 - \varepsilon$ and otherwise chooses an action uniformly at random.

For the purpose of choosing ε and the number of training runs, we ran our segmentation algorithm (once again, on the data from the Botnar Research Centre, as described in Section 5.1) with each of a variety of choices for them.

Again, the results show a mean DSC on the evaluation set after learning on the training set. Because of the high randomness involved with some choices of parameters, we ran this experiment 10 times for each configuration and took a mean of these means. These results can be seen in Table 6.2.

As with our previous experiment, the results and conclusions drawn here should be taken with a pinch of salt, since they are specific to a certain choice of evaluation measure and data-set.

The choice of ε is quite delicate. Choosing a large value like ½ means we can quickly (even with only a couple of runs per image) get sensible results. However, it also means that we will sometimes consider regions far from the femur, reducing the overall quality of the policy we learn since it will include information about regions that are not relevant.

On the other hand, choosing a small value like $\frac{1}{100}$ means we will need many training runs before we reach a sensible policy. This is because the agent will be reluctant to attempt to include new regions, resulting in learning taking more time.

So as a compromise we use an intermediate value of $\frac{1}{10}$. This is large enough to consistently give good results even with a relatively small number of training runs but not so large that it causes us to consider many irrelevant regions and learn a poor policy.

While smaller values likes $\frac{1}{20}$ and $\frac{1}{40}$ seem slightly more promising from the results in this section, we avoid them because they make learning less consistent since on particularly unlucky runs we might never consider important regions and end up learning a poor policy.

Notice that because of the non-determinism involved, a larger number of training runs does not always improve results so the DSC is not strictly increasing as we vary the number of training runs. Nonetheless, we can see that it does overall increase with the number of training runs, roughly stabilizing after around 40 runs (hence our choice of this).

From this point in the report onwards, we fix $\varepsilon = \frac{1}{10}$ and 40 training runs per image.

		Number of Training Runs					
		2	5	10	20	40	100
ε	$\frac{1}{2}$	0.789	0.831	0.838	0.826	0.823	0.827
	$\frac{1}{5}$	0.918	0.932	0.934	0.932	0.936	0.931
	$\frac{1}{10}$	0.918	0.947	0.956	0.959	0.963	0.957
	$\frac{1}{20}$	0.953	0.956	0.969	0.970	0.972	0.969
	$\frac{1}{40}$	0.939	0.963	0.946	0.956	0.973	0.969
	$\frac{1}{100}$	0.927	0.942	0.923	0.936	0.937	0.930

Table 6.2: *Mean DSC of a segmentation of the evaluation set after training on the training set, with a varying ε and number of runs.*

7 Evaluation

7.1 Results on Botnar Data

		Before Training			After Training		
		DSC	TPVF	FPVF	DSC	TPVF	FPVF
Training	A	0.830	0.710	0.000	0.998	0.996	0.000
	B	0.999	0.998	0.000	0.994	0.999	0.002
	C	0.998	0.996	0.000	0.999	0.997	0.000
	D	0.763	0.617	0.000	0.999	0.998	0.000
	E	0.864	0.760	0.000	0.998	0.998	0.000
Evaluation	F	0.657	0.489	0.000	0.971	0.944	0.000
	G	0.678	0.512	0.000	0.959	0.922	0.000
	H	0.762	0.616	0.000	0.969	0.939	0.000
	I	0.734	0.579	0.000	0.988	0.976	0.000
	J	0.566	0.395	0.000	0.975	0.952	0.000

(a) *Full Results*

	Before Training			After Training		
	DSC	TPVF	FPVF	DSC	TPVF	FPVF
Training Set	0.891	0.816	0.000	0.998	0.998	0.000
Evaluation Set	0.679	0.518	0.000	0.972	0.947	0.000

(b) *Mean Results*

Table 7.1: *Results for the Botnar data-set (images A-J).*

Having fixed all configurable parameters of our algorithm, we now give a more thorough evaluation of our algorithm's performance, once again on the data from the Botnar Research Centre described in Section 5.1. Notice that since this is the same data we used to configure our algorithm in Section 6, there may be some bias in our configuration that makes it particularly suited to this data. However, this will not be the case for the data discussed in the next sections.

In Figure 7.1 we can see the learning in action for the first knee in the training data. Notice the occasional upward jumps in the false positive (and corresponding drops in DSC) when the ε-greedy policy randomly decides to try to include a new region that is not part of the knee. The results for the other training images are similar, although they have better starting positions due to previous learning (this is good, but also makes the graphs less interesting to look at since there is less improvement to be made).

In Table 7.2 we can see a part of the policy learnt by our agent. Notice how, as expected, a high maximum gradient leads to the region being less likely to be included. The relationship between average intensity and inclusion is less obvious from this fragment but from the whole data it can also be seen to be as expected (higher intensities lead to higher likelihood of inclusion).

It is interesting to note that while this was chosen from a part of the policy where most of the states had been encountered, very few of the theoretically possible values

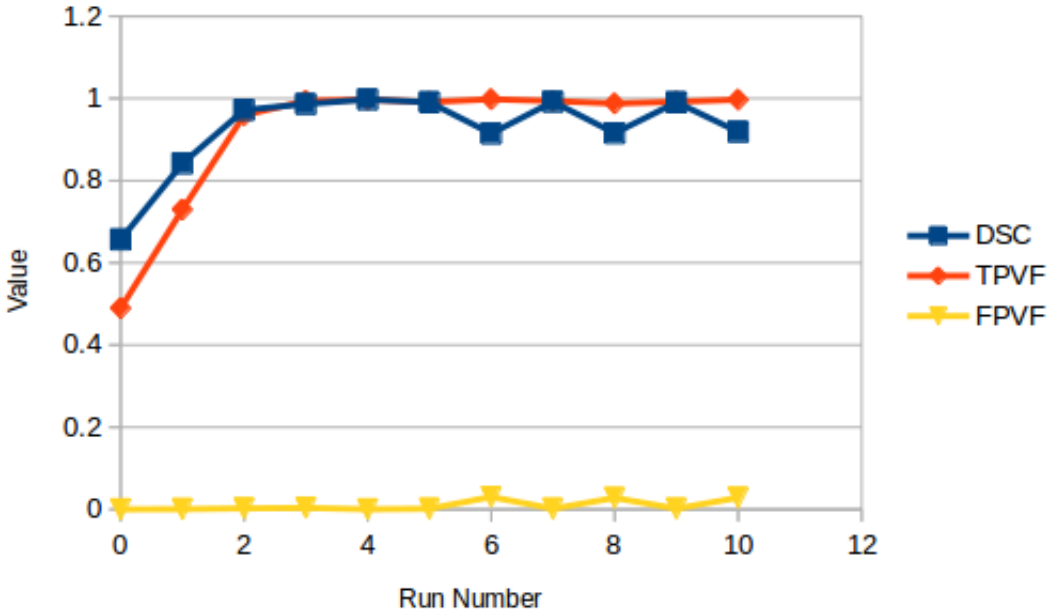


Figure 7.1: *The change in various evaluation measures over the course of 10 training runs on the first knee in the data-set. Notice the evaluation measures settle close to 1 (100%) after a few runs.*

		Maximum Gradient (Range: 0–127)					
...	...	...	...	...	...	...	...
...	107	108	109	110	111	...
Average Intensity (Range: 0–255)	129	Unseen	Include	Exclude	Exclude	Include	...
	130	Include	Include	Include	Include	Exclude	...
	131	Include	Include	Include	Exclude	Exclude	...
	132	Include	Include	Include	Exclude	Exclude	...
	133	Exclude	Include	Exclude	Include	Exclude	...
...	...	...	...	...	...	...	...

Table 7.2: *A part of the policy of whether or not to include regions learned by the agent after training on the Botnar data-set. The full table is 32640 cells big.*

for intensity and maximum gradient are actually encountered in real images - only around 8% (of the 32640 possible combinations) were encountered in the training data-set.

This does not necessarily mean the images did not contain regions with these values, but merely that our agent never considered those regions (since it will in most cases never explore very far from the femur). This highlights one of the advantages reinforcement learning gives over more conventional statistical machine learning techniques - we only consider the parts of the data that are useful to our agent's decision making and, thus, can train faster.

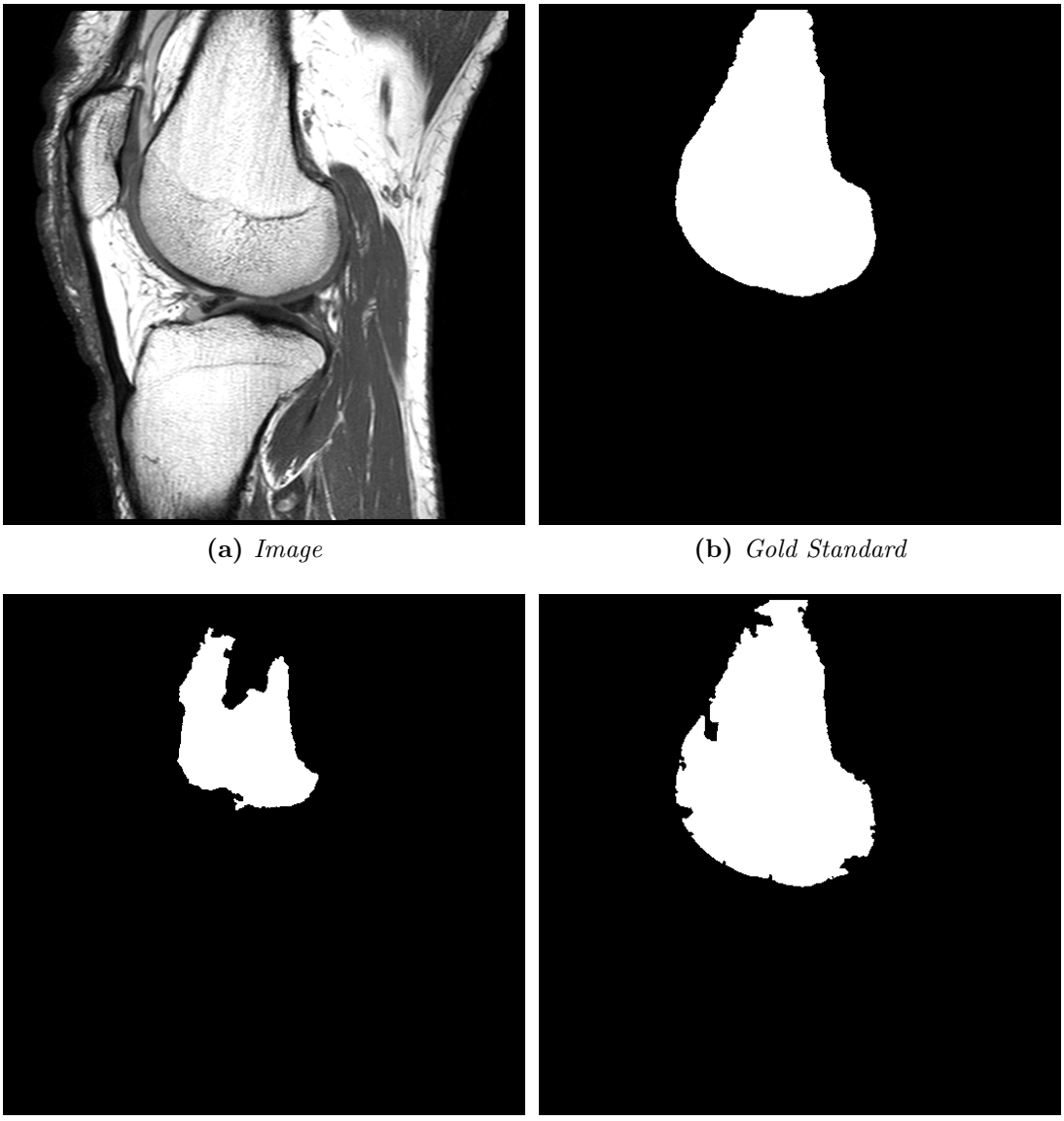
In Table 7.1 we can see the success of the segmentation before and after training (the values for before training correspond simply to the seed region in the IPF, since initially the agent will choose to not include any regions it has no knowledge about), as measured by a number of the evaluation measures defined in Section 2.5.

Reassuringly, all but one image had a better segmentation after training than before. The only exception is training image B, which has a slight drop in DSC (from an already very good 0.999 to a still quite good 0.994). This is due to the inclusion of a small region that should not have been included (as shown by the FPVF going up from 0 to 0.002), due to a region with the same maximum gradient and average intensity being desirable to include in one of the other images.

This highlights a limitation of our method that means it can never achieve a perfect segmentation: if two regions are indistinguishable by these two properties and one should be included and the other should not, the algorithm will always incorrectly classify one. While this case is rare in this sequence of scans, we will see it causes more issues in sequences of scans with poorer contrast like the ones in the next section. It will also make analysis of X-Rays infeasible, as discussed in Section 7.3.

The improvement in the evaluation data-set is very good: the average true positive increases from 0.518 ± 0.086 to 0.998 ± 0.020, with false positive remaining at 0. One example showing this improvement can be seen in Figure 7.2. It is worth noting, however, that the evaluation data-set was picked to be the images where the IPF gives a poorer segmentation in order to highlight improvements resulting from the learning. Had the data-sets been switched around, the improvement would be less noticeable.

Of particular interest is the average DSC we can achieve on an unseen image after training. This is 0.972 ± 0.009, which is comparable to that achieved by current state-of-the-art algorithms presented in Section 3.1. As we will discuss in Section 9.1, our method also offers a number of advantages compared to many of these.



(a) *Image* **(b)** *Gold Standard*

(c) *Before Training* **(d)** *After Training*

Figure 7.2: *Evaluation image E of the Botnar data-set, its gold standard segmentation and its machine segmentations before and after training.*

7.2 Results on SKI10 Data

		Before Training			After Training		
		DSC	TPVF	FPVF	DSC	TPVF	FPVF
Training	M	0.864	0.765	0.002	0.970	0.954	0.003
	N	0.740	0.588	0.000	0.954	0.971	0.019
	P	0.784	0.651	0.003	0.881	0.986	0.067
	Q	0.920	0.885	0.012	0.924	0.970	0.041
	R	0.971	0.949	0.002	0.976	0.970	0.005
Evaluation	S	0.497	0.332	0.002	0.880	0.815	0.012
	T	0.474	0.311	0.000	0.967	0.960	0.008
	U	0.518	0.350	0.001	0.952	0.915	0.002
	V	0.524	0.355	0.000	0.810	0.695	0.005
	W	0.682	0.518	0.001	0.845	0.947	0.083

(a) *Full Results*

	Before Training			After Training		
	DSC	TPVF	FPVF	DSC	TPVF	FPVF
Training Set	0.856	0.768	0.004	0.941	0.970	0.027
Evaluation Set	0.539	0.373	0.001	0.891	0.866	0.022

(b) *Mean Results*

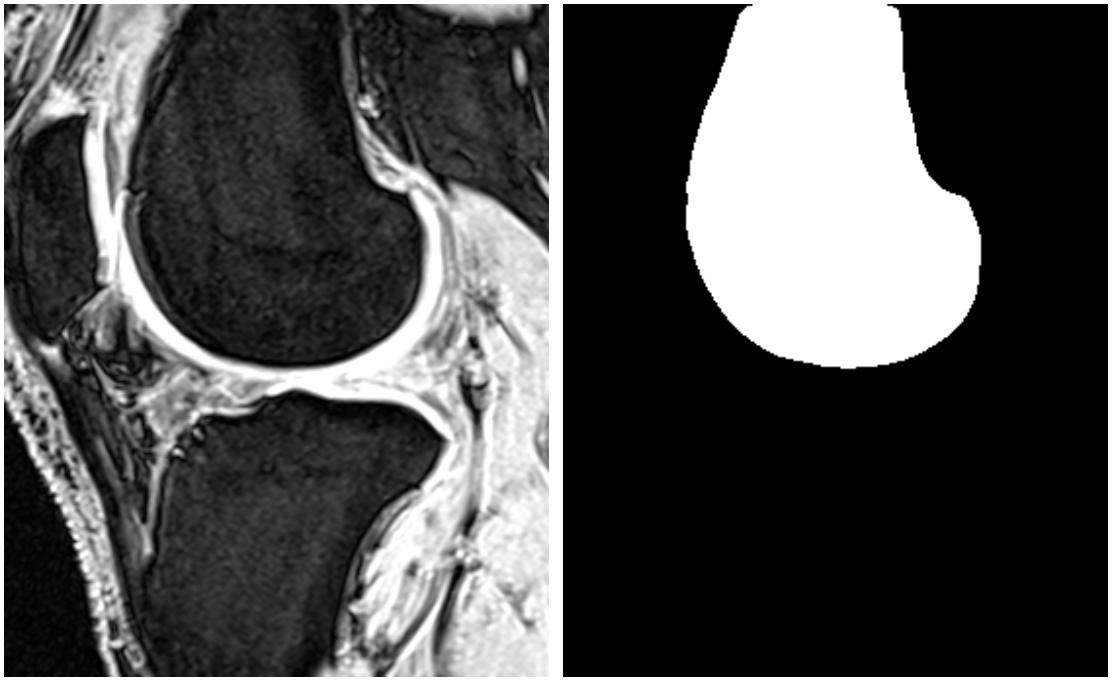
Table 7.3: *Results for the SKI10 data-set (images M-W).*

Having seen our algorithm perform well on some fairly uniform data from a clinical study, we now consider its performance on more demanding data from the SKI10 grand challenge [29], as described in Section 5.2. Once again, we select 10 slices from the data, and split these into two groups of 5, one of which is used for training and the other for evaluation. Our results are presented in Table 7.3.

The data presents a number of challenges that the data from the Botnar Research Centre did not. Firstly, the scans are much less uniform, having probably been taken with different machines or at least very different configurations of the same machine. Secondly, all the images are much lower resolution. Lastly, the contrast on the images is much poorer, making it more difficult to distinguish bone from soft tissue even with a good choice of windowing.

As expected, the results are significantly worse, with the lower contrast leading to a large increase in false positive (see Figure 7.3 for an example of this), even in the training data. This is a limitation of the method on the whole: more training data would not improve this. Nonetheless, training still gives an improvement over just selecting a region in the IPF: the average DSC of a segmentation on the evaluation data-set improves from 0.539 ± 0.082 to an acceptable (although somewhat inferior to current state-of-the-art algorithms) 0.891 ± 0.067.

It is worth noting that because the tuning of our parameters in Section 6 (inclusion criteria, choice of ϵ and training runs) was not done using this data-set, there is also perhaps room for improvement by changing those.



(a) *Image*

(b) *Gold Standard*

(c) *Before Training*

(d) *After Training*

Figure 7.3: *Evaluation image E of the SKI10 data-set, its gold standard segmentation and its machine segmentations before and after training. This shows one of the issues caused by the low contrast of the data-set: the agent decides to include the region in the top right since the change in intensity is not enough to distinguish it from the bone.*

7.3 Investigation Into Other Data Types

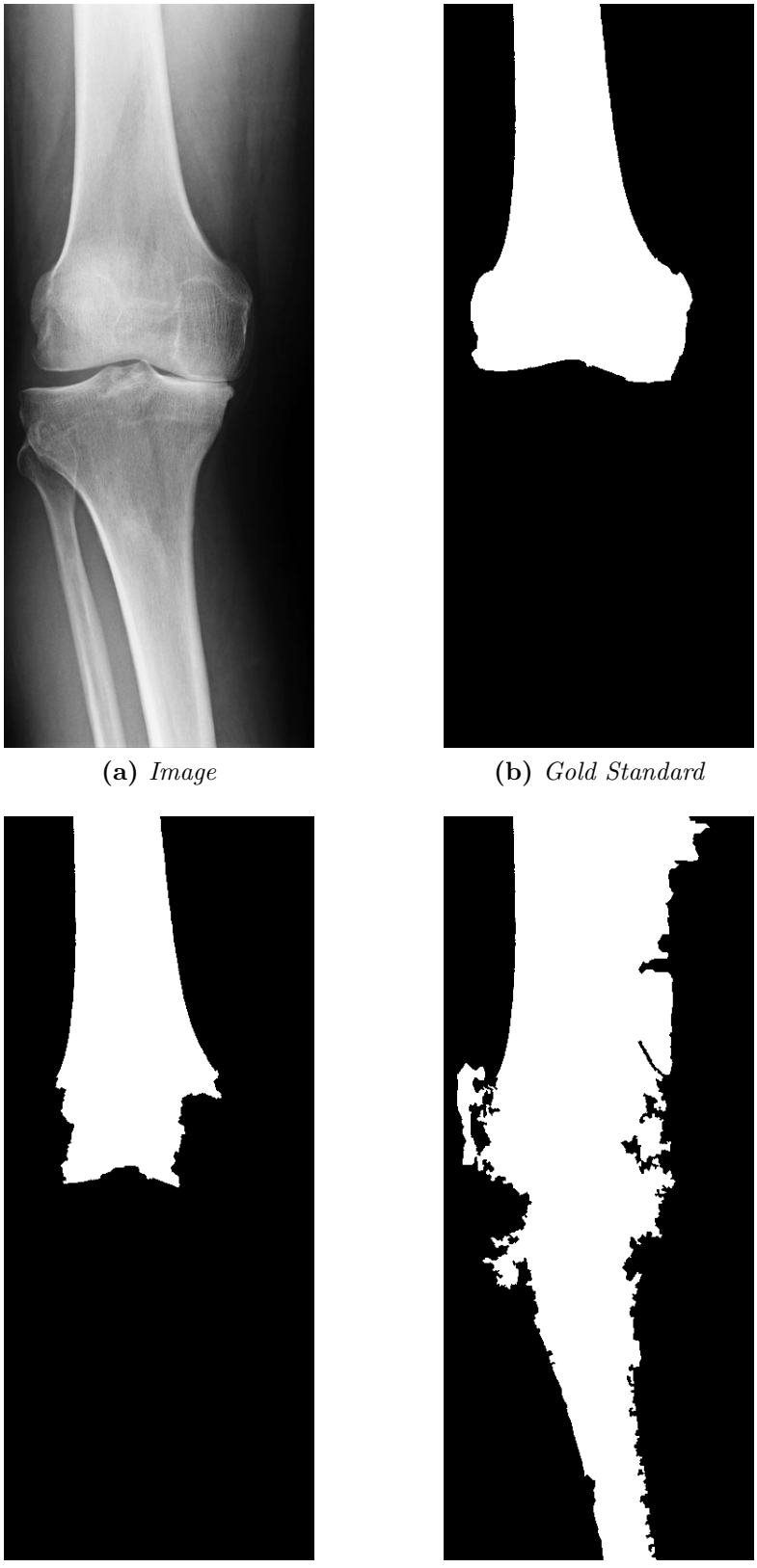
		Before Training			After Training		
		DSC	TPVF	FPVF	DSC	TPVF	FPVF
Training	X	0.911	0.841	0.001	0.980	0.968	0.001
	Y	0.938	0.888	0.002	0.984	0.982	0.005
Evaluation	Z	0.888	0.798	0.000	0.636	0.915	0.265

Table 7.4: *Results for our X-Ray data (images X-Z).*

Having seen our method work successfully on MRI scans, we decided to carry out a small investigation to verify whether or not our technique would carry over easily to other types of scans. For this, we used our algorithm on the three X-Ray images described in Section 5.3. Our results are shown in Table 7.4.

These results show us that our method does not immediately work on X-Ray scans. The problem is that the low contrast inherent to X-Rays does not lend itself to our method as we cannot distinguish regions that are part of the femur or not based solely on average intensity and maximum gradient. So, we will learn to include a region that in another image we would not want to include. This problem can be seen in Figure 7.4. More training data would not address this problem, since there would still be contradictions in the regions that should and should not be included based on their intensity and gradient.

This shows that some modification of our method is needed to make it effective on different types of scans. It is likely that by obtaining more X-Ray data and carrying out an experiment similar to that in Section 6.1, perhaps with some more different choices of criteria, our method could be adapted to work on X-Ray scans. We leave this extension for future work.



(a) *Image* **(b)** *Gold Standard*

(c) *Before Training* **(d)** *After Training*

Figure 7.4: *Results of the evaluation image for our X-Ray data.*

8 Critical Analysis

8.1 Paucity of Data

The aim of our experiments was to show that method works in the sense that we get sensible results even with a small data-set. To give a fair comparison of our method to state-of-the-art algorithms, we would need a much larger training and evaluation data-sets.

This was not feasible because, as discussed in the next section, there are several non-trivial manual steps that make analysing the entirety of a large data-set, like the one from the SKI10 competition, impractical. In future work, removing these manual steps would allow us to analyse our method's performance with a larger data-set.

8.2 Extent of Automation

While our algorithm is largely automatic, it does unfortunately include a few non-trivial manual steps (which is why the sets of training and evaluation data used are relatively small). The most significant manual step is choosing a windowing for each of the images being analysed. This is important for two reasons. Firstly, the windowing needs to make the boundary of the femur distinguishable by providing enough contrast between it and the background. Secondly, it needs to make the greyscale values for the femur and background across images in the training and evaluation data consistent so that the learning can be effective.

Whilst this may appear to be a very severe limitation since it makes the method much less automatic, it is not as much of a problem in a clinical setting as it may initially appear. This is because clinicians will usually order scans from the same machine with the same settings for many patients. So, once a good windowing has been chosen for one such scan, the same windowing will be effective for many others. Nonetheless, it makes analysing the entirety of a large data-set with scans coming from many different machines (like the data from the SKI10 competition) impractical.

Another limitation of our implementation is that it relies on a user-provided seed to start the region growing from. It should be possible to automate the process of choosing this seed by selecting a position in roughly the correct part of the image that is within the correct range of intensity values, in a way similar to [30].

We leave this extension for future work. It is also worth noting that, in practice, when the data is acquired from the same source even just simple constants will often work. For example, in almost all scans from the Botnar Research Centre the point $(200, 200, 10)$ was in the femur (admittedly, while this makes it an acceptable seed, it certainly does not make it the best one).

(This page is intentionally left blank to present the data in the following sub-section next to the text discussing it)

8.3 Brief Comparison to SKI10 Competitors

In this section, we will compare our algorithm's performance on the SKI10 challenge data (as summarised in Section 7.2) to the algorithms that were entered for the competition. While the competition opened in 2010 and most entries were submitted then, it is still open, with some entries being submitted as recently as 2016.

Before we begin this discussion, it is worth noting that all the comparisons made here should be taken with a pinch of salt because (due to the fact that, as discussed in the previous section, each scan has to be manually windowed and a seed point chosen) we only worked on 10 slices rather than the full data-set of 150 scans (each containing about 100 slices, which would have to be processed separately rather than as a 3D volume since the data is anisotropic).

This likely made our results worse than they could be, since the size of the training data-set is significantly smaller. However, it is also possible that the smaller data-set failed to highlight issues in our method. Additionally, the slices we worked on were all close to the middle of the femur where the contrast between bone and tissue is larger, likely improving our results.

Having said this, it is nonetheless interesting to compare our algorithm's performance to others in the SKI10 challenge (a summary of the top scores can be found in Table 8.1).[4] For this, it is necessary to calculate our performance in terms of the AvgD and RMSD measures (as defined in Section 2.5) used for placing participants in the challenge. These were calculated using the code provided as part of the competition and are given in Table 8.2.

Of particular interest is the average of these measures for the unseen evaluation data-set after training, which are 0.79 ± 0.99mm and 2.62 ± 9.61mm, respectively (this would place us around 15th out of the 21 participants in the competition). If we were to exclude the problematic image E, which has particularly poor contrast leading to a large false positive (as shown in Figure 7.3) they would improve to 0.38 ± 0.39mm and 1.31 ± 1.38mm, which would place us around 5th. Unfortunately, the large variance means our results are not very statistically significant.

It is interesting to note that the spread between our AvgD scores and our RMSD scores is much larger than any of the submissions in the competition. This is due to the fact that when our algorithm performs poorly (as in Figure 7.3), it overselects very far in one area and performs acceptably in the rest. This is punished more severely by the RMSD metric than the AvgD one. It is possible that a further post-processing step could remove these regions where we have overselected. We discuss this in Section 9.2.

While our method does not outperform most of the entries to the competition, it still offers a number of advantages. We discuss some of these in the next section.

[4]Full results, along with references to corresponding papers where these are available, can be found at `http://www.ski10.org/results.php`

Team	AvgD (mm)	RMSD (mm)
Imorphics	0.42	0.74
ZIB	0.57	0.88
UPMC_IBML	0.63	1.05
SNU_SPL	0.70	1.09
Biomediq	0.77	1.45
UPMC_IBML	0.66	1.06
shan_unc	0.76	1.22
AMC_MIRL	0.67	1.13
UIiibiKnee	0.72	1.17
shan_unc	0.78	1.25

Table 8.1: *Mean scores for femur segmentation for the top 10 teams in the SKI10 challenge.*

		Before Training		After Training	
		AvgD (mm)	RMSD (mm)	AvgD (mm)	RMSD (mm)
Training	M	0.706	2.454	0.019	0.130
	N	1.944	4.739	0.323	2.014
	P	2.174	6.013	2.256	7.854
	Q	0.213	1.053	0.349	1.855
	R	0.023	0.162	0.016	0.117
Evaluation	S	6.615	10.903	0.932	2.984
	T	6.154	10.446	0.035	0.239
	U	7.993	14.151	0.170	0.976
	V	4.422	7.962	0.375	1.033
	W	2.344	5.099	2.460	7.857

(a) *Full Results*

	Before Training		After Training	
	AvgD (mm)	RMSD (mm)	AvgD (mm)	RMSD (mm)
Training Set	1.012	2.884	0.593	2.394
Evaluation Set	5.506	9.712	0.794	2.618

(b) *Mean Results*

Table 8.2: *Our results for the SKI10 data-set in terms of AvgD and RMSD evaluation measures used as a standard in the challenge.*

9 Conclusion

9.1 Discussion of Main Advantages

One advantage of our algorithm is that it has a very short run-time if we already have an IPF: on a standard laptop (Intel i7-4980HQ 2.80GHz processor, 16GB RAM) our fairly unoptimised Scala implementation takes a couple of hundred milliseconds to process a single slice, so would likely take about 10 seconds to process a whole scan (a more optimised implementation in a lower-level language could probably take half this time). This is much quicker than every entry in the SKI10 competition that reported a running time (these ranged from about 30 seconds to 40 minutes per scan). Even if we include the time needed to build an IPF (about 15 minutes), this is still faster than most entries.

Additionally, our algorithm needs significantly less training data than traditional machine learning approaches, as evidenced by the fact that competitive results were obtained on the SKI10 data using just five slices of the gold standards. In contrast, the leading entry in the competition used 80 scans [31]. This is crucial because clinician time is expensive, so being able to perform well with only a limited amount of hand-contoured data is very useful.

Finally, there are differences in knee anatomy between patients based on race and gender [32]. This can lead to algorithms that use techniques based on probabilistic maps or deformable models (most of the entries in the competition) exhibiting bias based on the data they have been trained on. However, this should not be an issue for our algorithm, since we rely solely on the intensity values of the scan, which should not vary significantly across patients.

9.2 Future Work

One potential area for future improvement in our method would be in some further post-processing steps (in addition to morphological closing) to improve the result. Most notably, anti-aliasing to smooth the outline of the result would probably be desirable.

Another post-processing step could attempt to identify places where our selection "bled" into neighbouring regions of similar intensity (as occurred in Figure 7.3) and remove these. This identification could be based on regions that are poorly connected to the rest of the selection, or perhaps on statistical information about knee shape.

Another potential improvement would be using information beyond the intensity and gradient in the agent's decision to include or exclude regions, which would perhaps improve results on images with poorer contrast.

One possible idea for this would be to add a measure of how closely connected our region is to those that we have already included, the reasoning being that we are more likely to want to include regions that are closely connected to those we already present.

However, this has the downside that we will need to consider some regions many times (rather than at most once) since we may want to later include a region that

we previously excluded based on other nearby regions now being included. This will likely greatly increase the run-time of the algorithm.

Another area our algorithm could perhaps be improved is by changing how the IPF it relies on is constructed. In particular, the IPF construction depends on the data being smoothed by several passes of an anisotropic diffusion filter. Changing the number of passes will change how coarse or fine the regions in the first branch layer are and tuning this might improve results.

As previously discussed, it might also be helpful to grow our selection using layers higher up in the IPF rather than always using the first branch layer. This would certainly allow for faster segmentation since it would reduce the number of regions we need consider. However, this could be at the expense of segmentation accuracy since the regions being added will be coarser. To avoid this issue, it could be beneficial to give the agent a third option (in addition to including/excluding a region): to split up a region and consider its children separately.

A further extension of our method could also consider varying the value of ε based on some heuristic instead of having it fixed. In particular, it might be beneficial to reduce it as more training has occurred (this could be measured by, for instance, the number of distinct regions that we have encountered).

A limitation of our evaluation is that we chose to perform it only on 2D slices of images, rather than on 3D images (though it is worth noting that the implementation fully supports 3D images). This is primarily because we did not have access to any isotropic data and some concepts our method relies on do not readily translate to anisotropic images.

In particular, gradients become significantly more complicated to calculate and reason about and the concept of voxels being adjacent is meaningless if the slices are too far apart. In addition, constructing the required IPFs for 3D images is a more time-consuming process (with each IPF taking around 15 minutes to compute on a modern laptop).

Thus, we leave evaluation of our algorithm on 3D images for future work. It is expected that, similarly to other algorithms [16], our performance will be good in the middle layers, where the contrast between the femur and background is good but less so in the outside layers.

Another extension to the evaluation could include attempting to segment different features such as, for example, organs in an abdominal scan. It is likely that results will be worse here, since organs have much lower HU values than bone, so they are harder to distinguish from the soft tissue around them.

Having performed a more thorough evaluation of our algorithm, we will submit a paper for publication in the SPIE Medical Imaging Conference.

10 Acknowledgements

I would like thank my supervisor, Dr. Irina Voiculescu, for all her help throughout this year. We would also like to thank the Botnar Research Centre and the organisers of the SKI10 grand challenge [29] for providing the data that was used for training and evaluation of our algorithm.

Finally, we would like to thank the developers of the ImageJ libraries [23], which were used in this project for reading and writing images, the developers of the MorphoLibJ library [26] which was used for morphological closing of results, and the developers of Google Guava,[5] which was used for some file input stream operation.

[5] `https://github.com/google/guava`

Bibliography

[1] Z. Wu and J. M. Sullivan, "Multiple material marching cubes algorithm," *International Journal for Numerical Methods in Engineering*, vol. 58, pp. 189–207, sep 2003.

[2] L. Mattei, P. Pellegrino, M. Calo, A. Bistolfi, and F. Castoldi, "Patient specific instrumentation in total knee arthroplasty: a state of the art.," *Annals of Translational Medicine*, vol. 4, no. 7, p. 126, 2016.

[3] E. Thienpont, J. Bellemans, H. Delport, P. Van Overschelde, B. Stuyts, K. Brabants, and J. Victor, "Patient-specific instruments: Industry's innovation with a surgeon's interest," *Knee Surgery, Sports Traumatology, Arthroscopy*, vol. 21, no. 10, pp. 2227–2233, 2013.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, United States: MIT Press, 1998.

[5] S. Singh and D. Bertsekas, "Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems," in *Proceedings of the 9th International Conference on Neural Information Processing Systems*, (Denver, Colorado, United States), pp. 974–980, MIT Press, 1997.

[6] A. Barto and R. H. Crites, "Improving elevator performance using reinforcement learning," in *Proceedings of the 8th International Conference on Neural Information Processing Systems*, (Denver, Colorado, United States), pp. 1017–1023, MIT Press, 1995.

[7] G. Tesauro, "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, 1994.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[9] S. M. Golodetz, *Zipping and Unzipping: The Use of Image Partition Forests in the Analysis of Abdominal CT Scans*. PhD thesis, University of Oxford, 2010.

[10] J. Weickert, *Anisotropic Diffusion in Image Processing*. Stuttgart, Germany: BG Teubner Verlag, 1996.

[11] J. Roerdink and a. Meijster, "The Watershed Transform: Definitions, Algorithms and Parallelization Strategies," *Fundamenta Informaticae*, vol. 41, no. 1-2, pp. 187–228, 2000.

[12] V. Yeghiazaryan and I. Voiculescu, "An Overview of Current Evaluation Methods Used in Medical Image Segmentation (RR-15-08)," tech. rep., University of Oxford, Oxford, United Kingdom, nov 2015.

[13] V. Yeghiazaryan and I. Voiculescu, "Boundary Overlap for Medical Image Segmentation Evaluation," in *Proceedings of SPIE Medical Imaging*, (Orlando, Florida, United States), Society of Photooptical Instrumentation Engineers, 2017.

[14] Y. J. Zhang, "A review of recent evaluation methods for image segmentation," in *Proceedings of the Sixth International Symposium on Signal Processing and its Applications*, vol. 1, (Kuala Lumpur, Malaysia), pp. 148–151, IEEE, 2001.

[15] Y. J. Zhang, "A survey on evaluation methods for image segmentation," *Pattern Recognition*, vol. 29, no. 8, pp. 1335–1346, 1996.

[16] M. Lim, "Structure-enhanced local phase filtering using L0 gradient minimization for efficient semiautomated knee magnetic resonance imaging segmentation," *Journal of Medical Imaging*, vol. 3, no. 4, 2016.

[17] J. Fripp, S. Crozier, S. K. Warfield, and S. Ourselin, "Automatic segmentation and quantitative analysis of the articular cartilages from magnetic resonance images of the knee," *IEEE Transactions on Medical Imaging*, vol. 29, pp. 55–64, mar 2010.

[18] J.-G. Lee, S. Gumus, C. H. Moon, C. K. Kwoh, and K. T. Bae, "Fully automated segmentation of cartilage from the MR images of knee using a multi-atlas and local structural analysis method.," *Medical physics*, vol. 41, no. 9, p. 092303, 2014.

[19] P. Dodin, J. Martel-Pelletier, J. P. Pelletier, and F. Abram, "A fully automated human knee 3D MRI bone segmentation using the ray casting technique," *Medical and Biological Engineering and Computing*, vol. 49, no. 12, pp. 1413–1424, 2011.

[20] F. Sahba, H. R. Tizhoosh, and M. M. a. Salama, "for Medical Image Segmentation," in *Proceedings of the 2006 International Joint Conference on Neural Networks*, (Vancouver, BC, Canada), pp. 1238–1244, IEEE Computer Society Press, 2006.

[21] J. Wang, "Energy Efficient Backoff Hierarchical Clustering Algorithms for Multi-Hop Wireless Sensor Networks," *Journal of Computer Science and Technology*, vol. 26, no. 2008, pp. 283–291, 2011.

[22] V. Martin, M. Thonnat, and N. Maillot, "A learning approach for adaptive image segmentation," in *Proc. of Fourth IEEE International Conference on Computer Vision Systems*, vol. 40, pp. 431–454, IEEE Computer Society Press, 2006.

[23] M. D. Abràmoff, P. J. Magalhães, and S. J. Ram, "Image processing with ImageJ," *Biophotonics International*, vol. 11, no. 7, pp. 36–41, 2004.

[24] S. Golodetz, I. Voiculescu, and S. Cameron, "Simpler Editing of Spatially-Connected Graph Hierarchies using Zipping Algorithms (CS-RR-15-06)," tech. rep., University of Oxford, Oxford, United Kingdom, 2017.

[25] R. B. Fisher and K. Koryllos, "Interactive Textbooks; Embedding Image Processing Operator Demonstrations in Text," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 12, pp. 1095–1123, dec 1998.

[26] D. Legland, I. Arganda-Carreras, and P. Andrey, "MorphoLibJ: integrated library and plugins for mathematical morphology with ImageJ," *Bioinformatics*, vol. 32, no. 22, pp. 3532–3534, 2016.

[27] M. Marčan, E. C. Pegg, H. G. Pandit, and I. Voiculescu, "Calibrating the segmentation of the knee in order to expedite patient-specific instruments," 2017.

[28] M. Alinejad, *Artificial Anterior Cruciate Ligament Reconstruction*. PhD thesis, University of Oxford, 2014.

[29] T. Heimann, B. J. Morrison, M. A. Styner, M. Niethammer, and S. K. Warfield, "Segmentation of knee images: A grand challenge," in *Proc. MICCAI Workshop on Medical Image Analysis for the Clinic*, (Beijing, China), pp. 207–214, Springer, 2010.

[30] V. Yeghiazaryan and I. D. Voiculescu, "Automated 3D renal segmentation based on image partitioning," in *Proceedings of SPIE Medical Imaging*, (San Diego, California, United States), International Society for Optics and Photonics, mar 2016.

[31] G. Vincent, C. Wolstenholme, I. Scott, and M. Bowes, "Fully Automatic Segmentation of the Knee Joint using Active Appearance Models," 2010.

[32] B. Yue, K. M. Varadarajan, S. Ai, T. Tang, H. E. Rubash, and G. Li, "Differences of Knee Anthropometry Between Chinese and White Men and Women," *Journal of Arthroplasty*, vol. 26, no. 1, pp. 124–130, 2011.

A Data Parameters

In this appendix we give the details of which slices we used from each of the datasets, along with what seed point and windowings we chose for each one.

A.1 Botnar Research Centre Data

Image		Scan #	Slice	Seed	Window C	Window W
Training	A	1	10	(200, 200)	1214	2408
	B	10	10	(200, 200)	1050	2050
	C	11	9	(200, 200)	1100	2200
	D	12	10	(200, 200)	1050	2050
	E	13	10	(200, 200)	1050	2050
Evaluation	F	2	6	(200, 150)	1325	1604
	G	3	11	(200, 200)	1126	2231
	H	5	9	(200, 200)	1275	2500
	I	6	11	(200, 200)	1200	2400
	J	7	11	(200, 150)	1050	2050

Table A.1: *Details of the images used from the Botnar data and parameters used.*

A.2 SKI10 Grand Challenge Data

Image		Scan #	Slice	Seed	Window C	Window W
Training	M	4	66	(100, 100)	416	781
	N	15	70	(150, 100)	251	492
	P	41	59	(100, 140)	79	150
	Q	42	61	(100, 100)	492	985
	R	50	64	(100, 100)	64	128
Evaluation	S	39	69	(100, 100)	64	127
	T	46	69	(100, 100)	870	1493
	U	52	83	(100, 100)	162	321
	V	54	75	(100, 100)	598	1012
	W	60	76	(100, 100)	415	830

Table A.2: *Details of the images used from the SKI10 data and parameters used.*

B Code

This appendix includes all the code written for the project, with each subsection corresponding to a different package (except the first, which corresponds to the SBT build file).[6] In total, there are just over 1100 lines of Scala code.

B.1 build.sbt

```
lazy val commonSettings = Seq(
  organization := "org.edoardo",
  version := "1.0.0",
  scalaVersion := "2.12.1"
)

resolvers +=
    "ImageJ Releases" at
 ↪  "http://maven.imagej.net/content/repositories/releases/"

lazy val root: Project = (project in file(".")).
  settings(commonSettings: _*).
  settings(
    name := "RL Segmentation",
    scalaSource in Compile := baseDirectory.value / "src",
    maxErrors := 20,
    pollInterval := 1000,
    scalacOptions += "-deprecation",
    fork := true,
    javaOptions += "-Xmx4G",
    libraryDependencies +=
      "log4j" % "log4j" % "1.2.15" excludeAll(
        ExclusionRule(organization = "com.sun.jdmk"),
        ExclusionRule(organization = "com.sun.jmx"),
        ExclusionRule(organization = "javax.jms")
      ),
      libraryDependencies += "net.imglib2" % "imglib2" % "3.2.1",
      libraryDependencies += "net.imglib2" % "imglib2-algorithm" % "0.6.2",
      libraryDependencies += "net.imglib2" % "imglib2-ij" % "2.0.0-beta-35",
      libraryDependencies += "fr.inra.ijpb" % "MorphoLibJ_" % "1.3.2",
      libraryDependencies += "com.google.guava" % "guava" % "21.0"
  )
```

B.2 rl

B.2.1 Policy.scala

```
package org.edoardo.rl

import scala.collection.concurrent.TrieMap
import scala.util.Random

/**
  * Abstract class to represent a state.
  * @tparam T class of the actions that are performed from this state
```

[6] An electronic copy can also be found at https://github.com/EdoDodo/rl-segmentation

```
9     */
10  abstract class State[T <: Action] {
11    def getAvailableActions: List[T]
12  }
13
14  /**
15    * Trait to represent an action.
16    */
17  trait Action
18
19  /**
20    * Class to represent a policy.
21    * @tparam A the type of actions the agent can perform under this policy
22    * @tparam S the type of states the agent can encounter under this policy
23    */
24  class Policy[A <: Action, S <: State[A]] {
25    /**
26      * Create a mapping for storing estimated values.
27      */
28    var values: TrieMap[S, TrieMap[A, (BigDecimal, Long)]] = TrieMap()
29
30    /**
31      * Return a random action with probability 1/epsilonReciprocal or the greedy
    ↪  action otherwise.
32      * @param state the state to consider
33      * @param epsilonReciprocal the reciprocal of epsilon that we want
34      * @return the action chosen
35      */
36    def epsilonSoft(state: S, epsilonReciprocal: Int): A =
37      if (Random.nextInt(epsilonReciprocal) == 0) randomPlay(state)
38      else greedyPlay(state)
39
40    /**
41      * Choose a random action to play.
42      * @return a random action
43      */
44    def randomPlay(state: S): A = Random.shuffle(state.getAvailableActions).head
45
46    /**
47      * Choose the greedy action to play.
48      * @param state the state to consider
49      * @return the current greedy action from the given state
50      */
51    def greedyPlay(state: S): A = {
52      if (!haveEncountered(state)) state.getAvailableActions.head
53      else values(state).maxBy(_._2._1)._1
54    }
55
56    /**
57      * Update the policy by adding an observed reward for a given play.
58      * @param state the state the play was made from
59      * @param action the action performed
60      * @param reward the reward obtained
61      */
62    def update(state: S, action: A, reward: Double): Unit = {
```

```
63      var map: TrieMap[A, (BigDecimal, Long)] = values.getOrElseUpdate(state, {
64        val result: TrieMap[A, (BigDecimal, Long)] = new TrieMap()
65        for (a <- state.getAvailableActions)
66          result += ((a, (BigDecimal(0.0), 0L)))
67        result
68      })
69      val old: (BigDecimal, Long) = map(action)
70      map += ((action, (((old._1 * old._2) + reward) / (old._2 + 1), old._2 + 1)))
71      values += ((state, map))
72    }
73
74    /**
75      * Check if a state has been encountered before.
76      * @param state the state to check
77      * @return whether or not we have seen this state  before
78      */
79    def haveEncountered(state: S): Boolean = {
80      if (values.get(state).isDefined) true
81      else false
82    }
83
84    /**
85      * Clear the policy, forgetting everything learnt.
86      */
87    def clear(): Unit = {
88      values = TrieMap()
89    }
90  }
```

B.3 parser

B.3.1 Parser.scala

```
1   package org.edoardo.parser
2
3   import java.io.InputStream
4
5   /**
6     * Abstract class containing a few helper methods used in classes that
    ↪  implement Parsers.
7     */
8   abstract class Parser {
9     /**
10      * Read a line and check it has a certain value (throwing an exception if
    ↪  this is not the case).
11      * @param value the value the line should have
12      * @param in the input stream to read the line from
13      */
14    def checkLineIs(value: String)(implicit in: InputStream): Unit = {
15      val line: String = readLine
16      if (line != value)
17        throw new IllegalArgumentException("File is not a valid: Expected " +
    ↪  value + " but got " + line)
18    }
19
20    /**
```

```
21      * Read a line from an input stream
22      * @param in the input stream to read from
23      * @return the line read
24      */
25    def readLine(implicit in: InputStream): String = {
26      var out = ""
27      var b: Int = in.read
28      while (b != 0xA) {
29        out += b.toChar
30        b = in.read
31      }
32      out
33    }
34  }
```

B.3.2 IPF.scala

```
1  package org.edoardo.parser
2
3  import java.io.{BufferedInputStream, FileInputStream, InputStream}
4
5  import scala.collection.mutable
6
7  /**
8    * Describes a pixel (leaf node) of the IPF.
9    */
10 case class PixelProperties(baseValue: Int, gradientMagnitude: Int, greyValue:
   ↪  Int, parent: Int)
11
12 /**
13   * Describes a branch node of the IPF.
14   */
15 case class Node(centroid: (Float, Float, Float), maxGrey: Int, meanGrey: Float,
   ↪  minGrey: Int, xMin: Int, yMin: Int,
16         zMin: Int, xMax: Int, yMax: Int, zMax: Int, voxelCount: Int, children:
   ↪  List[Int], parent: Int)
17
18 /**
19   * Describes the leaf layer of an IPF.
20   */
21 case class LeafLayer(sizeX: Int, sizeY: Int, sizeZ: Int, pixelInfo:
   ↪  Array[Array[Array[PixelProperties]]],
22             regionToPixels: mutable.Map[Int, List[(Int, Int, Int)]])
23
24 /**
25   * Describes a branch layer of an IPF.
26   */
27 case class BranchLayer(nodes: mutable.Map[Int, Node], edges: mutable.Map[Int,
   ↪  List[(Int, Int)]])
28
29 /**
30   * Describes an IPF and contains helper methods for accessing useful properties
   ↪   of it.
31   * @param width the width of the image
32   * @param height the height of the image
33   * @param depth the depth of the image
```

```
34    * @param leafLayer the leaf layer of the IPF
35    * @param branchLayers the branch layers of the IPF
36    */
37  case class VolumeIPF(width: Int, height: Int, depth: Int, leafLayer: LeafLayer,
    ↪  branchLayers: List[BranchLayer]) {
38    /**
39      * Get all pixels in a region, which is assumed to be in the first branch
    ↪   layer.
40      * @param region the identifier for the region
41      * @return a list of pixels in the region
42      */
43    def getRegionPixels(region: Int): List[(Int, Int, Int)] = {
44      leafLayer.regionToPixels(region)
45    }
46
47    /**
48      * Get all pixels in a region, in the given layer
49      * @param layer the layer of the IPF the region is in
50      * @param region the identifier for the region
51      * @return a list of pixels in the region
52      */
53    def getRegionPixels(layer: Int, region: Int): List[(Int, Int, Int)] = {
54      var regions: List[Int] = List(region)
55      for (i <- 0 until (layer - 1)) {
56        var newRegions: List[Int] = List()
57        for (region <- regions)
58          newRegions = newRegions ++ branchLayers(branchLayers.length - layer +
    ↪   i).nodes(region).children
59        regions = newRegions
60      }
61      regions.flatMap(region => getRegionPixels(region))
62    }
63
64    /**
65      * For a given seed point and layer, find the corresponding region and then
    ↪   return a list of all layer 1 children
66      * of this seed region (used to initialise a selection).
67      * @param layer the layer to look in
68      * @param x the x coordinate of the seed
69      * @param y the y coordinate of the seed
70      * @param z the z coordinate of the seed
71      * @return a list of layer 1 children of the selected seed reigon
72      */
73    def getRegionsInLayer(layer: Int, x: Int, y: Int, z: Int): List[Int] = {
74      if (layer == 1) return List(leafLayer.pixelInfo(x)(y)(z).parent)
75      var regions: List[Int] = getRegionInLayer(layer, x, y, z).children
76      for (i <- branchLayers.length - layer + 1 until branchLayers.length - 1)
77        regions = regions.flatMap(r => branchLayers(i).nodes(r).children)
78      regions
79    }
80
81    /**
82      * Find which region contains a point in a given layer of the IPF.
83      * @param layer the layer to look in
84      * @param x the x coordinate to look at
```

```
85      * @param y the y coordinate to look at
86      * @param z the z coordinate to look at
87      * @return the identifier corresponding to the region at (x, y, z) in the
    ↪  given layer
88      */
89    def getRegionInLayer(layer: Int, x: Int, y: Int, z: Int): Node = {
90      var region: Node =
    ↪  branchLayers.last.nodes(leafLayer.pixelInfo(x)(y)(z).parent)
91      for (i <- 2 to layer)
92        region = branchLayers(branchLayers.length - i).nodes(region.parent)
93      region
94    }
95
96    /**
97      * Find all neighbours of a region (assumed to be in the last branch layer).
98      * @param region the region identifier
99      * @return a list of region identifiers of neighbouring regions
100     */
101   def getNeighbours(region: Int): List[Int] = {
102     branchLayers.last.edges(region).map(edge => edge._1)
103   }
104
105   /**
106     * Find the z coordinate of a given region (assumed to be in the last branch
    ↪  layer).
107     * Note this assumes the IPF is an axial one (ie. has been made with separate
    ↪  forests for each X-Y slice).
108     * @param region the region identifier
109     * @return the z coordinate of the region
110     */
111   def getZ(region: Int): Int = {
112     getRegionPixels(region).head._3
113   }
114 }
115
116 /**
117   * Implements a parser for an IPF.
118   */
119 object IPF extends Parser {
120
121   /**
122     * Read an IPF from a file.
123     * @param fileName the file to read from
124     * @return the IPF in the file
125     */
126   def loadFromFile(fileName: String): VolumeIPF = {
127     implicit val in = new BufferedInputStream(new FileInputStream(fileName))
128     checkLineIs("VolumeIPF")
129     checkLineIs("{")
130
131     val sizeLine: Array[String] = readLine.drop(1).dropRight(1).split(", ")
132
133     val leafLayer: LeafLayer = readLeafLayerSection()
134
135     val topLayer: Int = readLine.toInt
```

```
136     var branchLayers: List[BranchLayer] = List()
137     for (i <- 1 to topLayer)
138       branchLayers ::= readBranchLayer()
139
140     checkLineIs("}")
141
142     VolumeIPF(sizeLine(0).toInt, sizeLine(1).toInt, sizeLine(2).toInt,
    ↪  leafLayer, branchLayers)
143   }
144
145   private def readBranchLayer()(implicit in: InputStream): BranchLayer = {
146     checkLineIs("ImageBranchLayer")
147     checkLineIs("{")
148
149     var stop = false
150     val nodes: mutable.Map[Int, Node] = mutable.Map.empty
151     while (!stop) {
152       val line: String = readLine
153       if (line == "|") stop = true
154       else {
155         val node: Int = line.toInt
156         val properties: Array[String] =
    ↪  readLine.drop(2).dropRight(2).split("\\|")
157         val centroid: Array[String] =
    ↪  properties(0).drop(1).dropRight(1).split(",")
158         checkLineIs("{")
159         var stopInner = false
160         var children: List[Int] = List()
161         while (!stopInner) {
162           val innerLine: String = readLine
163           if (innerLine == "}") stopInner = true
164           else children ::= innerLine.toInt
165         }
166         val parent: Int = readLine.toInt
167         nodes.put(node, Node((centroid(0).toFloat, centroid(1).toFloat,
    ↪  centroid(2).toFloat),
168           properties(1).toInt, properties(2).toFloat, properties(3).toInt,
    ↪  properties(4).toInt,
169           properties(5).toInt, properties(6).toInt, properties(7).toInt,
    ↪  properties(8).toInt,
170           properties(9).toInt, properties(10).toInt, children, parent))
171       }
172     }
173
174     val edges: mutable.Map[Int, List[(Int, Int)]] = mutable.Map.empty
175     stop = false
176     while (!stop) {
177       val line: String = readLine
178       if (line == "}") stop = true
179       else {
180         val split: Array[String] = line.drop(1).dropRight(1).split(", ")
181         val fromAndTo: Array[String] = split(0).drop(1).dropRight(1).split(" ")
182         edges.put(fromAndTo(0).toInt,
183           (fromAndTo(1).toInt, split(1).toInt) ::
    ↪  edges.getOrElseUpdate(fromAndTo(0).toInt, List()))
```

```
184         edges.put(fromAndTo(1).toInt,
185           (fromAndTo(0).toInt, split(1).toInt) ::
    ↪  edges.getOrElseUpdate(fromAndTo(1).toInt, List()))
186       }
187     }
188
189     BranchLayer(nodes, edges)
190   }
191
192   private def readLeafLayerSection()(implicit in: InputStream): LeafLayer = {
193     checkLineIs("ImageLeafLayer")
194     checkLineIs("{")
195     val sizeX: Int = readLine.trim.split(" = ")(1).toInt
196     val sizeY: Int = readLine.trim.split(" = ")(1).toInt
197     val sizeZ: Int = readLine.trim.split(" = ")(1).toInt
198     checkLineIs("|")
199
200     var stop = false
201     val pixelProperties: Array[Array[Array[PixelProperties]]] =
    ↪  Array.ofDim[PixelProperties](sizeX, sizeY, sizeZ)
202     val parentRegionToPixels: mutable.Map[Int, List[(Int, Int, Int)]] =
    ↪  mutable.Map.empty
203     var x = 0
204     var y = 0
205     var z = 0
206     while (!stop) {
207       val line: String = readLine
208       if (line == "}") stop = true
209       else {
210         val properties: Array[String] = line.drop(1).dropRight(1).split("\\|")
211         val parent: Int = readLine.toInt
212         pixelProperties(x)(y).update(z,
213           PixelProperties(properties(0).toInt, properties(1).toInt,
    ↪  properties(2).toInt, parent))
214         parentRegionToPixels.put(parent, (x, y, z) ::
    ↪  parentRegionToPixels.getOrElseUpdate(parent, List()))
215         x += 1
216         if (x == sizeX) {
217           x = 0
218           y += 1
219           if (y == sizeY) {
220             y = 0
221             z += 1
222           }
223         }
224       }
225     }
226
227     LeafLayer(sizeX, sizeY, sizeZ, pixelProperties, parentRegionToPixels)
228   }
229 }
```

B.3.3 MFS.scala

```
1 package org.edoardo.parser
2
```

```scala
import java.io.{BufferedInputStream, FileInputStream}

import org.edoardo.segmentation.SegmentationResult

/**
  * Implements a parser for a Multi Feature Set (MFS), which describes selections in an IPF. Note we make the simplifying
  * assumption that there is only one feature in the image, and it is called "Level Set 0"
  */
object MFS extends Parser {

  /**
    * Load a MFS from a file.
    * @param fileName the file to load the MFS from
    * @param ipf the IPF corresponding to the MFS we are loading
    * @return a segmentation result containing the region described by the MFS
    */
  def loadFromFile(fileName: String, ipf: VolumeIPF): SegmentationResult = {
    implicit val in = new BufferedInputStream(new FileInputStream(fileName))
    checkLineIs("MFS Text 0")
    checkLineIs("{")
    checkLineIs("Level Set 0")
    checkLineIs("{")

    var done = false
    var regions: List[(Int, Int)] = List()
    while (!done) {
      val line: String = readLine
      if (line == "}")
        done = true
      else {
        val splitLine: Array[String] = line.split(",")
        regions ::= (splitLine(0).drop(1).toInt, splitLine(1).dropRight(1).toInt)
      }
    }

    checkLineIs("}")

    val result: Array[Array[Array[Boolean]]] = Array.fill[Boolean](ipf.width, ipf.height, ipf.depth)(false)
    for ((x, y, z) <- regions.flatMap(region => ipf.getRegionPixels(region._1, region._2)))
      result(x)(y)(z) = true
    new SegmentationResult(result)
  }
}
```

B.4 image

B.4.1 Raw.scala

```scala
package org.edoardo.image

import java.io.{BufferedInputStream, File, FileInputStream}
```

```
4
5  import com.google.common.io.LittleEndianDataInputStream
6  import ij.process.ImageProcessor
7  import ij.{IJ, ImagePlus}
8
9  /**
10   * Provides a few methods for working with Raw image files.
11   */
12 object Raw {
13   sealed trait ByteType
14   case object UCHAR extends ByteType
15   case object USHORT extends ByteType
16
17   /**
18     * Read the image described by the given metadata file.
19     * @param name the name of the metadata file for a Raw image
20     * @param layer the layer to read (or -1 to read all layers)
21     * @return the image read
22     */
23   def openMetadata(name: String, layer: Integer): ImagePlus = {
24     val (width, height, depth, dataFile, byteType) = readMetadata(name)
25     assert(byteType == USHORT)
26     implicit val in = new LittleEndianDataInputStream(new
   ↪  BufferedInputStream(new FileInputStream(dataFile)))
27     val image: ImagePlus = IJ.createImage(name, "16-bit", width, height, if
   ↪  (layer == -1) depth else 1)
28     for (z <- 0 until depth) {
29         for (y <- 0 until height) {
30           for (x <- 0 until width) {
31             val intensity: Short = in.readShort()
32             if (layer == -1 || z == layer)
33               image.getProcessor.putPixel(x, y, intensity)
34           }
35         }
36       if (layer == -1)
37         image.setZ(image.getZ + 1)
38     }
39     image
40   }
41
42   /**
43     * Open a label image (contains 1 for marked voxels, 0 for unmarked ones).
44     * @param name the name of the label image to open
45     * @return an image containing white for marked voxels and black for unmarked
   ↪   ones
46     */
47   def openLabels(name: String): ImagePlus = {
48     val (width, height, depth, dataFile, byteType) = readMetadata(name)
49     implicit val in = new LittleEndianDataInputStream(new
   ↪  BufferedInputStream(new FileInputStream(
50       if (new File(name).getParent == null) dataFile else new
   ↪  File(name).getParent + "/" + dataFile)))
51     val labels: ImagePlus = IJ.createImage(name, "8-bit", width, height, depth)
52     for (z <- 1 to depth) {
53       labels.setZ(z)
```

```
54        val processor: ImageProcessor = labels.getProcessor
55        for (y <- 0 until height) {
56          for (x <- 0 until width) {
57            val value: Short = if (byteType == USHORT) in.readShort else
    ↪  in.readByte
58            processor.putPixel(x, y, if (value == 1) 255 else 0)
59          }
60        }
61      }
62      labels
63    }
64
65    /**
66      * Read metadata from a metadata file.
67      * @param name the name of the metadata file
68      * @return a tuple containing, in order, the width of the image, the height
    ↪   of the image, the depth of the image,
69      *         the name of the file containing the image data, the type of the
    ↪   each voxel in the image data
70      */
71    def readMetadata(name: String): (Int, Int, Int, String, ByteType) = {
72      var width = 0
73      var height = 0
74      var depth = 0
75      var dataFile = ""
76      var byteType: ByteType = USHORT
77      for (line <- scala.io.Source.fromFile(name).getLines().map(line =>
    ↪   line.split(" = "))) {
78        if (line(0) == "DimSize") {
79          val dims: Array[String] = line(1).split(" ")
80          width = dims(0).toInt
81          height = dims(1).toInt
82          depth = dims(2).toInt
83        }
84        if (line(0) == "ElementDataFile")
85          dataFile = line(1)
86        if (line(0) == "ElementType")
87          byteType = line(1) match {
88            case "MET_UCHAR" => UCHAR
89            case "MET_USHORT" => USHORT
90            case "MET_SHORT" => USHORT
91            case _ => throw new IllegalArgumentException("Byte type of MHD file
    ↪   unsupported: " + line(1))
92          }
93      }
94      (width, height, depth, dataFile, byteType)
95    }
96
97    /**
98      * Read a short from an input stream.
99      * @param in the stream to read from
100     * @return the short read
101     */
102   def readShort(implicit in: LittleEndianDataInputStream): Int = in.readShort
103 }
```

B.4.2 WindowedImage.scala

```scala
package org.edoardo.image

import ij.process.ImageProcessor
import ij.{IJ, ImagePlus}
import net.imglib2.FinalInterval
import net.imglib2.`type`.numeric.integer.UnsignedByteType
import net.imglib2.algorithm.gradient.PartialDerivative
import net.imglib2.algorithm.pde.PeronaMalikAnisotropicDiffusion
import net.imglib2.img.Img
import net.imglib2.img.array.ArrayImgFactory
import net.imglib2.img.display.imagej.ImageJFunctions
import net.imglib2.util.Intervals
import net.imglib2.view.Views
import org.edoardo.segmentation.SegmentationResult

import scala.Array.ofDim

/**
  * A class storing a windowed image and providing some operations on it.
  *
  * @param originalImage the original image
  * @param windowing     the windowing to apply in the form of (centre, width)
 ↪   or (0,0) to not perform windowing
  */
class WindowedImage(val originalImage: ImagePlus, val windowing: (Int, Int) =
 ↪   (0, 0)) {
  val width: Int = originalImage.getWidth
  val height: Int = originalImage.getHeight
  val depth: Int = originalImage.getDimensions()(3)

  val image: ImagePlus = IJ.createImage("windowed", "8-bit", width, height,
 ↪   depth)

  for (z <- 1 to depth) {
    image.setZ(z)
    originalImage.setZ(z)
    for (x <- 0 until width; y <- 0 until height) {
      if (windowing != (0, 0))
        image.getProcessor.putPixel(x, y,
          127 + Math.round(255 * ((originalImage.getPixel(x, y)(0) -
 ↪   windowing._1).toFloat / windowing._2)))
      else
        image.getProcessor.putPixel(x, y, originalImage.getPixel(x, y)(0))
    }
  }

  val wrapped: Img[UnsignedByteType] = ImageJFunctions.wrapByte(image)

  val diffusionFilter = new PeronaMalikAnisotropicDiffusion(wrapped, 0.13, 30)
  for (i <- 0 until 3)
    diffusionFilter.process()

  val dimensions: Int = if (depth == 1) 2 else 3
```

```
50    val processors: Array[ImageProcessor] = (0 until depth).map(z =>
      ↪  image.getImageStack.getProcessor(z + 1)).toArray
51    var gradientProcessors: Array[Array[ImageProcessor]] = _
52
53    /**
54      * Get the intensity of a given voxel.
55      *
56      * @param x the x coordinate of the voxel
57      * @param y the y coordinate of the voxel
58      * @param z the z coordinate of the voxel
59      * @return the intensity at (x, y, z)
60      */
61    def getVoxel(x: Int, y: Int, z: Int): Int = processors(z).getPixel(x, y)
62
63    /**
64      * Do any required preprocessing on the image before it can be used.
65      */
66    def doPreProcess(): Unit = {
67      computeGradientImage()
68    }
69
70    /**
71      * Convert this image (with white representing voxels chosen, and black
      ↪  representing voxels not chosen) to a
72      * segmentation result.
73      *
74      * @param stayInLayer the layer to remain in, or -1 to consider every layer
75      * @return a segmentation result corresponding to convering this image
76      */
77    def toSegmentationResult(stayInLayer: Int): SegmentationResult = {
78      if (stayInLayer == -1) {
79        val result: Array[Array[Array[Boolean]]] = ofDim[Boolean](width, height,
      ↪  depth)
80        for (x <- 0 until width; y <- 0 until height; z <- 0 until depth)
81          result(x)(y)(z) = getVoxel(x, y, z).equals(255)
82        new SegmentationResult(result)
83      } else {
84        val result: Array[Array[Array[Boolean]]] = ofDim[Boolean](width, height,
      ↪  1)
85        for (x <- 0 until width; y <- 0 until height)
86          result(x)(y)(0) = getVoxel(x, y, stayInLayer).equals(255)
87        new SegmentationResult(result)
88      }
89    }
90
91    /**
92      * Get the maximum of the two (or three) gradients at a given point in the
      ↪  image.
93      *
94      * @param x the x coordinate of the point
95      * @param y the y coordinate of the point
96      * @param z the z coordinate of the point
97      * @return the maximum gradient in x oe y direction at the given point
98      */
99    def getGradient(x: Int, y: Int, z: Int): Int = {
```

```
100       (0 until 2).map(dir => gradientProcessors(dir)(z).getPixel(x, y)).max
101     }
102
103     private def computeGradientImage(): Unit = {
104       val gradients: Img[UnsignedByteType] = new
      ↪  ArrayImgFactory[UnsignedByteType]().create(
105         if (dimensions == 2) Array(width, height, 2)
106         else Array(width, height, depth, dimensions), new UnsignedByteType())
107       val gradientComputationInterval: FinalInterval = Intervals.expand(wrapped,
      ↪  -1)
108       for (d <- 0 until dimensions)
109         PartialDerivative.gradientCentralDifference(wrapped,
110           Views.interval(Views.hyperSlice(gradients, dimensions, d),
      ↪  gradientComputationInterval), d)
111       val gradientImage: ImagePlus = ImageJFunctions.wrap(gradients,
      ↪  image.getShortTitle + "-gradients")
112       gradientProcessors = (0 until dimensions).map(dir => {
113         (0 until depth).map(z => gradientImage.getImageStack.getProcessor((dir *
      ↪  depth) + z + 1)).toArray
114       }).toArray
115     }
116   }
```

B.5 segmentation

B.5.1 RegionInfo.scala

```
1   package org.edoardo.segmentation
2
3   import org.edoardo.rl.{Action, State}
4
5   /**
6     * Represents the possible actions for agent.
7     * @param include whether the action was to add a region or exclude it
8     */
9   case class Decision(include: Boolean) extends Action
10
11  /**
12    * Represents the possible states for the agent.
13    * @param info the information corresponding to region on which the agent
      ↪  should base its decision to include or exclude
14    */
15  case class RegionInfo(info: List[Int]) extends State[Decision] {
16    /**
17      * Gives the actions available for a region, which are always to include or
      ↪  exclude it.
18      * @return the actions available for a region
19      */
20    override def getAvailableActions: List[Decision] = List(Decision(false),
      ↪  Decision(true))
21  }
```

B.5.2 SegmentationResult.scala

```
1   package org.edoardo.segmentation
2
```

```
3  import ij.io.FileSaver
4  import ij.process.ImageProcessor
5  import ij.{IJ, ImagePlus}
6  import inra.ijpb.morphology.{GeodesicReconstruction, GeodesicReconstruction3D}
7
8  /**
9    * Stores the result of a segmentation and provides a few methods for working
   ↪  with it.
10   * @param selected an array containing which voxels were selected
11   */
12 class SegmentationResult(selected: Array[Array[Array[Boolean]]]) {
13   val width: Int = selected.length
14   val height: Int = selected(0).length
15   val depth: Int = selected(0)(0).length
16
17   /**
18     * Morphologically close the result stored by this object.
19     */
20   def closeResult(): Unit = {
21     val result: ImagePlus = buildResult()
22     val closedResult: ImagePlus = IJ.createImage("closedResult", "8-bit", width,
   ↪  height, depth)
23     if (depth > 1)
24       closedResult.setStack(GeodesicReconstruction3D.fillHoles(result.getImageStack))
25     else
26       closedResult.setProcessor(GeodesicReconstruction.fillHoles(result.getProcessor))
27     for (z <- 0 until depth) {
28       closedResult.setZ(z + 1)
29       for (y <- 0 until height; x <- 0 until width)
30         selected(x)(y)(z) = closedResult.getPixel(x, y)(0) == 255
31     }
32   }
33
34   /**
35     * Build up an image of the result stored by this object.
36     * @param value the value to put for pixels that are set (defaults to 255)
37     * @return an image corresponding to the result, with selected pixels white
   ↪  and the rest black
38     */
39   def buildResult(value: Int = 255): ImagePlus = {
40     val result: ImagePlus = IJ.createImage("result", "8-bit", width, height,
   ↪  depth)
41     for (z <- 1 to depth) {
42       result.setZ(z)
43       val processor: ImageProcessor = result.getProcessor
44       for (y <- 0 until height; x <- 0 until width) {
45         if (doesContain(x, y, z - 1))
46           processor.putPixel(x, y, value)
47         else
48           processor.putPixel(x, y, 0)
49       }
50     }
51     result
52   }
53
```

```
54    /**
55      * Build a result image and write it out to a file, overwriting it if it
 ↪     already exists.
56      * @param fileName the name of the file to write the result image to
57      * @param saveAsRaw whether to save the results as a RAW file (will be saved
 ↪     as TIFF otherwise)
58      */
59    def writeTo(fileName: String, saveAsRaw: Boolean): Unit = {
60      if (!saveAsRaw) new FileSaver(buildResult()).saveAsTiff(fileName + ".tiff")
61      else new FileSaver(buildResult(1)).saveAsRaw(fileName + ".raw")
62    }
63
64    /**
65      * Check if a given voxel is in the selection.
66      * @param x the x coordinate
67      * @param y the y coordinate
68      * @param z the z coordinate
69      * @return whether or not the given voxel is in the selection
70      */
71    def doesContain(x: Int, y: Int, z: Int): Boolean = selected(x)(y)(z)
72  }
```

B.5.3 Selection.scala

```
1   package org.edoardo.segmentation
2
3   import org.edoardo.parser.VolumeIPF
4
5   import scala.collection.mutable
6
7   /**
8     * Keeps track of a growing selection and provides methods to include or
 ↪     exclude regions and find out which regions (if
9     * any) still need to be considered for inclusion.
10    * @param height the height of the image this selection is in
11    * @param width the width of the image this selection is in
12    * @param depth the depth of the image this selection is in
13    * @param ipf the IPF we are using for the image
14    * @param stayInLayer whether or not we should remain in the layer we start in
15    */
16  class Selection(val height: Int, width: Int, depth: Int, ipf: VolumeIPF,
 ↪     stayInLayer: Boolean) {
17    val toConsider: mutable.Set[Int] = mutable.Set[Int]()
18    var toConsiderQueue: mutable.Queue[Int] = mutable.Queue[Int]()
19    val excluded: mutable.Set[Int] = mutable.Set[Int]()
20    val included: mutable.Set[Int] = mutable.Set[Int]()
21    var firstZ: Int = -1
22
23    /**
24      * Check if we have finished considering regions.
25      * @return whether or not we have finished considering regions
26      */
27    def completed(): Boolean = toConsider.isEmpty
28
29    /**
```

```
30      * Get a region that we still need to consider. Note this should only be
    ↪  called if completed() is false.
31      * @return a region that still needs considering
32      */
33    def getRegion: Int = {
34      val result: Int = toConsiderQueue.dequeue()
35      toConsider.remove(result)
36      result
37    }
38
39    /**
40      * Add a region to our selection, and add its neighbours to the regions we
    ↪  need to consider (if they have not
41      * already been included or excluded).
42      * @param region the region to add to our selection
43      */
44    def includeRegion(region: Int): Unit = {
45      included += region
46      for (neighbour <- ipf.getNeighbours(region)) {
47        if (!excluded.contains(neighbour) && !included.contains(neighbour)) {
48          if (!(stayInLayer && ipf.getZ(region) != firstZ) &&
    ↪  toConsider.add(neighbour))
49            toConsiderQueue.enqueue(neighbour)
50        }
51      }
52    }
53
54    /**
55      * Seed our selection with a starting region corresponding to the region at a
    ↪  given coordinate in a given layer
56      * of the IPF.
57      * @param x the x coordinate of our seed
58      * @param y the y coordinate of our seed
59      * @param z the z coordinate of our seed
60      * @param layer the layer of the IPF to start our selection in
61      */
62    def startPixel(x: Int, y: Int, z: Int, layer: Int): Unit = {
63      val startRegions: List[Int] = ipf.getRegionsInLayer(layer, x, y, z)
64      firstZ = z
65      for (region <- startRegions)
66        includeRegion(region)
67    }
68
69    /**
70      * Exclude a region from our selection, preventing it from being considered
    ↪  again. This is necessary to avoid
71      * infinitely looping if there is a cycle of regions we do not wish to
    ↪  include.
72      * @param region the region to exclude
73      */
74    def excludeRegion(region: Int): Unit = {
75      excluded += region
76    }
77
78    /**
```

```
79      * Get the result of the segmentation corresponding to this selection.
80      * @return the result of the segmentation corresponding to this selection
81      */
82    def getResult: SegmentationResult = {
83      val status: Array[Array[Array[Boolean]]] = Array.ofDim(width, height, depth)
84      for ((x, y, z) <- included.flatMap(region => ipf.getRegionPixels(region)))
85        status(x)(y)(z) = true
86      new SegmentationResult(status)
87    }
88
89  }
```

B.5.4 RLSegmentation.scala

```
1   package org.edoardo.segmentation
2
3   import java.io.File
4
5   import ij.IJ
6   import ij.io.{FileSaver, Opener}
7   import ij.plugin.FolderOpener
8   import org.edoardo.image.{Raw, WindowedImage}
9   import org.edoardo.parser.{IPF, MFS, VolumeIPF}
10  import org.edoardo.rl.Policy
11
12  import scala.collection.mutable
13
14  /**
15    * Contains the main code for running the segmentation algorithm.
16    */
17  object RLSegmentation {
18    val policy = new Policy[Decision, RegionInfo]
19    val opener = new Opener()
20    val regionInfoCache: mutable.Map[Int, RegionInfo] = mutable.Map.empty
21    var epsilonReciprocal = 10
22
23    def experimentFive(): Unit = {
24
25      val imageInfos = List(
26        // Training data
27        ImageInfo(4, "image-004.mhd", 66, (100, 100, 66), (416, 781)),
28        ImageInfo(15, "image-015.mhd", 70, (150, 100, 70), (251, 492)),
29        ImageInfo(41, "image-041.mhd", 59, (100, 140, 59), (79, 150)),
30        ImageInfo(42, "image-042.mhd", 61, (100, 100, 61), (492, 985)),
31        ImageInfo(50, "image-050.mhd", 64, (100, 100, 64), (64, 128)),
32  //
33  //      // Evaluation data
34        ImageInfo(39, "image-039.mhd", 69, (100, 100, 69), (64, 127)),
35        ImageInfo(46, "image-046.mhd", 69, (100, 100, 69), (870, 1493))
36  //      ImageInfo(52, "image-052.mhd", 83, (100, 100, 0), (162, 321)),
37  //      ImageInfo(54, "image-054.mhd", 75, (100, 100, 0), (598, 1013)),
38  //      ImageInfo(60, "image-060.mhd", 76, (100, 100, 0), (415, 830))
39      )
40      println("-- Before Training --")
41      for (imageInfo <- imageInfos)
42        doImage(imageInfo.fileName, "image" + imageInfo.id + ".ipf",
```

```
43          "preTraining-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
44          Some("labels-%03d.mhd".format(imageInfo.id.toInt)), -1, 0, saveAsRaw =
    ↪  false)
45
46      println("-- Training --")
47      for (imageInfo <- imageInfos.take(5))
48        doImage(imageInfo.fileName, "image" + imageInfo.id + ".ipf",
49          "training-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
50          Some("labels-%03d.mhd".format(imageInfo.id.toInt)), -1, 40, saveAsRaw =
    ↪  false)
51
52      println("-- After Training --")
53      for (imageInfo <- imageInfos)
54        doImage(imageInfo.fileName, "image" + imageInfo.id + ".ipf",
55          "postTraining-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
56          Some("labels-%03d.mhd".format(imageInfo.id.toInt)), -1, 0, saveAsRaw =
    ↪  false)
57    }
58
59    /**
60      * Main method to run our segmentation algorithm.
61      *
62      * @param args A number between one and three containing the number of the
    ↪  experiment to run.
63      */
64    def main(args: Array[String]): Unit = {
65      if (args(0) == "1")
66        experiementOne()
67      else if (args(0) == "2")
68        experiementTwo()
69      else if (args(0) == "3")
70        experimentThree()
71      else if (args(0) == "4")
72        experiementFour()
73      else
74        experimentFive()
75    }
76
77    /**
78      * The first experiment we ran on MRI scans from the Botnar Research Centre.
79      */
80    def experiementOne(): Unit = {
81      val imageInfos = List(
82        // Training data
83        ImageInfo(1, "Knee1/Knee1_0009.dcm", 10, (200, 200, 0), (1214, 2408)),
84        ImageInfo(10, "Knee10/Knee10_0009.dcm", 10, (200, 200, 0), (1050, 2050)),
85        ImageInfo(11, "Knee11/Knee11_0009.dcm", 9, (200, 200, 0), (1100, 2200)),
86        ImageInfo(12, "Knee12/Knee12_0009.dcm", 10, (200, 200, 0), (1050, 2050)),
87        ImageInfo(13, "Knee13/Knee13_0010.dcm", 10, (200, 200, 0), (1050, 2050)),
88
89        // Evaluation data
90        ImageInfo(2, "Knee2/Knee2_0006.dcm", 6, (200, 150, 0), (1325, 1604)),
91        ImageInfo(3, "Knee3/Knee3_0010.dcm", 11, (200, 200, 0), (1126, 2231)),
92        ImageInfo(5, "Knee5/Knee5_0008.dcm", 9, (200, 200, 0), (1275, 2500)),
93        ImageInfo(6, "Knee6/Knee6_0010.dcm", 11, (200, 200, 0), (1200, 2400)),
```

```
94        ImageInfo(7, "Knee7/Knee7_0010.dcm", 11, (200, 150, 0), (1050, 2050))
95      )
96      println("-- Before Training --")
97      for (imageInfo <- imageInfos)
98        doImage(imageInfo.fileName, "knee" + imageInfo.id + "-layer" +
    ↪   imageInfo.layer + ".ipf",
99          "preTraining-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
100         Some("knee" + imageInfo.id + "-layer" + imageInfo.layer + ".mfs"),
    ↪   imageInfo.layer, 0, saveAsRaw = false)
101
102     println("-- Training --")
103     for (imageInfo <- imageInfos.take(5))
104       doImage(imageInfo.fileName, "knee" + imageInfo.id + "-layer" +
    ↪   imageInfo.layer + ".ipf",
105         "training-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
106         Some("knee" + imageInfo.id + "-layer" + imageInfo.layer + ".mfs"),
    ↪   imageInfo.layer, 40, saveAsRaw = false)
107
108     // printPolicy()
109
110     println("-- After Training --")
111     for (imageInfo <- imageInfos)
112       doImage(imageInfo.fileName, "knee" + imageInfo.id + "-layer" +
    ↪   imageInfo.layer + ".ipf",
113         "postTraining-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
114         Some("knee" + imageInfo.id + "-layer" + imageInfo.layer + ".mfs"),
    ↪   imageInfo.layer, 0, saveAsRaw = false)
115   }
116
117   /**
118     * The second experiment we ran on MRI scans from SKI10 grand challenge.
119     */
120   def experiementTwo(): Unit = {
121     val imageInfos = List(
122       // Training data
123       ImageInfo(4, "image-004.mhd", 66, (100, 100, 0), (416, 781)),
124       ImageInfo(15, "image-015.mhd", 70, (150, 100, 0), (251, 492)),
125       ImageInfo(41, "image-041.mhd", 59, (100, 140, 0), (79, 150)),
126       ImageInfo(42, "image-042.mhd", 61, (100, 100, 0), (492, 985)),
127       ImageInfo(50, "image-050.mhd", 64, (100, 100, 0), (64, 128)),
128
129       // Evaluation data
130       ImageInfo(39, "image-039.mhd", 69, (100, 100, 0), (64, 127)),
131       ImageInfo(46, "image-046.mhd", 69, (100, 100, 0), (870, 1493)),
132       ImageInfo(52, "image-052.mhd", 83, (100, 100, 0), (162, 321)),
133       ImageInfo(54, "image-054.mhd", 75, (100, 100, 0), (598, 1013)),
134       ImageInfo(60, "image-060.mhd", 76, (100, 100, 0), (415, 830))
135     )
136     println("-- Before Training --")
137     for (imageInfo <- imageInfos)
138       doImage(imageInfo.fileName, "image" + imageInfo.id + "-layer" +
    ↪   imageInfo.layer + ".ipf",
139         "preTraining-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
140         Some("labels-%03d.mhd".format(imageInfo.id.toInt)), imageInfo.layer, 0,
    ↪   saveAsRaw = false)
```

```
141
142     println("-- Training --")
143     for (imageInfo <- imageInfos.take(5))
144       doImage(imageInfo.fileName, "image" + imageInfo.id + "-layer" +
    ↪  imageInfo.layer + ".ipf",
145         "training-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
146         Some("labels-%03d.mhd".format(imageInfo.id.toInt)), imageInfo.layer, 40,
    ↪  saveAsRaw = false)
147
148     println("-- After Training --")
149     for (imageInfo <- imageInfos)
150       doImage(imageInfo.fileName, "image" + imageInfo.id + "-layer" +
    ↪  imageInfo.layer + ".ipf",
151         "postTraining-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
152         Some("labels-%03d.mhd".format(imageInfo.id.toInt)), imageInfo.layer, 0,
    ↪  saveAsRaw = false)
153   }
154
155   /**
156     * The third experiment we ran on XRay images.
157     */
158   def experimentThree(): Unit = {
159     val xrayInfos = List(
160       XRayInfo(2, "knee2.mfs", (135, 250, 0)),
161       XRayInfo(3, "knee3-gt.pgm", (135, 237, 0)),
162       XRayInfo(1, "knee1-gt.pgm", (264, 579, 0))
163     )
164     println("-- Before Training --")
165     for (xrayInfo <- xrayInfos)
166       doImage("knee" + xrayInfo.id + ".pgm", "knee" + xrayInfo.id + ".ipf",
    ↪  "preTraining-" + xrayInfo.id,
167         xrayInfo.seed, (127, 255), Some(xrayInfo.gt), 0, 0, saveAsRaw = false)
168
169     println("-- Training --")
170     for (xrayInfo <- xrayInfos.take(2))
171       doImage("knee" + xrayInfo.id + ".pgm", "knee" + xrayInfo.id + ".ipf",
    ↪  "training-" + xrayInfo.id,
172         xrayInfo.seed, (127, 255), Some(xrayInfo.gt), 0, 40, saveAsRaw = false)
173
174     println("-- After Training --")
175     for (xrayInfo <- xrayInfos)
176       doImage("knee" + xrayInfo.id + ".pgm", "knee" + xrayInfo.id + ".ipf",
    ↪  "postTraining-" + xrayInfo.id,
177         xrayInfo.seed, (127, 255), Some(xrayInfo.gt), 0, 0, saveAsRaw = false)
178   }
179
180   /**
181     * The fourth experiment we ran to see the effect of changing epsilon.
182     */
183   def experiementFour(): Unit = {
184     val imageInfos = List(
185       // Training data
186       ImageInfo(1, "Knee1/Knee1_0009.dcm", 10, (200, 200, 0), (1214, 2408)),
187       ImageInfo(10, "Knee10/Knee10_0009.dcm", 10, (200, 200, 0), (1050, 2050)),
188       ImageInfo(11, "Knee11/Knee11_0009.dcm", 9, (200, 200, 0), (1100, 2200)),
```

```
189       ImageInfo(12, "Knee12/Knee12_0009.dcm", 10, (200, 200, 0), (1050, 2050)),
190       ImageInfo(13, "Knee13/Knee13_0010.dcm", 10, (200, 200, 0), (1050, 2050)),
191
192       // Evaluation data
193       ImageInfo(2, "Knee2/Knee2_0006.dcm", 6, (200, 150, 0), (1325, 1604)),
194       ImageInfo(3, "Knee3/Knee3_0010.dcm", 11, (200, 200, 0), (1126, 2231)),
195       ImageInfo(5, "Knee5/Knee5_0008.dcm", 9, (200, 200, 0), (1275, 2500)),
196       ImageInfo(6, "Knee6/Knee6_0010.dcm", 11, (200, 200, 0), (1200, 2400)),
197       ImageInfo(7, "Knee7/Knee7_0010.dcm", 11, (200, 150, 0), (1050, 2050))
198     )
199     val repeats = 10
200     var resultString = ""
201     for (epsilon <- List(2, 5, 10, 20, 40, 100)) {
202       for (numRuns <- List(2, 5, 10, 20, 40, 100)) {
203         var results: List[Float] = List()
204         for (i <- 0 until repeats) {
205           epsilonReciprocal = epsilon
206           println("-- Training --")
207           for (imageInfo <- imageInfos.take(5))
208             doImage(imageInfo.fileName, "knee" + imageInfo.id + "-layer" +
    ↪  imageInfo.layer + ".ipf",
209               "training-" + imageInfo.id, imageInfo.seed, imageInfo.windowing,
210               Some("knee" + imageInfo.id + "-layer" + imageInfo.layer + ".mfs"),
    ↪  imageInfo.layer, numRuns, saveAsRaw = false)
211
212           println("-- After Training " + numRuns + " Runs with epsilon = 1/" +
    ↪  epsilonReciprocal + " (Repeat " + i + ")--")
213           var runResults: List[Float] = List()
214           for (imageInfo <- imageInfos.drop(5))
215             runResults ::= doImage(imageInfo.fileName, "knee" + imageInfo.id +
    ↪  "-layer" + imageInfo.layer + ".ipf",
216               "postTraining-" + imageInfo.id, imageInfo.seed,
    ↪  imageInfo.windowing,
217               Some("knee" + imageInfo.id + "-layer" + imageInfo.layer + ".mfs"),
    ↪  imageInfo.layer, 0, saveAsRaw = false)
218           results ::= runResults.sum / 5
219           policy.clear()
220         }
221         resultString += results.sum / repeats + "\t"
222       }
223       resultString = resultString.dropRight(1) + "\n"
224     }
225     println("-- Average Results --")
226     println(resultString)
227   }
228
229   /**
230     * Apply our algorithm to an image.
231     *
232     * @param name            the name of the file (or folder) the image (or
    ↪  layers of the image) can be found in
233     * @param ipfName         the name of the file containing the IPF for the
    ↪  image
234     * @param resultName      the name of the result file to store the
    ↪  segmentation result in, should end in .tiff
```

```
235     * @param seed            the seed point to begin growing the region from
236     * @param windowing       the windowing to use, in the form of a pair of
    ↪  (centre, width)
237     * @param gtName          the name of the file containing the gold standard
    ↪  to compare with (and learn from, if applicable)
238     * @param stayInLayer     the layer to explore in (-1 to explore the whole
    ↪  image)
239     * @param numPracticeRuns the number of times to practice on this image (0 to
    ↪  not train on this image)
240     * @param saveAsRaw       whether to save the image as RAW (will be saved as
    ↪  TIFF otherwise)
241     * @return the DSC of the segmentation after training (-1 if no gold standard
    ↪  was provided)
242     */
243    def doImage(name: String, ipfName: String, resultName: String, seed: (Int,
    ↪  Int, Int), windowing: (Int, Int) = (0, 0),
244            gtName: Option[String] = None, stayInLayer: Integer = -1,
    ↪  numPracticeRuns: Int = 40, saveAsRaw: Boolean): Float = {
245      val img: WindowedImage = new WindowedImage(
246        if (name.takeRight(3) == "mhd")
247          Raw.openMetadata(name, stayInLayer)
248        else if (new File(name).isDirectory)
249          new FolderOpener().openFolder(name)
250        else IJ.openImage(name), windowing)
251      new FileSaver(img.image).saveAsTiff(resultName + "-original.tiff")
252      val ipf: VolumeIPF = IPF.loadFromFile(ipfName)
253      val gt: Option[SegmentationResult] = gtName.map(name =>
254        if (name.takeRight(3) == "mfs")
255          MFS.loadFromFile(name, ipf)
256        else
257          new WindowedImage(
258            if (name.takeRight(3) == "mhd")
259              Raw.openLabels(name)
260            else opener.openImage(name)
261          ).toSegmentationResult(stayInLayer))
262      if (gt.isDefined)
263        gt.get.writeTo(resultName + "-gt", saveAsRaw)
264      img.doPreProcess()
265      if (gt.isDefined) {
266        for (i <- 0 until numPracticeRuns) {
267          val result: SegmentationResult = analyseImage(img, ipf, gt, seed,
    ↪  stayInLayer != -1)
268          println(name + "\t" + i + "\t" + score(result, gt.get))
269          result.writeTo(resultName + "-" + i, saveAsRaw)
270        }
271      }
272      val result: SegmentationResult = analyseImage(img, ipf, None, seed,
    ↪  stayInLayer != -1)
273      var dsc: Float = -1
274      if (gt.isDefined) {
275        val scoreString: String = score(result, gt.get)
276        println(name + "\tfin\t" + scoreString)
277        dsc = scoreString.takeWhile(x => x != '\t').toFloat
278      }
279      result.writeTo(resultName, saveAsRaw)
```

```
280     regionInfoCache.clear()
281     dsc
282   }
283
284   /**
285     * Get the information for a given region from the first branch layer of the
      ↪  IPF.
286     *
287     * @param region the identifier for the region
288     * @param ipf    the IPF of the image we are considering
289     * @param img    the image we are considering
290     * @return the information to be used by the agent to decide whether or not
      ↪  to include this region
291     */
292   def getInfo(region: Int, ipf: VolumeIPF, img: WindowedImage): RegionInfo = {
293     regionInfoCache.getOrElseUpdate(region, {
294       val pixels: List[(Int, Int, Int)] = ipf.getRegionPixels(region)
295       val avgIntensity: Int = pixels.map(p => img.getVoxel(p._1, p._2,
      ↪  p._3)).sum / pixels.size
296       // val minIntensity: Int = pixels.map(p => img.getVoxel(p._1, p._2,
      ↪  p._3)).min
297       // val maxIntensity: Int = pixels.map(p => img.getVoxel(p._1, p._2,
      ↪  p._3)).max
298       // val avgGradient: Int = pixels.map(p => img.getGradient(p._1, p._2,
      ↪  p._3)).sum / pixels.size
299       val maxGradient: Int = pixels.map(p => img.getGradient(p._1, p._2,
      ↪  p._3)).max
300       RegionInfo(List(avgIntensity, maxGradient))
301     })
302   }
303
304   /**
305     * Analyse the given image.
306     *
307     * @param img         the image to analyse
308     * @param ipf         the IPF for the image
309     * @param gt          the gold standard to compare to, if applicable
310     * @param seed        the seed point to grow from
311     * @param stayInLayer whether or not to remain in the same layer
312     * @return the result of segmenting the image
313     */
314   def analyseImage(img: WindowedImage, ipf: VolumeIPF, gt:
      ↪  Option[SegmentationResult], seed: (Int, Int, Int),
315             stayInLayer: Boolean): SegmentationResult = {
316     if (gt.isDefined)
317       assert(img.width == gt.get.width && img.height == gt.get.height &&
      ↪  img.depth == gt.get.depth)
318     val selection = new Selection(img.height, img.width, img.depth, ipf,
      ↪  stayInLayer)
319     var decisions: List[(RegionInfo, Int, Decision)] = List()
320     selection.startPixel(seed._1, seed._2, seed._3, if (gt.isDefined) 2 else 2)
321     while (!selection.completed()) {
322       val region: Int = selection.getRegion
323       val state: RegionInfo = getInfo(region, ipf, img)
324       val decision: Decision =
```

```
325         if (gt.isEmpty) policy.greedyPlay(state)
326         else policy.epsilonSoft(state, epsilonReciprocal)
327       decisions ::= (state, region, decision)
328       if (decision.include)
329         selection.includeRegion(region)
330       else
331         selection.excludeRegion(region)
332     }
333     if (gt.isDefined)
334       decisions.foreach {
335         case (state, region, dec) =>
336           policy.update(state, dec, reward(region, dec.include, ipf, gt))
337       }
338     val result: SegmentationResult = selection.getResult
339     result.closeResult()
340     result
341   }
342
343   /**
344     * Calculates the reward to give our agent for deciding to include or exclude
      ↪  a given region.
345     *
346     * @param region   the region considered
347     * @param decision whether or not the agent chose to include it
348     * @param ipf      the IPF for the
349     * @param gt       the gold standard we are comparing against (this function
      ↪  will return constant 0 if this is None)
350     * @return a value corresponding to how many more pixels the decision was
      ↪  correct for (so, this will be a positive
351     *         value if the correct decision was made, and negative otherwise)
352     */
353   def reward(region: Int, decision: Boolean, ipf: VolumeIPF, gt:
      ↪  Option[SegmentationResult]): Int = {
354     if (gt.isEmpty) return 0
355     val pixels: List[(Int, Int, Int)] = ipf.getRegionPixels(region)
356     var reward: Int = 0
357     for ((x, y, z) <- pixels)
358       reward += (if (gt.get.doesContain(x, y, z)) 1 else -1)
359     if (decision) reward
360     else -reward
361   }
362
363   /**
364     * Create a string containing the scores of a segmentation compared to a gold
      ↪  standard.
365     *
366     * @param result the result of a segmentation
367     * @param gt     the gold standard we are comparing against
368     * @return A tab separated String of values consisting of the DSC, TPVF and
      ↪  FPVF of the segmentation.
369     */
370   def score(result: SegmentationResult, gt: SegmentationResult): String = {
371     var overlap = 0
372     var resultSize = 0
373     var gtSize = 0
```

```
374     var falsePositive = 0
375     val imageSize: Int = gt.height * gt.width * gt.depth
376     for (x <- 0 until result.width; y <- 0 until result.height; z <- 0 until
    ↪  result.depth) {
377       if (result.doesContain(x, y, z) && gt.doesContain(x, y, z)) overlap += 1
378       if (result.doesContain(x, y, z)) resultSize += 1
379       if (gt.doesContain(x, y, z)) gtSize += 1
380       if (result.doesContain(x, y, z) && !gt.doesContain(x, y, z)) falsePositive
    ↪  += 1
381     }
382     (2f * overlap) / (gtSize + resultSize) + "\t" +
383       overlap.toFloat / gtSize + "\t" +
384       falsePositive.toFloat / (imageSize - gtSize)
385   }
386
387   /**
388     * Prints out the current policy (ie. what the agent believes to be the best
    ↪  action in every state).
389     */
390   def printPolicy(): Unit = {
391     println("-- Policy Learnt --")
392     for (x <- 0 to 255) {
393       for (y <- 0 to 255)
394         print((if (!policy.haveEncountered(RegionInfo(List(x, y)))) 0
395         else if (policy.greedyPlay(RegionInfo(List(x, y))).include) 1
396         else -1) + "\t")
397       println()
398     }
399     printPercentageSeen()
400   }
401
402   /**
403     * Prints out the percentage of theoretically possible states actually
    ↪  encountered.
404     */
405   def printPercentageSeen(): Unit = {
406     var total = 0
407     var seen = 0
408     for (x <- 0 to 255; y <- 0 to 255) {
409       total += 1
410       if (policy.haveEncountered(RegionInfo(List(x, y))))
411         seen += 1
412     }
413     println("Percentage of states seen: " + 100 * (seen.toFloat / total))
414   }
415
416   private case class ImageInfo(id: Integer, fileName: String, layer: Integer,
    ↪  seed: (Int, Int, Int), windowing: (Int, Int))
417
418   private case class XRayInfo(id: Integer, gt: String, seed: (Int, Int, Int))
419
420 }
```