Deep Reinforcement Learning for Medical Image Registration

Ian Watt

Department of Electrical & Computer Engineering
McGill University

December 14, 2019

A thesis submitted to McGill University in partial fulfillment of the requirements for a Master's in Engineering

2019/12/14

Résumé

Le recalage d'image consiste à aligner différentes images en fonction de leur contenu. Le recalage multi-modal, un défi courant dans le domaine de l'imagerie médicale, concerne spécifiquement les images acquises à travers des mécanismes d'imageries différents. Ceci peut permettre une compréhension plus fine de l'état d'un patient en combinant des informations uniques à chaque modalité. Cependant, le recalage multi-modal présente un bon nombre de défis dépendamment des modalités utilisées. Les structures anatomiques peuvent avoir des apparences très diverses dans différentes modalités, à un point que des tissus qui sont facilement différenciables dans une modalité peuvent apparaître comme homogènes dans une autre. Le mouvement du patient, la croissance et la progression de la maladie induisent des déformations non-uniformes pour lesquelles le recalage non-rigide donne de meilleurs résultats. La complexité additionnelle des techniques non-rigides peut cependant présenter un défi supplémentaire, et le coût de calcul additionnel rend difficile les applications en temps réel telles que le recalage intra-opératoire.

Le présent manuscrit examine l'applicabilité de réseaux profonds d'apprentissage par renforcement pour la tâche de recalage d'images multi-modale. Un réseau profond d'apprentissage-Q est entraîné pour recaler itérativement des images de diverses modalités. Ceci est fait en utilisant une approche de séparation des sujets pour lier une instance distincte de l'agent d'apprentissage par renforcement à chaque node d'une transformation B-spline. Ces instances multiples travaillant en tandem peuvent recaler une image entière relativement rapidement.

Cette approche a été évaluée sur des images par résonance magnétique de cerveaux (séquences T1 et T2), dans un contexte mono-modal et multi-modal basse résolution. Les résultats pour le contexte multi-modal à résolution complète ont été jugés peu concluants, apparemment limités par la taille du réseau.

Abstract

Image registration is the process of aligning different images based on their content. Multimodal registration, a common challenge in medical imaging, specifically concerns images acquired via different imaging mechanisms. This can enable a more complete understanding of a patient's state by combining information unique to each modality. However, multimodal registration presents a number of challenges depending on the modalities used. Anatomical structures may have very dissimilar appearances in different imaging modalities, to the point that tissue types which are easily differentiated in one modality may appear homogeneous in another. Patient movement, growth, and disease progression all introduce non-uniform deformations that are best addressed by non-rigid registration. The additional complexity of such registration can present a challenge, though, and increased computation cost presents a challenge for real-time applications such as intra-operative registration.

This thesis examines the applicability of deep reinforcement learning to the task of multimodal medical image registration. A modified deep-Q network is trained to iteratively register images of a chosen target and pair of modalities, using a separation of concerns approach to attach a separate instance of the RL agent to each node of a B-spline transform. These multiple instances working in tandem may then register the entire image relatively quickly.

The approach was tested with T1 and T2 MRI brain images and was found to function with monomodal and low resolution multimodal registration. The results for full resolution multimodal registration were deemed inconclusive, seemingly limited by the achievable network size.

Acknowledgements

I would like to thank Professor Tal Arbel for her wide range of advice, guidance, and instruction; Professor Doina Precup for sharing her expertise on all things Machine Learning; the students of the Probabilistic Vision Group for thought provoking discussion and a sense of camaraderie; and my family for their constant support.

Contents

List of Figures

List of Acronyms

ANTs	Advanced Normalization Tools
CT	Computed Tomography
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DQN	Deep Q-Network
ERB	Experience Replay Buffer
FC	Fully Connected
GPU	Graphics Processing Unit
HCP	Human Connectome Project
IEEE	Institute of Electrical and Electronics Engineers
ITK	Insight Segmentation and Registration Toolkit
MICCAI	Medical Image Computing & Computer Assisted Intervention
ML	Machine Learning
MR	Magnetic Resonance
MRI	Magnetic Resonance Imaging
MSER	Maximally Stable Extremal Region
NR	Non-Rigid
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
SIFT	Scale-invariant Feature Transform
SoC	Separation of Concerns

Chapter 1

Introduction

In this thesis we consider a novel approach to non-rigid medical image registration based on deep reinforcement learning. Image registration in the most general sense refers to the process of automatically aligning images with related content. In medical imaging this can take many forms, including aligning scans of the same organ taken at different times; aligning different modality images of the same organ; or aligning a patient image with a standardised reference. Each of these applications can potentially help multiple endeavours, from diagnosis to treatment monitoring to research.

We are specifically interested in deformable registration [1]. In many medical applications local accuracy in regions of interest is key, and while rigid registration may provide a good global alignment an organ's soft tissue movement can lead to significant non-uniform local deformation. Non-rigid registration can correct for this, but comes with the drawback that it is often slower to complete registrations, and more difficult to implement and verify compared to rigid methods [2].

Multimodal registration is particularly useful, as it allows integration of modality specific information to form a more complete description of a subject. This presents additional difficulties beyond monomodal registration, as one must first discover and quantify the relationships between different representations of the subject - relationships that can be quite complex, given the vast functional differences between imaging technologies. The relationships are also modality dependant and must be custom designed for every pair of modalities considered.

The intersection of these two, non-rigid registration of multimodal images, is therefore a highly desired application [3]. Unfortunately, it is by many accounts the most challenging problem within medical image registration. The challenge of interpreting multimodal image pairs limits the useful information that most algorithms can extract, whereas deformable registrations often require more information to accurately solve given the greater number of parameters associated with those image transforms.

Machine learning has contributed to attempts at solving this problem, and in recent years new systems have been introduced that notably outperform previous approaches. Deep learning in particular, enabled by modern hardware, counts registration among the fields it has dramatically influenced. This is because machine learning excels at finding patterns and learning relationships, such as feature correspondence between modalities, directly from the data of interest. While domain specific expertise is still very important, the automatically learned features and patterns of ML systems often improve performance beyond that found with traditional handcrafted features [4].

However, even with these improved techniques, it is still difficult to find a solution for the large number of parameters in a deformable transformation [5]. While impressive, modern deep learning based multimodal registration systems still must simplify the problem to succeed, often by imposing limits on the transform to reduce it's dimensionality.

Here we consider a different method of simplification. By using a b-spline transform the parameters are isolated into pairs (or triads, for 3D images) and are associated with specific sub-regions of the image. Finding the solution for a single parameter group is then a much easier problem, and due to the common properties between them it is possible to treat all the parameter groups within an image as examples of the same problem type, and to train a single deep learning based system to individually solve each such group.

There is a catch to this approach, however. As each parameter group primarily controls the deformation of its associated sub-region, useful information for the solution of the group is located within that subregion, and finding the solution to each group almost becomes its own local sub-registration problem. One must then consider that local complexity within medical images is highly variable, containing both detailed and homogeneous regions. This is especially true when considering multimodal pairs, as their local complexity may vary independently of each other, and useful features must exist in both modalities to find a solution. It would therefore seem highly unlikely that a system could learn to consistently find accurate solutions for every parameter group in an image if it must consider them fully

independently. And, beyond local inaccuracy, such errors could introduce implausible local deformations into the aggregate solution.

If local group solutions are to influence each other, as it seems they must, this makes one-step approaches undesireable. Predefined restrictions on allowable transforms is exactly what we are trying to avoid, but a learned relationship becomes incredibly complex if the mutual influence of pairs must be recursively considered. An iterative solution would therefore seem preferable, essentially allowing message passing between identical nodes with each step. And, as each parameter pair is tied to a location within an image, there is a helpful predefined spacial relationship between these nodes.

Reinforcement learning presents itself as a natural fit for this problem [6]. RL systems are used for problems requiring iterative action selection based on observations of some changing state influenced by those actions - in this case, iterative parameter updates based on the resulting image deformations. More specifically, we will focus on deep-RL, as it incorporates the same deep neural network technology that has been so successful in other image registration systems.

The use of deep-RL for this type of problem was further encouraged by the work of Ghesu et al. [7] on feature localization. Their system randomly places a small moving window within an image, and a trained deep-RL agent updates the window's position based on its contents, seeking out the feature of interest. If a similar system could be adapted such that the target feature was determined by the contents of a second window, overlaid upon a second image, that would be a perfect match for the b-spline system under consideration. With the added benefit of message passing between nodes, this seems to be an increasingly plausible approach.

Iterative solutions will necessarily be slower than one-shot systems, but we may still expect this sort of system to solve problems quickly. Preliminary training of deep networks often takes significant time, but once that has been completed most such systems can process new data very quickly. Based on deep-RL image processing applications of similar complexity, it is likely that applying the simple image deformations this technique uses will take longer than selecting those actions.

For testing purposes, this thesis will focus on registration of T1 and T2 MRI brain images taken from the Human Connectome Project [8], an example pair of which is shown in Figure 1.1. We will begin with an overview of image registration and of machine learning in general, and then proceed to a deeper discussion of reinforcement learning and deep-RL. These topics will primarily be discussed in the overarching context of medical image processing. After outlining the appropriate background material, we examine the registration system developed for this thesis, explore the effect that various sub-components of the system have on the registration process, and evaluate the system's potential.

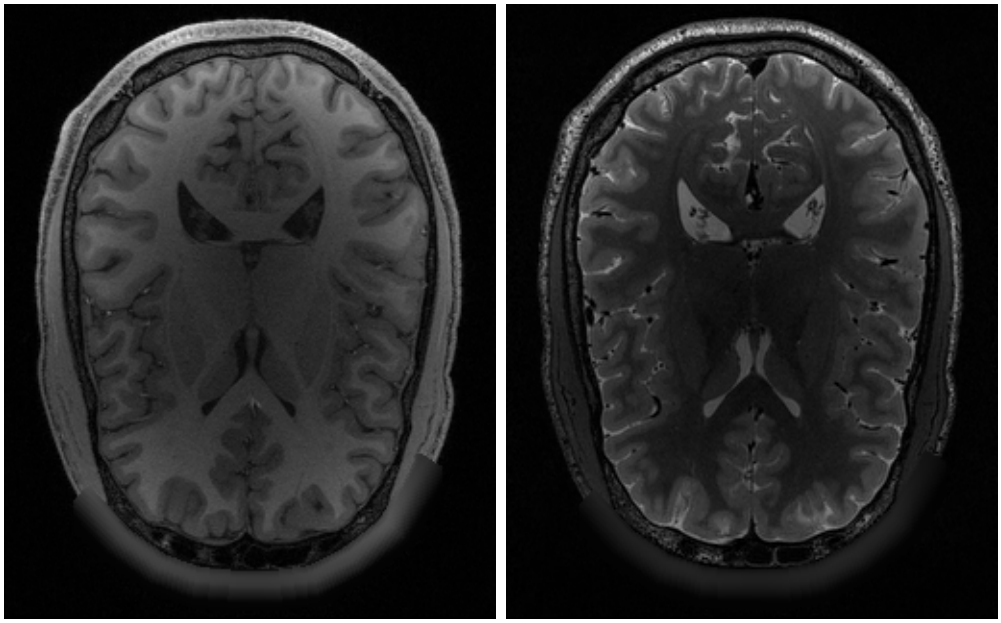

Figure 1.1 Example HCP [8] brain image pair, T1 on left, T2 on right.

Chapter 2

Background

Before discussing image registration via deep reinforcement learning, it is useful to discuss the component pieces that make up such a system, as well as some established previous methods that inform the motivation behind those modern components. This chapter will therefore briefly discuss image registration, machine learning, and deep learning, before reviewing a number of non-RL deep learning approaches to medical image registration. Reinforcement learning, being more central to the method proposed in this thesis, will be discussed in greater depth in Chapter 3.

2.1 Image Registration

2.1.1 Introduction

In the context of images, 'registration' is the process of finding a transform that when applied to one image of a pair will align it with its counterpart. It is generally assumed that the two images are of the same object, or at least of the same type, and that after registration the objects' corresponding features in each image will occupy the same location in the image space. Performing registration requires four related components to be defined: a similarity metric, a transform, and a method of optimization. Additionally a method of regularization is often useful and sometimes required [1].

Medical images present their own domain-specific registration challenges, some common to all image types and some unique to specific modalities or subjects. Non-rigid deformation is frequent, as most of the the human body consists of flexible soft tissue. Some modalities,

such as ultrasound, are orientation dependant and 3D images will change depending on what angle they are acquired from. Noise and artifacts are common.

Multimodal registration is a particularly potentially useful application, but introduces yet more hurdles. Tissue distinguishability in particular depends on modality, and so a homogeneous region in one modality may contain significant detail in another.

2.1.2 Similarity Metrics

A similarity metric is a function that receives two images and produces a scalar value (Equation 2.1), defined in such a way that "more similar" images consistently produce higher values than "less similar" images (Equation 2.2). It follows that the highest value should be returned when comparing an image to itself (Equation 2.3).

What exactly qualifies as more or less similar in the colloquial sense has no singular mathematical definition; there exist an arbitrarily high number of metrics, measuring any specific image properties one may wish to consider. It is therefore necessary for the registration algorithm designer to choose a specific function definition that will lead to desirable results given the nature of the images to be registered. We will explore a number of the more common methods here.

$$S(I_1, I_2) = k;\ k \in \mathbb{R} \tag{2.1}$$

$$S(I_1, I_2) > S(I_3, I_4) \Rightarrow I_1 \text{ and } I_2 \text{ are more mutually similar than } I_3 \text{ and } I_4 \tag{2.2}$$

$$S(I_1, I_1) \geq S(I_1, I_2)\ \forall\ I_2 \neq I_1 \tag{2.3}$$

Pixel Difference Summation

These metrics are often the simplest, directly comparing and aggregating the difference in intensity for each element l of the image (pixel or voxel). The most basic form would be the negative sum of differences, shown in Equation 2.4 and Figure 2.1. It is common to apply a non-linear function to these differences, such that small regions with large differences outweigh larger regions with only minor discrepancies. A popular form is the negative sum of squared differences metric shown in Equation 2.5. Such aggregates are often normalized by the number of elements considered, N_l, to remove the effect of image size.

$$S_{SD}(I_1, I_2) = -\sum_l |I_1(l) - I_2(l)| \tag{2.4}$$

$$S_{SSD}(I_1, I_2) = -\frac{1}{N_l}\sum_l |I_1(l) - I_2(l)|^2 \tag{2.5}$$

While this category of metric can be very effective when used correctly, it is only applicable in very limited circumstances. It does not adapt well to any changes that render the images less than identical, such as deformations that may occur in soft tissue. Modalities sensitive to direction, such as ultrasound, likewise reduce effectiveness. If the images are of different modalities then this class of metric is not useful, as shown in Figure 2.2.

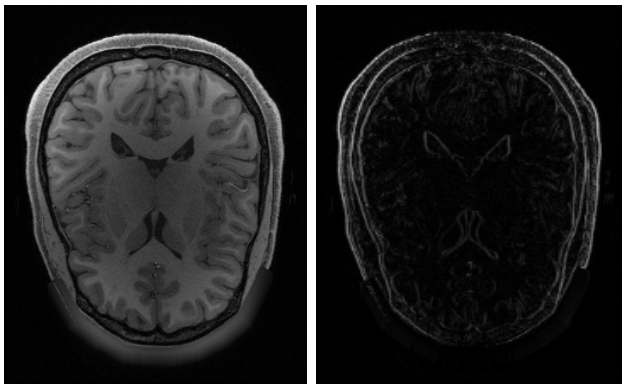

Figure 2.1 Monomodal pixel difference example, with I (left) and $|I - I'|$ (right), where I' is identical to I but offset horizontally by 2 pixels. Non-black elements in the right image are caused by misalignment, and the sum of their values is assumed to be proportional to the degree of misalignment. Images from the Human Connectome Project [8].

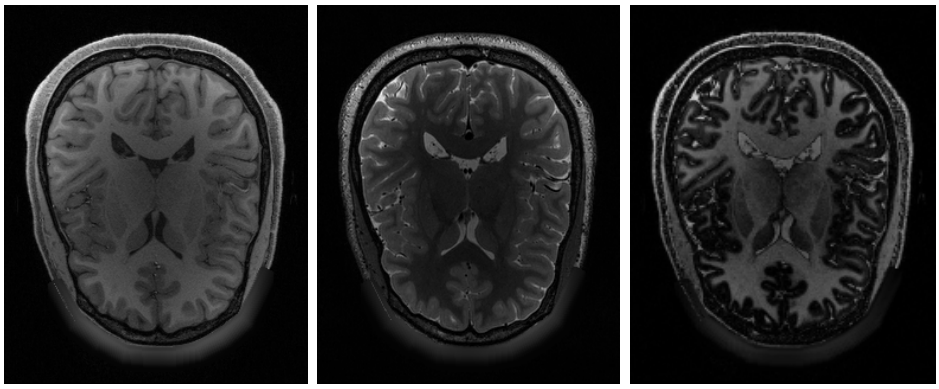

Figure 2.2 Multimodal pixel difference example, with I_1 (T1 MRI, left), I_2 (T2 MRI, centre), and $|I_1 - I_2|$ (right). I_1 and I_2 are aligned, and so we can see that the highlighted "error" in $|I_1 - I_2|$ is unrelated to alignment. Images from the Human Connectome Project [8].

Mutual Information

Rather than directly comparing element intensities, mutual information as a similarity metric measuring the individual and negative joint entropy of images' element values. In a slightly more intuitive phrasing, it is a measure of the information content of both images combined with how well the elements in one image predict the values of their counterparts in the other. It is assumed that in aligned pictures element values will have greater predictive power than in misaligned images.

$$S_{MI}(I_1, I_2) = H(I_1) + H(I_2) - H(I_1, I_2) \tag{2.6}$$

$$H(I) = -\sum_{v} p(I = v) \log(p(I = v)) \tag{2.7}$$

$$H(I_1, I_2) = -\sum_{v_1} \sum_{v_2} p(I_1 = v_1, I_2 = v_2) \log(p(I_1 = v_1, I_2 = v_2)) \tag{2.8}$$

Equation 2.6 is one of the standard forms of this metric [9]. The first two terms $H(I)$ measure the individual entropies of the images I_1 and I_2, as per Equation 2.7, based on the probability $p(I = v)$ that any given element of I will have value v, as calculated for each possible v. During registration the fixed image entropy term will not change, and the moving image term is generally expected to not change very much. It will decrease,

however, if a transform would homogenize the moving image. There are multiple ways this could occur, such as by scaling down the image excessively, or by moving portions of the image far outside the "working area" of the algorithm such that they are cropped out. Penalizing reduced entropy discourages these destructive transformations.

Alignment between images affects the joint entropy term $H(I_1, I_2)$, defined in Equation 2.8, which causes the lowest penalty when the predictive power of image element values is highest. High predictive power in this case refers to when for all possible values, if one selects all elements in one image with that given value, the values of the elements at corresponding locations in the other image have low entropy. Figure 2.3 illustrates this by plotting the corresponding element intensities for two different offsets of a posterized brain image.

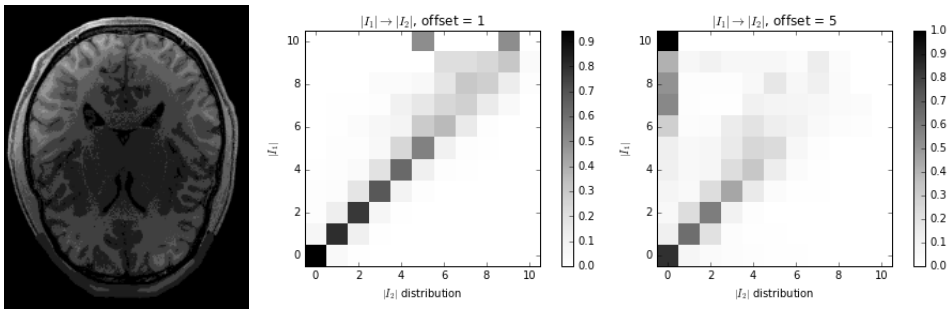

Figure 2.3 Mutual information example. A sample from the Human Connectome Project was posterized (left), dividing the full range of pixel values into to 11 non-overlapping subintervals of equal length, and then mapping all pixels to an integer value in $[0, 10]$ according to the subinterval membership of their original value. This image was then compared to a copy of itself that was offset one pixel (centre) and five pixels (right). The charts show, for a given $|I_1|$ (vertical axis), the distribution $|I_2|$ at corresponding element locations (horizontal axis), with the intensity of the grey value in a square indicating the likelihood of each possible $|I_2|$. The example results show that $|I_1|$ is a better predictor of $|I_2|$ (and similarly $|I_2|$ of $|I_1|$), and thus that mutual information is maximized, when the images are more closely aligned. Here $|I|$ is used as shorthand for "element value(s) within I".

Mutual information is a useful similarity metric for certain types of multimodal registration, and is far better than pixel difference metrics. However, it still relies on element intensities being individually informative without considering local context. As such it is potentially vulnerable to degraded performance or failure when used with modalities that

are directionally dependent, introduce significant noise and artifacts, or otherwise have non-homogeneous intensity for tissue types within a given image. This is a significant limitation given the ubiquity and utility of ultrasound imaging, for which all of these obstacles are a concern.

Gradient Alignment

Rather than focusing directly on pixel values, gradient alignment metrics calculate the gradient of both images and compare these derived images. For each location with high gradient magnitude (i.e., above some selected threshold m), the gradient angle is compared. This metric is highest when the edges of objects within the images are overlapping and aligned, and the metric is applicable for any pair of images in which object boundaries are consistently detected [10]. Figure 2.4 shows which features of the brain are highlighted by this approach.

$$S_{grad}(I_1, I_2) = \sum_{l|\nabla I_1(l)>m} cos(\angle\nabla I_1(l) - \angle\nabla I_2(l))^2 \tag{2.9}$$

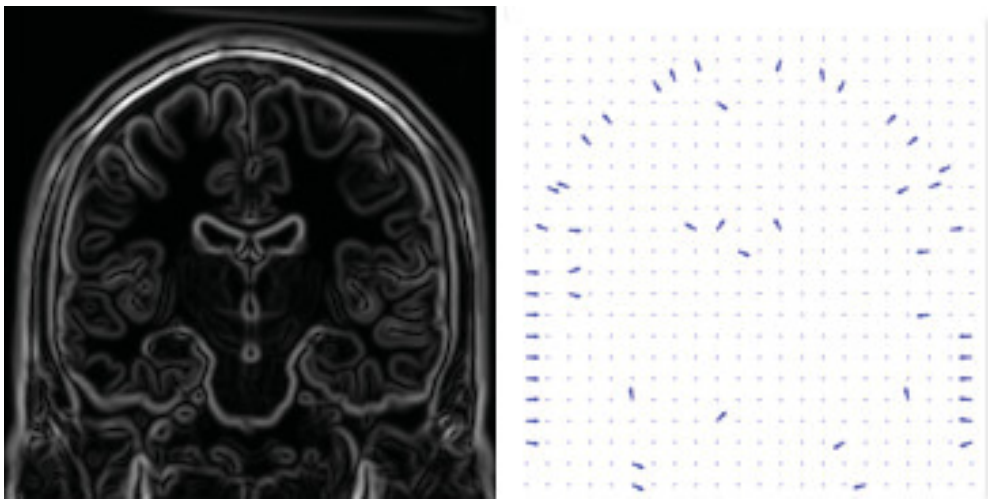

Figure 2.4 Gradient alignment example, showing magnitude (left) and orientation (right) of a single image. From [10].

Feature Matching

Feature or landmark matching differs from most other methods in that it does not globally consider all pixels. Rather, a feature detection algorithm is used to select points of interest ("features") within the images, and a matching algorithm is used to pair corresponding features to each other. The similarity metric is then determined by the distances between all corresponding points - most often the sum, or the sum of squares. Figure 2.5 highlights how corresponding features can have very different appearances between modalities.

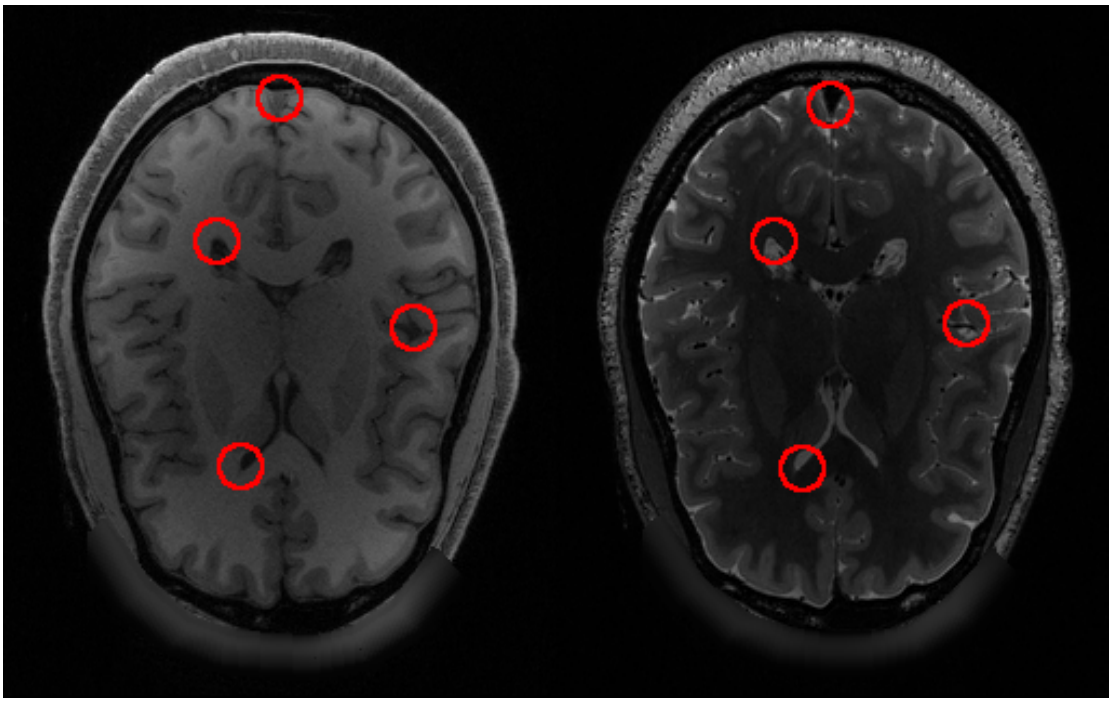

Figure 2.5 Landmark example. A definition of a feature's appearance in each modality allows the same feature to be localized in each image despite apparent visual differences, and once a set of linked locations have been selected (as in the above images) registration is dramatically simplified. MRI from HCP [8].

Maximally Stable Extremal Region Detection An extremal region [11] [12] is a set of connected pixels such that every pixel in the group has the same relation, either brighter or darker, when compared to all pixels along the group's border. A region of this type may be defined by a point belonging to the region, a threshold used to determine inclusion, and an indication of whether it is to be brighter or darker.

Stability, in this case, refers to the area change as a function of the threshold used to determine group membership. Therefore the maximally stable regions are those little to no area change over the widest range of thresholds.

Once a region has been detected an ellipse may be fit to its border, thereby defining an affine transform. MSER detection is affine covariant, and as with the Harris affine detector this reduces the effects of perspective when comparing detected regions. SIFT descriptors again may be generated from the transformed regions.

SIFT Feature Description The scale-invariant feature transform [13] allows a region of an image to be described in a form that is not affected by changes in scale, and is resistant to a degree to changes in illumination or viewpoint. The encoding algorithm has an associated method of region detection, but in this case alternative methods are used.

To encode a region of interest, it is first divided into sections. Each section is then further subdivided, and the direction of the gradient in each subsection is calculated. A histogram of gradient directions is then formed for each section, each with a fixed and equal number of bins. The final descriptor consists of the concatenation of these histograms.

2.1.3 Transforms

An image transform is a function that receives an image and produces a modified version of that image, mapping the elements of the output image to those of the input image. This is a specific application of the more general mathematical concept of a transform, which consists of any mapping that maps elements of set back onto itself [14].

Parametric vs Non-Parametric Transformations

If each element of a transform's output image has an associated location on the input image specifically and explicitly defined then the transform is called non-parametric, though in certain contexts it may have a more specific name (e.g., "optic flow map"). While versatile, the lack of constraints and the sheer number of variables that define such a transformation result in an extremely large and complex transform space. The difficulty of searching such a space means that these transform definitions are often ill suited to registration tasks.

The transform space may be simplified by parameterizing the transform function - specifying a method of mapping all elements using only a relatively small number of input parameters.

Rigid and Affine Transformations

Affine transformations are those in which a transformed element's location along each dimension is defined by a linear combination of its displacements along the original dimensions, applied homogeneously to all elements, ensuring any parallel lines in the original image remain parallel in the transformed image. The transformation may be composed of any combination of translation, rotation, shearing, or scaling along an axis.

An N-dimensional affine transformation may be fully defined by a $N \times (N+1)$ matrix. For a given transformation matrix M the relation between a location l_{in} in the initial image and location l_{out} in the transformed image is as shown in Equation 2.10. Additionally, two affine transformations may be combined as shown in Equation 2.11, constructing a temporary $(N+1) \times (N+1)$ matrix by appending a row to M_1 and then multiplying it by M_2. Applying the combined M_{cmb} has the same effect as applying both M_1 and M_2.

These transformations are useful when no significant local deformations are expected to have occurred between images, such as when aligning images acquired at the same time, or images of solid structures (e.g., implants, or bones in many circumstances). A specific

and even more restrictive type of affine transform, often called a rigid transform, is one in which no shearing or scaling occurs.

$$l_{out} = Ml_{in} \tag{2.10}$$

$$M_{cmb} = M_2 \begin{bmatrix} & M_1 & \\ 0 & \dots & 0 & 1 \end{bmatrix} \tag{2.11}$$

Spline Based Transformations

Spline interpolation is used to fit a polynomial to a series of fixed points. Each segment is controlled only by nearby points, where the number of points to consider is a predefined parameter.

As a transformation, this allows one to fully define image motion by specifying a series of control points with fixed positions and variable associated offsets. A spline is then fit to the offset values, allowing smooth (no discontinuities in optic flow or its gradient) offset interpolation between control points.

These transforms are appropriate when aligning images with local deformations that are smooth and continuous. This is applicable to most images containing soft-tissue in which no disjunction has occurred.

B-Splines B-spline based transforms are popular for image transformation due to the method's relative computational simplicity. The offset of each pixel is controlled by a small number of local parameters, and it is easy to identify all pixels associated with each parameter or vice-versa. This allows each region to be updated semi-independently, while ensuring global smoothness.

To perform the transformation, a regular grid of control points is defined that covers and extends beyond the boundaries of the image, and each control point is assigned a number of parameters equal to the dimensionality of the image. The offset at each pixel is then defined by Equation 2.12 [15], where $T_{bsp}(x, y)$ returns the coordinates in the transformed image corresponding to the coordinates (x, y) in the original image, n_d is the size of the control point grid in dimension d, and $\theta_{f,g}$ is the parameter vector associated with the control

point at grid coordinates (f, g). Functions B_{0-3} are the basis functions of the B-spline, and determine the degree of influence that nearby control points have on local deformation.

$$T_{bsp}(x, y) = \sum_{l=0}^{3} \sum_{m=0}^{3} B_l(u) B_m(v) \theta_{i+l, j+m} \tag{2.12}$$

$$\begin{aligned} B_0(t) &= (1-t)^3/6 \\ B_1(t) &= (3t^3 - 6t^2 + 4)/6 \\ B_2(t) &= (-3t^3 + 3t^2 + 3t + 1)/6 \\ B_3(t) &= t^3/6 \end{aligned}$$

$$\begin{aligned} i &= \lfloor x/n_x \rfloor - 1 \qquad & u &= x/n_x - \lfloor x/n_x \rfloor \\ j &= \lfloor y/n_y \rfloor - 1 \qquad & v &= y/n_y - \lfloor y/n_y \rfloor \end{aligned}$$

For images of a higher dimensionality the transform is adapted by adding another basis function term and nesting another summation for each new dimension, such as with the 3D form shown in Equation 2.13. The number of operations required grows as an exponential function of dimensionality, as does the amount of data per image.

$$T_{bsp}(x, y, z) = \sum_{l=0}^{3} \sum_{m=0}^{3} \sum_{n=0}^{3} B_l(u) B_m(v) B_n(w) \theta_{i+l, j+m, k+n} \tag{2.13}$$

$$k = \lfloor z/n_z \rfloor - 1 \qquad w = z/n_z - \lfloor z/n_z \rfloor$$

Spline based transformations are useful for medical imaging applications, as a large portion of human anatomy is soft tissue that deforms in a non-rigid and smooth manner (barring major disjunction caused by injury). B-splines are additionally useful in that local deformation is controlled by localized parameters, reducing overall complexity and allowing distinct segments of anatomy to deform in a semi-independent manner. Figure 2.6 shows an example of this sort of regionally semi-independent yet smooth deformation.

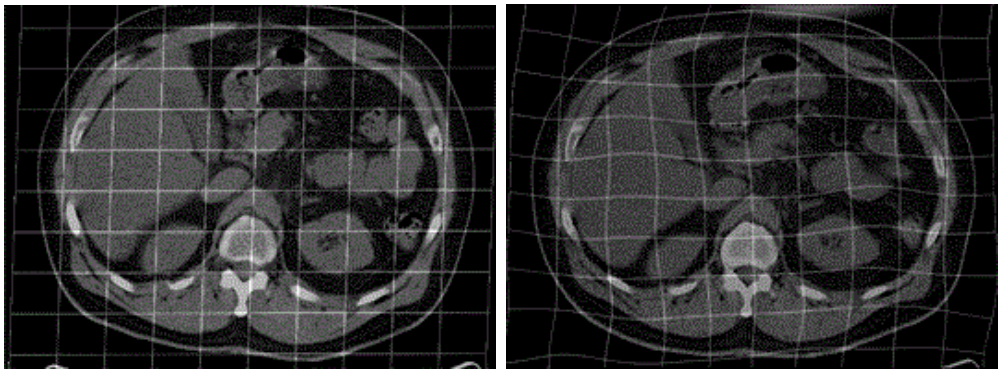

Figure 2.6 Example of B-spline transformation (from [16]). Original image (left) is shown after deformation (right). Control points fall on the intersection of grid lines, while the grid lines illustrate deformation between points. This example is purely illustrative, rather than computed from HCP data.

Resampling

Often a transform will map elements of the output image not to exact elements of the input image, but to locations on the borders between elements (i.e., non-integer pixel addresses). In these cases the output element intensity must be derived from the intensities of the elements near its mapped location.

Nearest Neighbour The fastest method of interpolation is to simply select the element nearest to the indicated location. However, this may noticeably change the shape of fine-scale image details. This can be inappropriate for medical applications, in which an image's low resolution may mean that a single element represents a large region of space.

Bilinear and Trilinear Interpolation These methods consider the 2^D elements (given dimensionality D) nearest to the target location and perform iterative linear interpolation along each dimension. The order in which dimensions are processed does not affect the result, and the algorithm may be simplified to a single step equation.

Higher-dimensional interpolation is an extension of the simplest form, one-dimensional linear interpolation. For a single-dimensional "Image" $I(x)$, with known values $I(x_1)$ and $I(x_2)$, $x_1 < x_2$, linear interpolation may be expressed in the following parametric form:

$$I(x) = (1-t)I(x_1) + tI(x_2), \tag{2.14}$$

$$t = (x - x_1)/(x_2 - x_1) \in [0, 1], \qquad x_1 \leq x \leq x_2.$$

The formulae for $D = 2$ are shown in Equation 2.15 [17], assuming (x, y) is within a rectangle with corners at $(x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2)$ for which the values are known.

$$\begin{aligned} I(x,y) = (1-t)(1-u)I(x_1,y_1) + t(1-u)I(x_2,y_1) \\ +tuI(x_2,y_2) + (1-t)uI(x_1,y_2) \end{aligned} \tag{2.15}$$

$$t = (x - x_1)/(x_2 - x_1) \in [0, 1], \qquad x_1 \leq x \leq x_2$$

$$u = (y - y_1)/(y_2 - y_1) \in [0, 1], \qquad y_1 \leq y \leq y_2.$$

The $D = 3$ case is defined similarly, with three parameters $x_1 \leq x \leq x_2$, $y_1 \leq y \leq y_2$ and $z_1 \leq z \leq z_2$.

This interpolation reduces spatial distortion and edge-jaggedness that may be caused by nearest-neighbour methods, while remaining quite fast. The main drawback is that it can blur sharp region boundaries from the original image.

More advanced interpolation There are many other forms of interpolation, but most of them are significantly slower. This makes them less suitable for speed-sensitive applications involving iterative updates and therefore they will not be discussed in detail here.

The general form of these methods is similar to an upgraded linear interpolation, fitting more complex functions (often higher-order polynomials) to larger patches of pixels. Additionally, as with most applications involving fitting functions to measured data, there are some recent machine learning approaches to the problem [18].

2.1.4 Regularization

Image registration often requires additional constraints to become a well-posed problem [19]. If the parameters of the transform are not limited then it is often possible to find an arbitrarily high number of equally valid solutions.

Regularization helps by augmenting the similarity metric with a penalty term to form a single measure of registration quality. The penalty term depends on the type of transform being considered, as it is a function of the transform parameters, and if properly implemented it guarantees that the resulting function will have a single maximum.

Global regularization Global regularization considers all parameters of the transform at once, and is generally used to influence large-scale movements that affect most or all of the image. This often takes the form of penalizing movement away from some starting position, as for many applications it is possible to perform a simple pre-registration to find a transform that, while incorrect, is very likely to be "near" the correct solution.

Local regularization For many non-rigid transforms, one may consider local regularization penalties based on the relative motion of nearby sections the moving image. For many non-rigid transforms parameters can be localized, and a local regularization penalty measuring such divergence can be defined as a function of the parameters of neighbouring regions. The overall regularization term then includes a summation of local penalties for all valid region pairs.

Physical Basis For both global and local regularization, it often possible to base penalties on known properties of the subject of the images. For example, in medical imaging the elasticity, strength, and other properties of the tissue can be used to define expected, reasonable, and impossible levels of deformation, allowing for accurate restrictions to be placed on image movement.

2.1.5 Optimization

Once an objective function has been defined, through some combination of similarity metric and regularization term, it is necessary to find the transform parameters which produce the maximum possible value. This process is called optimization.

Simple objective functions may have a closed form solution that can be found in some small number of steps. However, many objective functions complex enough to be useful in real world applications must be solved via an iterative algorithm.

Global

Global optimization seeks to find the absolute best parameters for the objective function. These solutions are often more difficult and computationally expensive than the alternative. The theoretically ideal form is often too expensive to be practical, and many implementations use approximations of some component of the process to reduce cost.

Examples include exhaustive search, simulated annealing [20], and Gibbs sampling [21].

Local

These methods find some local optimum. It may not correspond to the best possible value, but will find a set of parameters such that any other "nearby" parameter set will reduce the objective function. These approaches are often less computationally expensive and are therefore capable of handling larger datasets.

Examples include gradient descent [22] and expectation maximization [23].

2.2 Machine Learning

2.2.1 Introduction

"Machine learning" is the name for a broad category of data-driven modelling techniques. Data-driven refers to the fact that these approaches do not seek to directly analyse and recreate the mechanism producing the data of interest, but rather to fit a model directly to the data itself.

2.2.2 Common Terms and Methods

Learning and Supervision

The most common application is supervised learning, in which each data-point in the training set has both input variables (also called "features") and corresponding output variables (also called "target variables" or "labels"). The system's purpose is to accurately predict the output variables when supplied with the input variables, and eventually to generate accurate output for new inputs which would otherwise not have such. The act of adjusting the chosen model's parameters to be able to do this is called "training" or "learning".

Unsupervised learning techniques cover situations where the data has no available labels, and one seeks instead to uncover non-obvious relationships. In these cases learning most often takes the form of clustering - organizing the data-points into some number of discrete groups with similar properties, and setting model parameters such that new datapoints may be automatically clustered into one of these groups.

The intersection of these two scenarios is called semi-supervised learning, in which labels are available but only for some of the training data.

Overfitting

A common risk when training complex models with finite datasets is the potential to "overfit" the model. Overfitting refers to setting the parameters of a model such that it will very accurately predict labels for any given entry from the training data, yet will perform poorly for any new data. This can be conceptualized as the system memorizing specific examples that it is exposed to rather than learning the general trend relating inputs to outputs. The risk of overfitting is increased for smaller datasets and for higher-capacity

models (where model capacity refers to how much information can be stored in the model parameters, and is closely related to the number of parameter variables).

There are many different approaches to preventing overfitting, and many of those are specific to the type of model being trained. However, one mitigation technique applicable to all models is that of data segregation.

Regularization

One common method of avoiding overfitting is "regularization", a restriction on a model which seeks to limit its capacity/complexity. While it is possible to impose a strict limit, regularization more often takes the form of a penalty term representing model complexity. This penalty is balanced against the model's accuracy, ensuring that the model may only become more complex when it will commensurately increase performance.

The balance between accuracy and complexity is not easy to optimize, however, and tends to introduce one or more hyperparameters that must be manually determined.

Data Split

Standard practice for most machine learning applications is to divide the available data into three partitions: the "Training" set, the "Validation" set, and the "Testing" set. The training and validation sets are used while the model is learning, while the testing set is reserved for afterwards.

The training set is used to directly train the model in the usual manner, updating model parameters based on its contents. Throughout or after learning, the validation set is used to evaluate the model's accuracy. This can be used to detect overfitting if the accuracy on the training and validation sets begins to diverge, as well as to adjust model hyperparameters and observe the resulting performance change.

After the hyperparameters have been finalized, and the model has been trained, the testing set is used to judge the final accuracy. It is necessary to keep some portion of the data unused in this way if one wishes to have a true measure of accuracy. While overfitting is not a risk for the validation set, it is possible that the model selection and hyperparameter tuning will have some bias from any data used in developing or refining the model. It is therefore a good practice to segregate some data, such that the model cannot in any way

be affected by it other than through the true underlying pattern that would produce all gathered data.

Cross-validation

Cross-validation is a modification of the above method of estimating the accuracy of a machine learning system. Rather than subdividing the non-testing-set portion of the data into training and validation sets once, the subdivision is repeated multiple times to produce multiple temporary sets. Each such training/validation set pair is then used to train and evaluate the performance of the system, producing multiple different accuracy estimates. These are then averaged together to produce a final estimate that is closer to the true system accuracy than the result of any single subdivision could be expected to be.

While this method may produce superior estimates compared to a single training and validation split, it is vastly more costly. The system must be fully trained from the beginning for each new subdivision, and to usefully improve estimation accuracy many such iterations are often necessary. Consequentially, cross-validation is mainly suitable for simpler and faster to train systems.

Regression vs Classification

Most supervised machine learning tasks fall under either "regression" or "classification". Regression refers to cases where the output variables are continuous, and one attempts to use an ML model to predict their exact values. Classification is the class of problems where all datapoints are labelled as members of some finite number of discrete groups.

It is easy to adapt most regression models to work for classification tasks by using one-hot encoding - assigning a separate output variable to each class, and then for each datapoint setting the variable corresponding to the true class to be one and all others to be zero. The algorithm will then give a decimal score for each category, and selecting the highest score will give the final category.

It is also possible to do the reverse, and adapt a classification algorithm to a regression task, but accuracy is necessarily limited. This may be accomplished via "binning" - subdividing the range of possible values into several "bins", and treating each as a category. Predicting the correct category will then give an approximation of the correct output value.

Unsupervised methods are generally limited to classification tasks, as there are no targets values for regression algorithms to predict.

Gradient Descent

Gradient descent is a method of training many different weight-based models via a series of iterative improvements. This is necessary for models for which there is no closed-form solution, or for which such a solution cannot be feasibly calculated. It is first necessary to define an error function as a function of the model weights and the datapoints. The process then consists of repeatedly calculating the gradient of that error function and then updating the model weights to move them closer to the error function's minimum.

To allow this process to work with large datasets, it is often necessary to only use a small subset of the training data for any given update. This subset is called a "batch", and each new batch is constructed by randomly sampling the training set for the appropriate number of datapoints.

Stochastic Gradient Descent For some models it is not possible to mathematically derive the error as a function dependant (at least partially) on the model weights, but it is possible to estimate it using the training data. This is done by applying the model to the training dataset and comparing its predictions to the known true labels, which is used as the estimated error. It is then necessary to calculate the change in model weights required to eliminate this error, to be used as the estimated gradient. Fortunately this is relatively simple for most models.

After the gradient has been estimated it is most often scaled according to some constant "step size". This prevents the algorithm from passing over a local error minimum repeatedly without coming usefully close to it. It additionally helps to smooth noise introduced if batch training is used by limiting the change that may be caused by a single batch.

2.2.3 Classic Machine Learning

Linear Regression

One of the simplest ML methods, this performs regression by fitting a linear N-dimensional surface to the data, where N is the sum of input and output vector dimensionality. If one wishes to predict a specific answer, rather than a relation between vector elements, the target is therefore limited to a single scalar value. To predict multiple targets one may isolate and perform a separate regression for each target variable, but information on correlations between these variables will be lost.

The linear regression model may be expressed as in Equation 2.16 for output y, input vector $\vec{x}$, and weight vector $\vec{w}$, where $\vec{x}$ and $\vec{w}$ are the same size. In most cases $\vec{x}$ is augmented by appending a '1', allowing $\vec{w}$ to have a constant term.

$$y = \vec{w}^T \vec{x} \tag{2.16}$$

The weights may be found by minimizing the error function, which measures difference between the known output values and the values predicted by the model. Constructing $\vec{y}$ and matrix X by appending together all y and $\vec{x}$ datapoints, the error may be expressed as follows. Equation 2.17 includes L2 regularization, and Equation 2.18 includes L1 regularization.

$$Err(\vec{w}) = \|(\vec{y} - X\vec{w})\|^2 + \lambda\|\vec{w}\|^2 \tag{2.17}$$

$$Err(\vec{w}) = \|(\vec{y} - X\vec{w})\|^2 + \lambda \sum_i \vec{w}_i \tag{2.18}$$

In many cases, the solution may be found quite quickly.

$$\vec{w} = (X^T X + \lambda I)^{-1} X^T Y \tag{2.19}$$

If the dataset is too large to perform the necessary operations on the entire set at once, it is also possible to perform gradient descent. The per-step weight update is given by Equation 2.20.

$$\Delta\vec{w} = \alpha(X^T Y - X^T X\vec{w}) \tag{2.20}$$

Decision Trees

Decision trees are a method of classification (and occasionally regression) that differs from the other approaches discussed so far in that they do not consist of specific models with weights to be tuned. Rather, they describe a general structure and an algorithm to creating a model with that structure, unique to the problem at hand.

A decision tree comprises a collection of nodes, as shown in Figure 2.7. Each node consists of a test for specific input variable and two branches conditioned on the test result. Each branch can point to a specific class or to a new node. To classify a data point, the algorithm begins with the first ("root") node, performs the test at that node, and continues along the appropriate branch. This is repeated at each node until a branch to a class is reached, at which point that class is assigned to the data point.

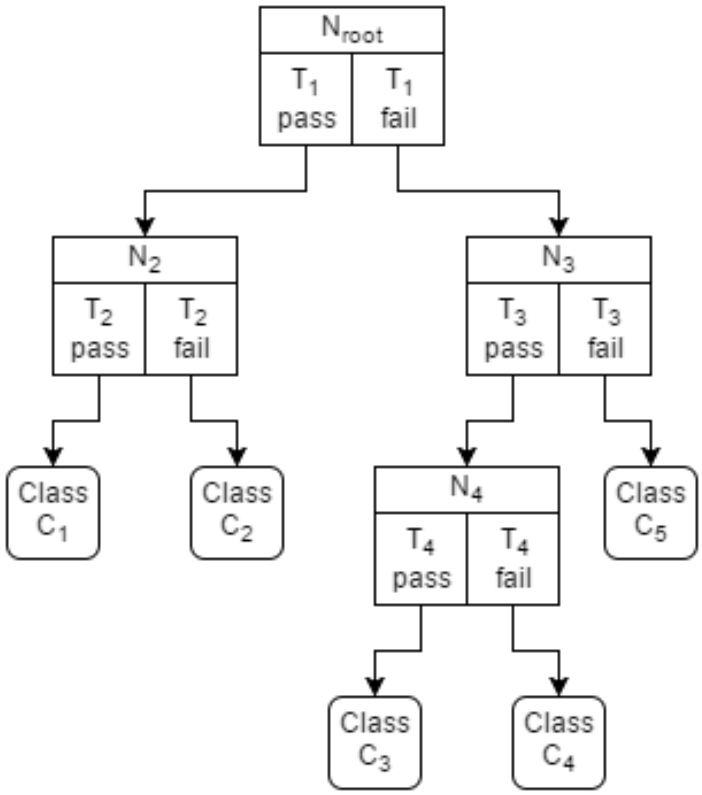

Figure 2.7 Example of Decision Tree structure.

Growing a Tree A decision tree is constructed recursively one node at a time, with the test at each node chosen to maximize information gain (minimize entropy) in the dataset when split according to that test. To avoid overfitting the training data is split into a "growing" set and "pruning" set. The first set is used to create an initial tree as shown in Algorithm 1. The second set is then used to remove superfluous nodes as shown in Algorithm 2, improving performance. Equation 2.21 gives the conditional entropy for a given test T and dataset D. In this context distinct tests are those that result in a different divisions of D.

Input: *Set D of data points and their classes*
if *All data points belong to class C* **then**
    Return class C;
else
    Evaluate $H(D|T)$ for all distinct tests;
    Select test $T_{best} = \text{argmax}_T\, H(D|T)$;
    Split D into D_{pass} and D_{fail}, containing the points that pass or fail T_{best};
    Grow decision tree R_{pass} using D_{pass};
    Grow decision tree R_{fail} using D_{fail};
    Construct root node containing T_{best}, branching to root nodes of R_{pass} and R_{fail};
    Return root node, R_{pass}, and R_{fail};
end

Algorithm 1: Growing a decision tree

Input: *Full tree R, set D of data points and their classes*
foreach *node N* **do**
    Create R_N by removing N and replacing it with class C, where C is the most prevalent class among samples from D that would reach n;
    Test accuracy of R_N using D;
end
Test accuracy of R using D;
if *All R_N less accurate than R* **then**
    Return R;
else
    Replace R with most accurate R_N, repeat;
end

Algorithm 2: Pruning a decision tree

$$H(D|T) = \frac{|D_{pass}|}{|D|} H(D_{pass}) + \frac{|D_{fail}|}{|D|} H(D_{fail}) \tag{2.21}$$

$$H(D) = -\sum_{\forall C} \frac{|D_{class=C}|}{|D|} \log \frac{|D_{class=C}|}{|D|} \tag{2.22}$$

Ensemble Methods

Different models may be combined to form an ensemble model, often with improved accuracy. This generally consists of separately training multiple models and then combining their predictions, either by voting in the case of classification or by averaging in the case of regression.

2.2.4 Deep Learning

Perceptrons

To explain deep learning and neural networks, it is first necessary to explain it's fundamental building block: the perceptron. On it's own a perceptron is a simple linear classifier, very similar to the model discussed in section 2.2.3. The difference is that the output is passed through a non-linear threshold function. This limits a single perceptron to binary classification, but by introducing non-linearity it allows multiple perceptrons in concert to model more complex non-linear functions.

Equation 2.23 shows how the label y is predicted for input x, input weights w, and activation function a. Figure 2.8 shows this same information visually.

$$y = a(\vec{w} \cdot \vec{x}) \tag{2.23}$$

Training A perceptron is trained by stochastic gradient descent, updating the model weights based on any incorrect predictions made on the training set. The change in weights δw for each incorrectly labelled datapoint is given by Equation 2.24, for input vector $\vec{x}$, label y, and step size α.

$$\Delta w = \alpha y \vec{x} \tag{2.24}$$

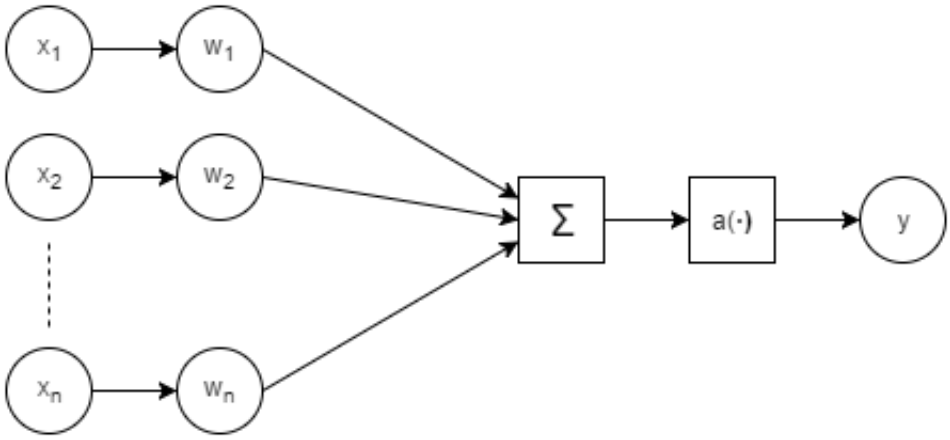

Figure 2.8 Perceptron structure.

Using this simple update step, a perceptron will eventually find the correct weights for any linearly separable dataset.

Activation Functions The activation function is used to introduce non-linearity to the output. While of limited utility for a single perceptron, it allows perceptrons to be combined into larger and more complex systems. If the activation function were linear then any combination of perceptrons could be reduced to a single larger perceptron with altered weights.

With non-linear activation functions, chained perceptrons become a more complicated system. There are a variety of commonly used functions, though they often resemble either a rectified linear unit (ReLU) [24] or a sigmoid [25]. Actual ReLU functions are often replaced with leaky ReLU [26] to avoid the perceptrons getting "stuck" during training [27].

Neural Networks

A neural net at its core consists of a large number of perceptrons working together. Specifically, neural nets are structured as a series of layers connected sequentially or in parallel. Each layer consists of some number of perceptrons, all operating simultaneously on the data output by the previous layer(s). The first layer uses the input vector directly, and the

final layer gives the predicted labels as output. The layers between these two, those with connections only to other layers' perceptrons, are called "hidden layers".

There are many kinds of layers, with each kind dictating a specific method of reading its input from the output of the previous layer(s). These are discussed in more detail in Section 2.2.4. Nets with multiple hidden layers are called "deep neural nets", with their training and use being "deep learning".

Calculating the output of a neural network is also referred to as performing a "forward pass". As the input of each layer is dependant on all previous layers, data is propagated "forward" from the input layer to the output layer.

Input: Input vector x_{in}
Set output of input layer L_{in} equal to x_{in};
foreach *Layer* $L \neq L_{in}$, *in ascending order* **do**
  foreach *Node* $N_i \in L$ **do**
    Calculate output $o_i = a(\vec{w}_i \cdot \vec{x}_i)$ (Eq. 2.23);
  end
end
Return output of final layer as predicted label: $\vec{y} = \vec{o}_{out}$;
If training, also return full $\vec{o}$;

Algorithm 3: Performing prediction with a neural net

Gradient Descent and Back Propagation Neural networks are trained using stochastic gradient descent, using the usual method briefly discussed in section 2.2.2. However, as the net is composed of a individual perceptrons, the implementation is in some ways similar to performing SGD on each individual perceptron. Error is calculated as usual for those in the final layer, and then "blame" is partitioned according to the contribution of each perceptron used as input by that layer. Error and updates are "back-propagated" from the final layer to the input layer in this manner.

Layers

The arrangement of each layer's perceptrons and the inter-layer connections are free to take any form, but there are a number of commonly recurring layer structures.

Notation for weight values is $w_{to,from}$

Input: Input vector $\vec{x}_{in}$, output vector $\vec{y}_{gt}$
foreach *Node* $N_i \in L_{out}$ **do**
  Find error $\delta_i = o_i(1 - o_i)(y_{gt,i})$;
end
foreach *Layer* $L \neq L_{out}$, *in descending order* **do**
  foreach *Node* $N_i \in L$ **do**
    Find "blame" for N_i, considering all nodes $N_k, k \in K$ that use o_i as input: $\delta_i = o_i(1 - o_i) \sum_{k \in K} w_{k,i}\delta_k$;
  end
end
Return the weight correction vector $\vec{\delta}$

Algorithm 4: Performing back propagation for a neural net

Input: Large number of x_{in} and y_{gt} pairs, step size α
Randomly initialize weights;
while *Stop condition not met* **do**
  Randomly sample pair from x_{in}, y_{gt};
  Use Alg. 3 to find full $\vec{o}$;
  Use Alg. 4 to find full $\vec{\delta}$;
  foreach *Node* N_k *to* N_i *connection* **do**
    Find weight update $\Delta w_{i,k} = \alpha\delta_i o_k$;
  end
  Update weights $\vec{w} \leftarrow \vec{w} + \vec{\Delta w}$;
end

Algorithm 5: Training a neural net

Fully Connected A fully connected layer is organizationally simple. The input of each node of the FC layer is connected to every output the previous layer, resulting in $N_{out,i-1} \times N_{out,i}$ weights associated with the layer. The arrangement of these elements, including inputs, outputs, weights, and activation functions, is shown in Figure 2.9.

While highly expressive, the large number of weights can make training difficult and time consuming. Additionally, the format does little to take advantage of structured data.

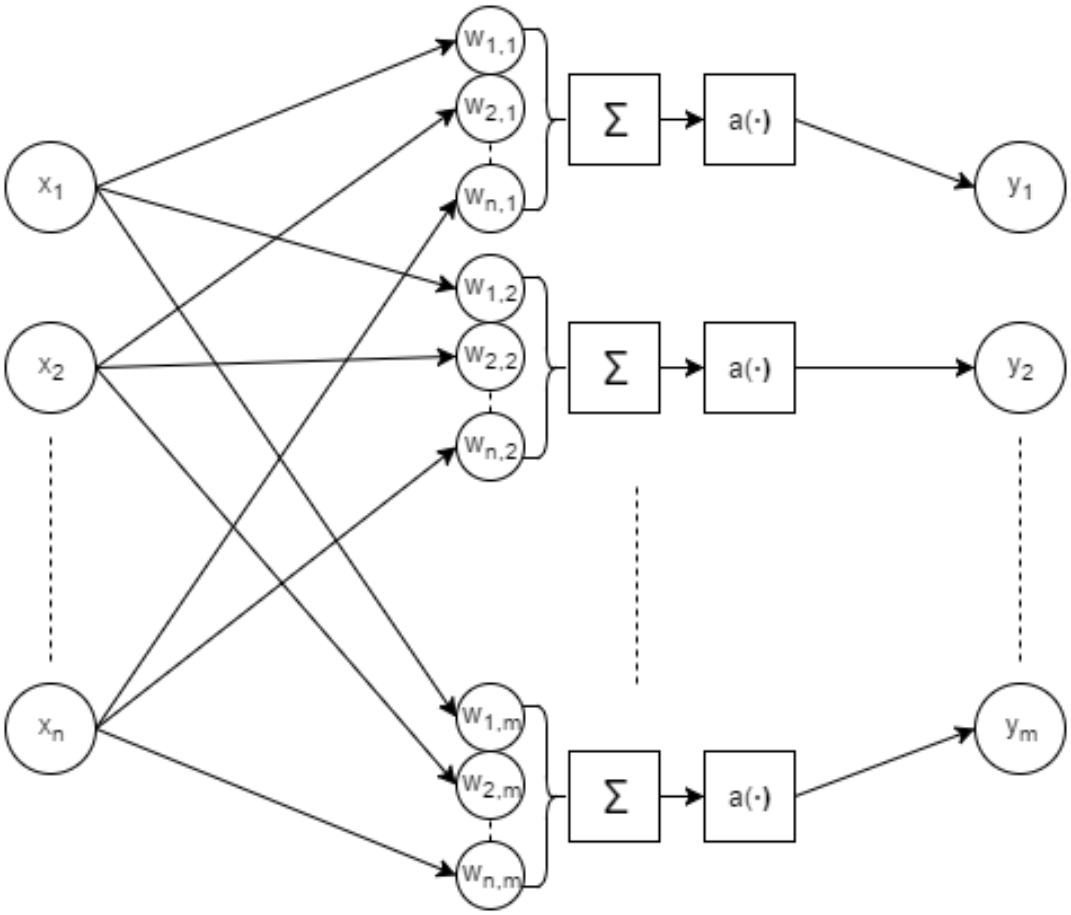

Figure 2.9 Fully Connected Layer structure.

Convolutional The elements of a convolutional layer are connected so as to take advantage of structured data, where the same patterns may occur in different clusters of "nearby" inputs. This is most often applied to images, in which objects with consistent local structure may be present at any location. For image data the implementation is nearly identical to the image processing convolution operation.

The layer's weights are grouped into multiple "kernels", as shown in Figure 2.10, where each kernel defines the way a node would apply those weights to a subregion of the structured data with a specific shape. Thus each node is connected to specific subregion, and the connection weights are shared for every node associated with a specific kernel. For an image these subregions are selections of pixels with a given spatial relation.

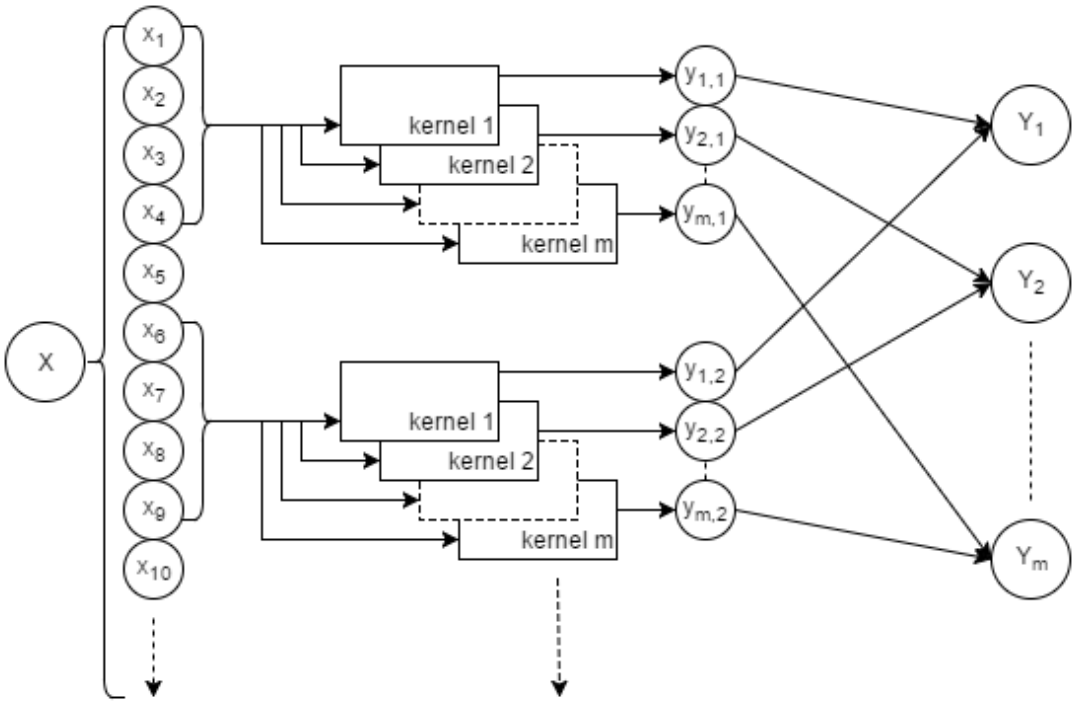

Figure 2.10 Convolutional Layer structure.

Each kernel is applied at regular intervals, called the "stride", along each dimension of the structured data. As a result the output nodes associated with a given kernel will also represent structured data, with the relation being defined in a space of the same dimensionality as the input data (i.e., a 2D image will result in another 2D image, though

the resolution may differ). An example of the application of a single kernel is shown in Figure 2.11 - note how the high stride results in a reduced number of outputs relative to the number of inputs.

If the kernel would not fully overlap the input data at a specific point there are two options. The first is to ignore that point, an option often referred to as "cropping". This reduces the output size, and can reduce the influence of datapoints near the edge of the input. The other option is to "pad" the image, filling in some value for the missing data. This avoids the aforementioned limitations of cropping, but can waste resources processing fabricated datapoints that add no new information.

For computational efficiency and simplicity, most deep learning libraries require that all kernels for a single layer be the same size and map to contiguous rectangular regions. However some allow "dilated" kernels, in which a rectangular kernel is defined but only the entries on some regular grid within the kernel are used, with the other weights assigned zero and ignored in calculations.

Pooling Pooling layers are not perceptron based, rather act as fixed filter layers used to reduce the dimensionality of structured data. This is accomplished by organizing the inputs into equally sized groups, with one outupt assigned to each group, and for each group setting that output equal to one of the inputs selected according to the specific pooling type. Common criteria are maximum or minimum values.

During back propagation all nodes connected to a pooling layer output are treated as if they were directly connected to the "active" output it had selected from the previous layer, ignoring all "inactive" elements. As a result, pooling layers add very little to a network's training time.

Pooling layers are most often used in conjunction with convolutional layers. They allow information to be relayed regarding the presence and general location of data activating the conv filters, while significantly lowering the total number of connections needed.

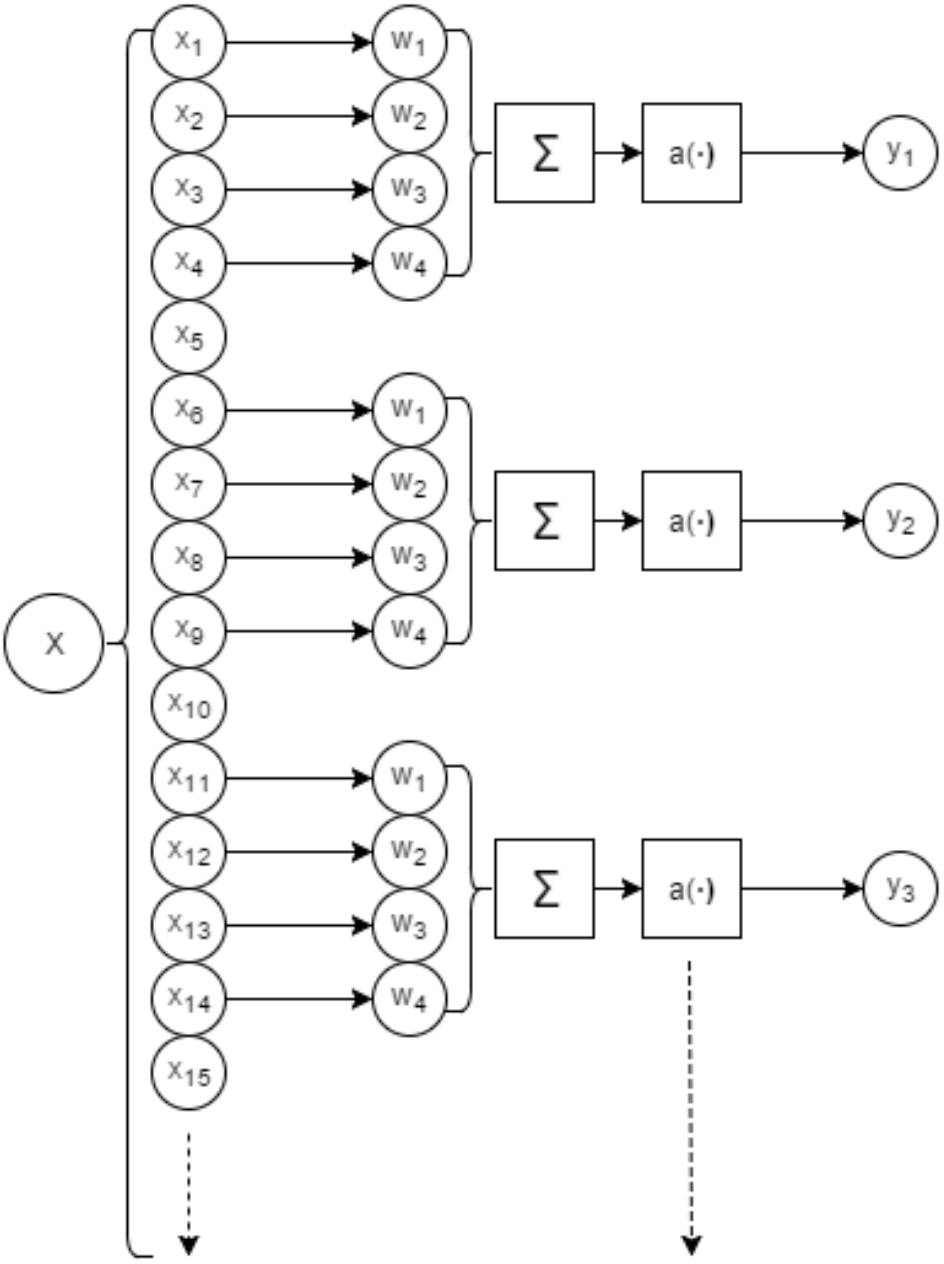

Figure 2.11 Application of a single kernel.

2.2.5 Deep Learning in Medical Image Registration

While the focus of this thesis is on the use of deep-RL for medical image registration, for the sake of completeness we will include a review of certain non-RL deep learning approaches to registration. The proliferation of deep learning in the field makes a full review beyond the scope of this thesis, but for more information one may address the 2019 survey paper by Haskins et al. [5]. For an overview of deep learning in the broader field of medical image analysis, including registration and other applications, see the 2017 survey paper by Litjens et al. [28]. More detail on older registration methods pre-dating the popularity of deep learning may be found in the 1992 survey paper by Brown [2].

Early applications of deep learning in registration often used it to learn application-specific feature representations, exploiting deep learning systems' pattern detection abilities, which are then used as a component of the similarity metric in a more traditional registration system. One of the first such approaches was by Wu et al. [29] in 2013, in which an architecture known as "stacked convolutional Independent Subspace Analysis" [30] was used to learn feature detection filters for brain MR images. These filters were then inserted into existing feature-based registration methods (e.g., HAMMER [31]), and the superior application-specific encoding learned by the network led to very good performance relative to other popular methods of the time.

Moving forward, researchers began to use deep learning systems to directly learn similarity metrics. Trained using datasets of images of known misalignment (which is easily generated from a dataset of pre-aligned images) deep learning systems were found to be able to accurately predict degrees of image alignment. This was especially effective for multimodal similarity metrics, relative to prior methods. Simonovsky et al. [4] presented an early version of this method, in which they trained a convolutional neural net to estimate alignment of T1 and T2 MRI volumes. The learned similarity metric would then be optimized with a standard gradient descent algorithm, resulting in a system that outperformed other systems using popular multimodal similarity metrics of the day.

Embracing deep learning even more fully, many researchers began creating architectures to estimate transform parameters directly from the raw image input, absent any recognizable traditional registration framework. There have been many different approaches to this, using a wide variety of architectures, modalities or modality pairs, and transform types. As a result, a usefully detailed review of such options would be quite voluminous. Some

common features, however, are that successful implementations tend to be much faster than almost any other alternatives, as they predict the transform in a small and finite number of steps rather than through some optimization algorithm. A downside, though is that it can be very difficult to accurately find transforms with many parameters.

Miao et al. [32] are notable for being the first to publish a successful implementation of a fully deep learning registration system. They use pre-registered radiograph and X-ray data to generate image pairs with known rigid transform, and then use that data to train convolutional neural net regressors to predict transform parameters from image data.

Chapter 3

Reinforcement Learning and Deep Reinforcement Learning

3.1 Problem Structure

Reinforcement learning (RL) is the field of machine learning that focuses on iterative decision making, and seeks to address the category of problems in which an agent must take iterative actions based on the observation of some environment that is in turn affected by those same actions. One defines the desired task by creating a reward function. RL algorithms then seek to maximise the cumulative reward the agent receives, and so while it is necessary to define some measure of success for a given task, the control policy itself is learned.

3.1.1 State

The state vector of a reinforcement learning problem is comparable to the input vectors of a standard machine learning problem. However, in an RL problem successive state vectors are observations of some persistent environment that is influenced by selected actions. Most algorithms consider the initial and resulting state of each step while learning.

3.1.2 Action

As the state corresponds to input, the action of a RL problem corresponds to the output vector of a standard machine learning problem. The action is often a single discrete value selected from a pre-defined list, akin to classification, but regression-like continuous control is also possible.

3.1.3 Reward

The reward function describes the agent's success or failure, thus defining the reward function serves to define the agent's task. It may be derived from the state, or from some part of the environment not directly observed. However it is defined, one must ensure that the function itself and any associated parameters cannot be directly changed by the agent's actions, or else agents are prone to "short-circuiting" and tuning the function to always give the maximum reward regardless of selected action or environmental observations.

3.2 Predicting Future Value

As the reward function measures success, the optimal policy is one that selects the series of actions that maximise the cumulative reward received. The agent must therefore have some method of predicting the action at each step that will maximise future rewards, which in turn necessitates estimating (directly or indirectly) the total future value that will result from each potential action. This is generally accomplished with a value estimation function that the learning process tunes to reflect the initially unknown reward function and environment dynamics.

It is possible to find an optimal policy without directly modelling a value function. However, approaches that attempt find policies without explicitly predicting value tend to fall outside of what is traditionally considered reinforcement learning [6].

Multiple optimal policies exist for many problems, but finding a general descriptor for all such policies is significantly more complex than finding a single policy, and there are few if any practical applications where it could improve an agent's performance.

3.3 Bellman Optimality Equations

When deriving the value function for an RL system, the Bellman optimality equations [6] are a useful recursive relation. They establish necessary and sufficient conditions for ensuring an optimal control policy when selecting the maximally valued action.

Equation 3.1 concerns the state-action value function $q_*(s, a)$, which gives the expected value of taking action a while observing state s. Equation 3.2 has a similar form but concerns the state value function $v_*(s)$, which gives the value based on the observed state alone, assuming that the best action will be selected. Both are defined as the expected value of the reward r resulting from a, plus the maximum value of the resulting state s' modified by some discount rate $\gamma \in [0, 1]$. The subscript $*$ indicates that $q_*(s, a)$ or $v_*(s)$ are the true values under the optimal policy. Without this subscript, $q(s, a)$ or $v(s)$ indicate an estimate of those values under some current policy. As mentioned previously, the goal of many RL systems is to find an accurate method of estimation such that $q(s, a) = q_*(s, a)$ and $v(s) = v_*(s)$.

$$q_*(s, a) = \mathbb{E}[r + \gamma \max_{a'} q_*(s', a')] \tag{3.1}$$

$$v_*(s) = \max_{a} \mathbb{E}[r + \gamma v_*(s')] \tag{3.2}$$

The discount rate is a hyperparameter used to control the tradeoff between current and future rewards. High-γ systems prefer to maximize future cumulative value even at the expense of immediate rewards, while low-γ systems will do the opposite. γ may be thought of as accounting for the uncertainty (and therefore unreliability) of future rewards or, depending on the application, as representing the opportunity cost of delayed gratification.

Which form of the equation is preferable depends on the exact learning method and target application.

3.4 Exploration and Exploitation

Unlike more traditional machine learning approaches, reinforcement learning does not directly make use of large sets of static data. The large potential state-space and interactive nature of the problems make it impractical to pre-generate or otherwise collect state-action-reward data. Rather, a reinforcement learning system will improve itself using the data it observes while interacting with its associated environment, real or simulated.

As the system learns during normal operation, it is necessary to balance the competing goals of exploration and exploitation. Exploration refers to taking actions with unknown consequences to examine new areas of the state space, thus gathering new information and hopefully improving the accuracy of the internal model, while exploitation refers to using the existing model to choose the predicted best actions and maximize rewards.

Too much focus on exploitation prevents the system from observing an adequate variety of data, and tends to result in a policy that appears locally optimal but overall is poor. Excessive exploration, on the other hand, can waste resources exploring low quality states. The state-action space of reinforcement learning problems is often quite large with only a small subset of it leading to desirable outcomes, and even with unlimited time to explore the entirety of that space an agent's information capacity is limited. It is counter-productive to waste space remembering how to deal with those regions in detail, as a properly trained agent should avoid them altogether. A common approach is to begin training with a strong focus on exploration, and slowly lower it as the agent becomes more able to identify value. Ideally this early training is performed in a simulated environment to increase training speed and eliminate any cost of poor action decisions.

In applications where the optimal policy may not be constant, this possibility of ongoing learning may allow an agent to adapt to changing environmental conditions. This does come at some expense, though, as it will necessarily require occasionally taking sub-optimal actions to ensure they are still sub-optimal.

3.5 On-Policy and Off-Policy Learning

All reinforcement learning methods can be divided into on-policy or off-policy approaches. On-policy approaches use the main policy they are training to generate all data, which means it must handle the exploration/exploitation problem itself. Off-policy approaches are a bit simpler, using a second policy to generate data for the main policy such that it only has to focuses on choosing the ideal actions.

A common off-policy approach is to create the data-generation policy by augmenting the main policy. One of the simplest and most popular augmentations is the single-parameter ϵ-greedy modification, where at each step the agent takes a random action with probability ϵ or the best action as determined by the main policy with probability $(1 - \epsilon)$. This allows the single variable ϵ to control the portion of actions spent on exploration, and it is easy to change as learning progresses.

More complex off-policy systems may examine the possible actions and estimate how likely a lower-reward action would be to lead to new useful data. These can introduce significantly more processing overhead than simpler approaches, though, and so one must consider whether the increased value of the datapoints is enough to outweigh the lower number of datapoints that may be used due to reduction in processing time.

3.6 Model Based and Model Free

If the dynamics of an application's environment may be predefined or accurately learned independently of the main RL controller then it is possible to create a model based RL system, in which the environment modelling layer makes available to the RL agent state transition probabilities dependent on actions. This is often more complicated than attempting to learn a policy directly, and is very application-specific. However, when possible it makes transfer learning much easier, which may be desirable if multiple environments with different dynamics require similar tasks to be performed in them.

Any system that does not divide the problem in this manner is called "model free".

3.7 Prior Work

3.7.1 General Reinforcement Learning

Deep Q-Network A significant contribution to reinforcement learning techniques was made by Mnih et al. [33], in which Q-value prediction was performed using a deep neural network. The new method of value function modelling greatly enhanced RL agents performance, allowing them to match or surpass that of humans at a majority of the Atari games used for testing and comparison. The system was also easily adaptable, as the convolutional input layers mean the system can be directly trained on visual data with no task specific feature extraction.

This deep Q-network system serves as the basis for the agents used in this thesis, and further explanation of its function (and modifications made) is given in Section 4.1.2.

Double DQN In some application DQN networks overestimate action Q-values, which can reduce the effectiveness of the learning algorithm and result in worse performance once trained. Hasselt et al. [34] attempt to reduce this with a modification to the previous approach called double DQN. This combats overestimation by dividing action selection into two parts, using the same network structure with two different sets of weights, θ_1 and θ_2. Optimal action selection is performed using θ_1, but Q-value prediction for that action is performed using θ_2, and the results are used to update θ_1. θ_2 is periodically updated by copying θ_1.

Separation of Concerns An interesting approach to handling complex RL tasks is presented by Seijen et al. [35] in their paper on Separation of Concerns. It details how a problem may be divided and agents coordinated such that individual agents solve specific sub-tasks within a problem, communicating with each other in the process, with an action mapping function to convert the resulting set of per-agent actions into a single environment-level action.

3.7.2 Deep Reinforcement Learning in Medical Image Registration

A number of groups have recently examined RL as a solution for medical image registration, though the majority have focused on rigid transforms. In broad terms these approaches are similar to the one examined in this thesis, in that they train an RL agent to perform iterative updates to some transform thereby moving it towards registration.

Liao et al. [36] examined rigid registration of CT and cone-beam CT images, providing complete 3D images as agent input and using a greedy supervised approach during training to address memory requirements due to dimensionality. Registration is performed in an attention-driven hierarchical (coarse-to-fine) manner.

Ma et al. [37] used a network structure based on Dueling Network architecture [38] to rigidly register MR to CT images. Dueling Network architectures split their network into two paths, one which estimates state value and one which estimates additional value attributable to actions, which are then combined to select an action.

Krebs et al. [39] examined non-rigid registration of prostate MR images. The transform used is a statistical deformation model defined from Free Form Deformations, using principal component analysis to reduce the transform's parameter dimensionality. Fuzzy action control is used to stochastically reject actions that would result in large parameter changes.

Miao et al. [40] consider a multi-agent approach to rigid registration of X-ray and CT images. Agents observe different regions of the image and calculate individual action proposals. A correlation is noted between agents' estimated rewards, agent confidence, and utility of the information in the agents' field of view. Therefore actions from high-confidence agents are aggregated to determine the overall action, serving as an auto-attention system to at each step consider only regions of interest.

For further background information, including non-RL methods, a recent and in depth survey of deep learning methods in the field of medical image registration is provided by Haskins et al. [5] covering a wide variety of methods and circumstances. The RL related papers have been covered here.

Chapter 4

Methodology

This thesis proposes a deep reinforcement-learning based non-rigid multimodal medical image registration framework. Deep-RL systems have been found to perform well in learned-feature localization tasks while only processing a fraction of the entire image[7], reducing computational cost relative to systems that must process the entire image. In addition, deep learning in general is very good at learning feature representations, even across modalities. We therefore suspect that a deep-RL system should be capable of performing both tasks together: identifying the per-modality representations of features, and then manipulating a transform to seek out corresponding feature locations in one image based on features present in another. Aligning all matching features would then result in registration of the images.

We also propose that a useful simplification of the non-rigid registration process naturally follows from the feature-location-seeking conceptualization of the registration process. Using a transform where local deformation is defined by the location and movement of parametrized control points allows an agent to perform local registration through simple movement of the corresponding point, in which it considers the region surrounding the initial position of the control point and then seeking out the corresponding region in the paired image. Performing such local registration for each control point would then result in registration of the whole image. Additionally, since each point exists in a similar environment and requires the same sort of task to be performed, a single trained agent should be capable of performing the registration of each point. Reducing a complex system into multiple parallel instances of a single simpler system should allow this method to make

great use of modern advances in GPU hardware, which favour parallel execution.

To test these hypotheses, we have created a test system that performs non-rigid registration in a coarse-to-fine manner using multiple agents. Each agent is assigned to a specific region of the image and is exposed to the sub-section of the moving and fixed images local to that region. This window combined with the action history of other nearby agents serves as the agent input, and from that the agent selects in which direction (if any) to move the local region of the moving image. The aggregate of all agent actions is used to update the transform, and the process is repeated until all agents choose not to move. As each agent instance is performing the same task regardless of position, only one DQN must be trained for each resolution of interest.

When designing an implementation, rather considering broad theory, it became necessary to carefully consider the computational burden imposed by each decision. This is a concern with many medical image tasks due to the large amount of data each such image may contain. In light of this, and considering the desirability of fast or even real-time registration, most design choices were made with the goal of simplifying resource heavy components. For these decisions post-training execution requirements are considered more important than training requirements, as execution speed and the required resources are a limiting factor for more applications.

The novelty of this approach is primarily in the previously unexamined combination of existant components. While there has been vary degrees of investigation into the behaviour of these individual components, it is insufficient to confidently predict the behaviour of a system combining them in this manner, and as such the proposed system requires specific examination. An overview of the structure and interaction of these components is provided in Section 4.1. Section 4.2 then describes the details of the agent training, including data augmentation. Section 4.3 covers how the system is used post-training to register previously unseen images. Specific implementation details are covered in Appendix A, as well as brief descriptions of some noteworthy aspects of development that did not make it into the final system.

4.1 Architecture

After being trained for a given pair of modalities, the purpose of the system is to process image pairs of those modalities and return a transformation that registers one image to the other. Input images are expected to have been modified by both rigid and non-rigid motions relative to each other, but to begin with at least some coarse pre-registration (i.e., organ of interest overlapping, with similar orientation). Registration is then handled in a coarse-to-fine manner, beginning with 1/8 scale and doubling resolution as each scale completes registration. Later scales use the preceding registration results as a starting point and further refine them, and the system's final output transformation is the aggregate of these intermediate transforms.

At each scale a single b-spline transform is used, iteratively updated by RL agents using a learned similarity metric, regularization policy, and optimization strategy (together subsumed into the more general "agent policy"). Each control point of the b-spline has an attached agent instance that takes as input the surrounding pixels and nearby agents' action histories, and returns an update to the control point's parameters. All agents' updates are applied simultaneously to form discrete steps, and registration continues until a step in which no agent chooses to update any transform parameter. During training an upper limit is applied to the number of steps spent processing any given image pair, to encourage sample diversity and increase the number of image pairs seen, as agents with limited or no training are prone to continuing such registration indefinitely. If the number of steps spent processing an image pair surpasses this limit before registration ends naturally, then the current pair is discarded and registration of the next new pair begins as normal.

4.1.1 B-Spline Transform

Image deformation is handled with a second-order b-spline transform (see Section 2.1.3). B-splines were chosen because pixel offsets are determined by a small number of local parameters, and it is easy to identify all pixels associated with each parameter or vice-versa. This allows the parameters to be subdivided by region such that each region is managed independently by a different agent. Using a second-order b-spline results in each pixel being affected by multiple control points, though the nearest control points have far greater influence. If each agent were exposed to all pixels it influenced, this would lead to identical data being analysed many times by different agents due to the inevitable high degree of overlap. Therefore, a window around each point is defined such that there is little or no overlap between windows, limiting the input vector to only those pixels that are closest to and therefore most strongly influenced by that agent's associated point. Using first-order b-spline would avoid this concern but would cause sharp angles at borders between regions of influence, thus less accurately modelling tissue deformation.

For the remainder of this chapter, $T(\theta, \phi)$ will indicate a b-spline transformation with control point parameters θ and predefined parameters ϕ (i.e., mesh resolution, spline order, default grid spacing). $I \circ T(\theta, \phi)$ will denote the transformation being applied to image I.

4.1.2 Deep-Q Network

Deep-Q Networks [33] are a popular type of reinforcement learning algorithm, in which a neural-net is trained to directly estimate the expected reward (Q-value) of each possible action for a given state. With image data pixel values of the full image, or some contiguous patch of it, are used as the state vector s. This is fed through some number of convolutional layers, and then through some number of fully connected layers, producing predicted action rewards $Q(s, a)$ at the final layer. The agent then selects and executes the action a with the highest Q-value, notes the reward signal r and repeats the process until some stop condition is met. Here, actions consist of "moving" an agent's associated control point by increasing or decreasing one associated parameter by some step size S. The only exception to this is the "stop" action, which indicates that point is currently considered to be registered and should not be moved.

In this thesis, the Q network is modified slightly as shown in Figure 4.1. The region around the target control point is extracted fed through the convolutional pathway. As

mentioned in Section 4.1.1, it is assumed that the local deformation of a given region of the moving image is most usefully informed by the regions from both images physically near the motion, as they will contain the structures able to be aligned or misaligned due to that motion. More distant information is indirectly available via message passing in the form of mutually visible action history between neighbouring agents. In cases where local structures alone do not provide enough useful information to certain agents, this allows the decisions of more confident agents to propagate. A combined action history of the control point and its neighbours, along with the output of the convolutional image processing path, is fed through a fully connected path which then produces as its outputs the predicted Q-value for each possible action. Action histories are converted to a one-hot encoding to allow easier processing by the fully connected layer. One-hot encoding consists of creating for each history entry a number of ordered boolean variables equal to the number of possible actions, and then setting only the variable corresponding to the action number to one and all others to zero. This prevents errors that would likely be introduced by other formats such as passing an integer indicating action number, as due to neuron operation that input channel would necessarily be assumed to be a continuous scalar quantity rather than an indicator of distinct categories.

When training the agent, it is necessary to define some reward signal to be used as a target while training the neural net. As the agent attempts registration while in the training environment it will at each step observe a state, select an action, and receive a reward. This state/action/reward data may then be used to train the agents by improving the accuracy of their reward prediction network. However, concurrent or consecutive actions are highly temporally correlated, resulting in a non-stationary target value function . Therefore, to render the target function stationary, network training data is randomly sampled from an "experience replay" buffer (ERB) [41] which stores a large number of data points from previous runs. The network is updated via gradient descent using these randomly sampled data points. More detail on the exact implementation of this process is provided in Section 4.2.

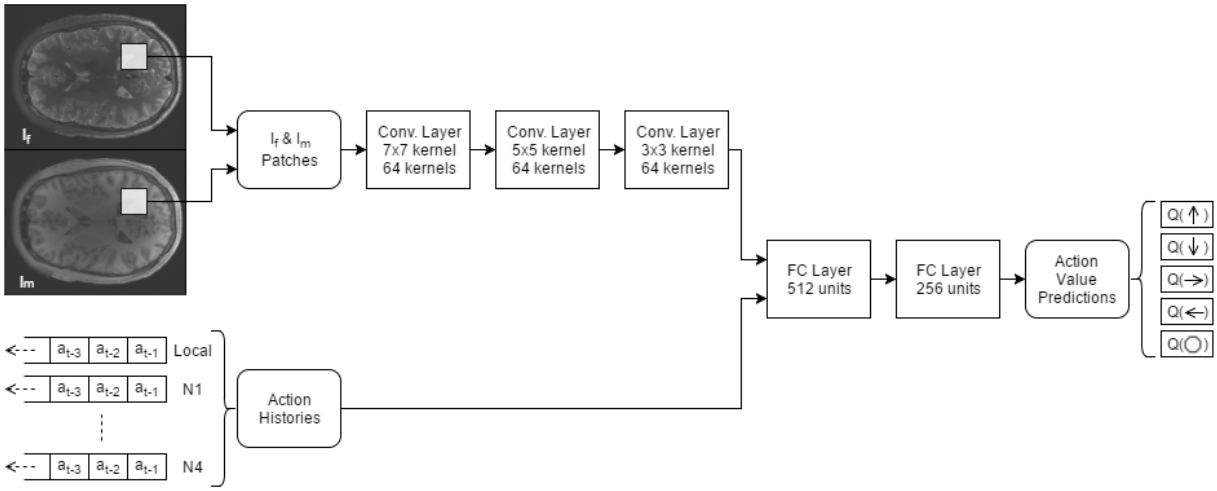

Figure 4.1 DQN structure and layer details. Input consists of agent state: image subregions surrounding the agent's associated b-spline control point, extracted from the moving and fixed images, combined with the action history of that agent and the four orthogonally neighbouring agents. The image data is passed through three convolutional layers, as shown in the upper path. The output of this convolution and the action histories are joined together and passed through two fully connected layers. The final output is an estimation of the state-action value function $Q(s, a)$ for each possible action. Brain images from the Human Connectome Project [8].

Parallelization

As mentioned previously, the same task required at each control point. Training time is reduced by only training one agent for each of the coarse-to-fine scales, and memory use is reduced as only a single shared set of network weights must be tracked regardless of the number of agent instances. However, action history must be tracked for each control point, and so the "agent" associated with each point is in reality just a unique history buffer and a reference to a shared Q-value estimation function. This enables easy parallelization of agent prediction, as many modern ML libraries have good support for batch processing, allowing a single network to process multiple inputs simultaneously. Calculating all agent actions for a step is therefore negligibly slower than calculating a single agent action.

During training, the multiple agents acting in parallel each generate their own unique state/action/reward sequences. As each agent is dealing with a different area of the image this reduces the problem of non-stationarity, but due to the shared base image and correlation between agent actions it does not resolve the problem completely. We therefore must retain the ERB concept, but replace the single buffer with a collection of smaller per-agent buffers, maintaining state/action/reward sequence chronology within each sub-buffer. Additionally, it is then possible to ensure that all sub-buffers receive sufficient attention during sampling, reducing the likelihood that anatomical features local to a specific region might be overlooked. Buffers for points that move outside of some defined region of interest (e.g., for brain registration those points outside the skull in free space) may also be ignored, reducing the influence of irrelevant data and most likely increasing learning rate or accuracy. For each training step the specific information each agent stores in its buffer is the initial image patch state s_p, the most recently selected action a, the resulting reward r, and a binary value t noting whether that step was the end of the registration sequence.

For Q-value prediction, the action histories of a node and its neighbours are converted to a 1-hot encoding s_{ah}. Histories are zero padded for any missing neighbours or in cases where history length reaches before the first step. The full state s then consists of the combined action history and image patch, $s = s_p \cup s_{ah}$. The state s is then used as input for the Q-value function.

4.1.3 Coarse-to-Fine Registration

Limiting the agent's state information to a window surrounding the associated point increases speed and prioritizes relevant information, but it limits the agent's capture radius. If the agent cannot see any corresponding features between the images at the same time, it is unlikely to be able to find a reasonable solution. To address this problem images are registered in an iterative coarse-to-fine manner. Registration at lower resolutions is much faster, but produces a less accurate result. However, by first registering a low-resolution with few control points, it is possible to find a transformation that, when applied to the higher-resolution version of the moving image, brings it within the capture radius of an agent trained to register at that finer scale.

4.2 Training

4.2.1 Data Augmentation

Due to the limited training data available for most medical applications, data augmentation is included in the training process to reduce the risk of model overfitting. Whenever a new training pair is to be loaded, a randomized b-spline transform is generated. Each parameter of the augmentation b-spline is independently sampled from a normal distribution with zero mean and a user-determined standard deviation, where that standard deviation is selected according to the images of interest and the plausible range of deformations they may contain. The resulting transform is applied to both members of the original image pair as read from disk to produce a modified unique aligned image pair, which is then used as input for agent training. As a result every image pair encountered by the agent should be at least slightly different, even when considering multiple pairs generated from the same base data.

Rigid transformations were not used in data augmentation as they would not change the appearance of local features, and the uniform and repeating nature of the agents means that small rigid translations should not have any noticeable effect. Large rigid translations are assumed to be addressed by simple pre-processing removing useless large empty regions of the images.

4.2.2 Agent Training

A separate agent is trained for each resolution scale to be used: full resolution, 1/2 full, 1/4 full, and 1/8 full. Doubling resolution between scales simplifies calculations and reduces noise from interpolation. Before training, for each resolution one must select spline-order, spline mesh size, step size, and state history length, action history length, and window size. Algorithms 6 and 7 are then used to train each agent. During training actions are selected according to an ϵ-greedy policy, and the value of ϵ changes throughout execution. It begins at $\epsilon = 1$, selecting all values randomly as the net is initially untrained. As training progresses ϵ decreases linearly with each step, until reaching and remaining constant at some value ϵ_{final} after a specified number of steps. This encourages high exploration of the state-space early on in training while agent performance is poor, but as performance improves and ϵ falls random actions are reduced, meaning late training focuses more on the areas of state-space very near regions encountered during successful operation. This allows greater accuracy, as network capacity is not wasted learning how to handle states that would only be encountered during already poor agent operation.

Input: *Image pair database D_{pairs}, training hyperparameters*
Output: *Trained DQN model parameters*
while *step* $<$ *step*$_{max}$ **do**
    `// Load images`
    $\{I_{f,file}, I_{m,file}\} \sim D_{pairs}$;
    `// Apply data augmentation to raw images`
    $\theta_{aug} \sim \mathcal{N}(0, \sigma^2_{aug})$;
    $I_{f,aug} = I_{f,file} \circ T(\theta_{aug}, \phi_{aug})$;
    $I_{m,aug} = I_{m,file} \circ T(\theta_{aug}, \phi_{aug})$;
    `// Sample ground truth warp`
    $\theta_{GT,r} \sim U(-x_{train}, x_{train})$;
    $\theta_{GT,nr} \sim \mathcal{N}(0, \sigma^2_{train})$;
    $\theta_{GT} = \theta_{GT,nr} + \theta_{GT,r}$;
    `// Apply ground truth warp to fixed image`
    $I_f = I_{f,aug} \circ T(\theta_{GT}, \phi)$;
    $I_m = I_{m,aug}$;
    `// Use current pair for training`
    Run per-pair inner training loop shown in Algorithm 7
end

Algorithm 6: Agent training process. Variables described in Table 4.1. Visualization shown in Figure 4.2

$t = 0$;
while $t < t_{max}$ *and not* $(a_t^p = \text{“}stop\text{”}\ \forall\ p)$ **do**
  $t = t + 1$;
  foreach *control point* p **do**
    Sample $s_t^p = \{I_f(W_p), I_m(W_p) \circ T(\theta_{bsp}, \phi)\}$;
    if $U(0,1) < \epsilon_{step}$ **then**
      Choose random action $a_t^p \in A$
    else
      Select best $a_t = \underset{a}{\text{argmax}} Q(s_t^p, a)$
    end
    Update θ_{bsp} *according to* a_t^p;
    if $a_t^p = \text{“}stop\text{”}$ **then**
      if $||\theta^P_{GT,t} - \theta^P_{bsp,t}|| \leq stepSize$ **then**
        $r_t^p = 1$;
      else if $||\theta^P_{GT,t} - \theta^P_{bsp,t}|| \geq rewardRad$ **then**
        $r_t^p = 0$;
      else
        $r_t^p = \frac{(||\theta^P_{GT,t} - \theta^P_{bsp,t}|| - rewardRad)}{(stepSize - rewardRad)}$;
      end
    else
      $r_t^p = (||\theta^p_{GT,(t-1)} - \theta^p_{bsp,(t-1)}|| - ||\theta^p_{GT,t} - \theta^p_{bsp,t}||)/stepSize$;
    end
  end
  Compare $\Delta\theta_{bsp}$ *and* θ_{GT} *to find rewards* r_t;
  Save all s_t, a_t, r_t *to ERB;*
  if *steps % updateFreq = 0* **then**
    Sample ERB and use for gradient descent update on DQN;
  end
end
Save final $s_{t+1}^p = \{I_f(W_p), I_m(W_p) \circ T(\theta_{bsp}, \phi)\}$ *to ERB;*

Algorithm 7: Inner loop of Algorithm 6, generating training data from prepared image pairs and training DQN. Variables described in Table 4.1. Visualization shown in Figure 4.2

Symbol	Description
D_{pairs}	Dataset containing all training images
$step$	Current training step
$step_{max}$	Total number of steps to take during training process
$I_{f,file}$ and $I_{m,file}$	Fixed and moving images, as loaded from files
$I_{f,aug}$ and $I_{m,aug}$	Fixed and moving images after data augmentation
I_f	"Target" fixed image, with a known ground-truth transform applied
θ_{aug}	Parameters for data-augmentation b-spline
$\theta_{GT,r}$ $\theta_{GT,nr}$ θ_{GT}	Rigid component, non-rigid component, and combined parameter vector for ground truth b-spline used to create I_f
θ_{bsp}	Parameters for "active" b-spline repeatedly applied to I_m
ϕ_{aug}	Hyperparameters for data-augmentation b-spline
ϕ	Hyperparameters for ground-truth and "active" b-splines
σ_{aug} σ_{train} x_{train}	Control of data augmentation and target image deformation
t	Step count for current image pair
t_{max}	Maximum number of steps to spend on a given image pair
s_t	Input state vectors for step t
r_t	Rewards received at step t
a_t	Actions taken at step t
x^p	Subset of vector x associated withcontrol point p, where $x \in \{s, r, a, \theta\}$
W_p	Region definition for "window" surrounding control point p
p	Control point currently under consideration
ϵ_{step}	Likelihood of selecting a random action, changes throughout training as $step$ increases
$Q(s, a)$	State-value estimation function as applied to state s and action a
$U(a, b)$	Uniform distribution ranging from a to b
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution with mean μ and standard deviation σ
ERB	Experience Replay Buffer
$updateFreq$	Number of steps to wait between performing learning updates on DQN and a new training pair is loaded
$stepSize$	Distance a control point moves with each step
$rewardRad$	Distance from ground truth beyond which "stop" actions return a reward of zero

Table 4.1 Agent training variables used in Algorithms 6 and 7 and Figure 4.2

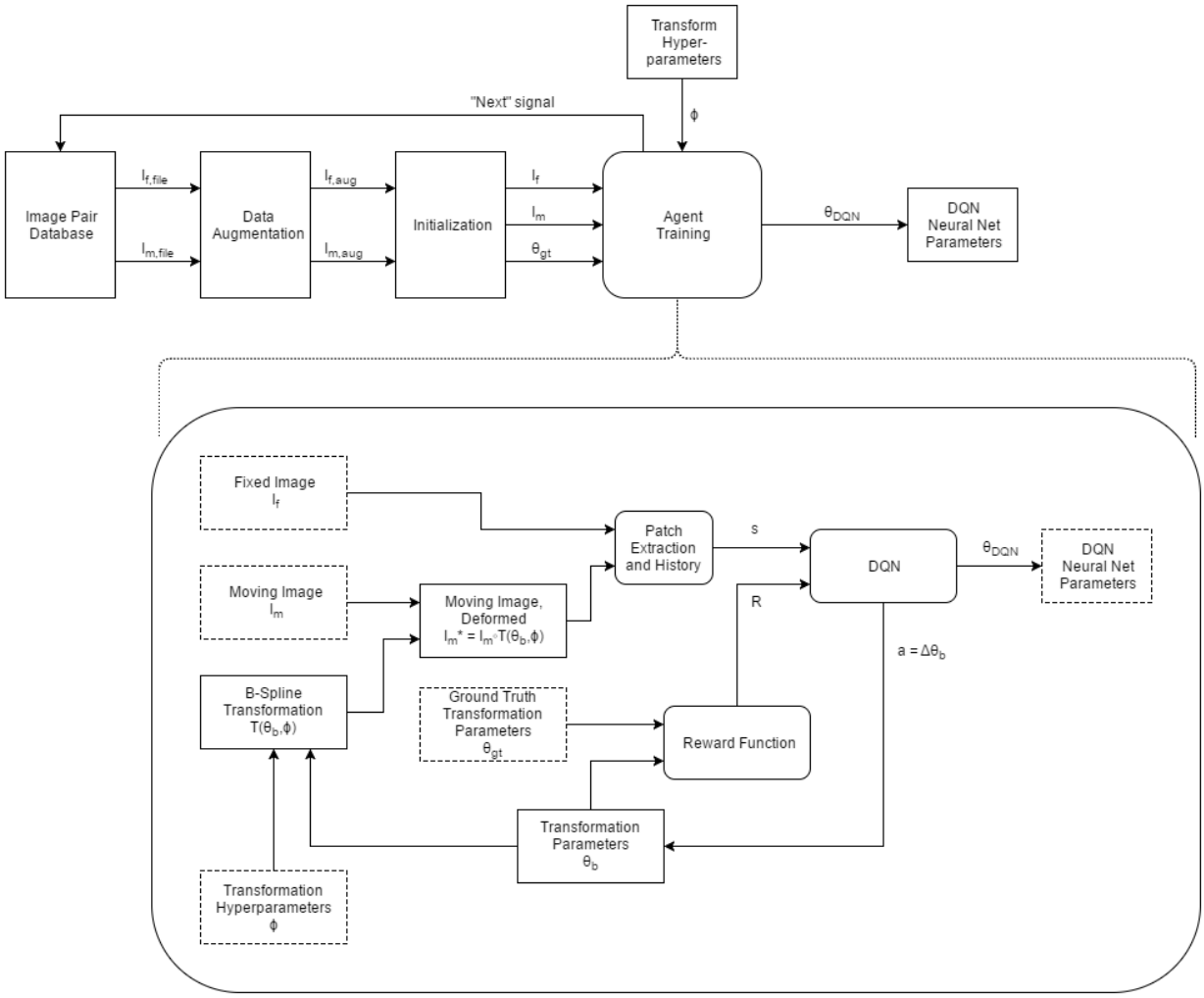

Figure 4.2 Visualization of the agent training process described by Algorithms 6 and 7. Variables described in Table 4.1.

For all movement actions, the reward signal is defined as the change in distance between the current b-spline parameter vector θ^P_{bsp} and the known ground-truth b-spline parameter vector θ^P_{GT}, where θ^P is the subset of parameters in θ associated with control point P:

$$r_t = \frac{||\theta^p_{GT} - \theta^p_{bsp,(t-1)}|| - ||\theta^p_{GT} - \theta^p_{bsp,t}||}{||\theta^p_{bsp,t} - \theta^p_{bsp,(t-1)}||} \tag{4.1}$$

This change in total error $\Delta||(\theta^p_{GT} - \theta^p_{bsp})||$ is divided by step size $\Delta||\theta^p_{bsp}||$ to normalize the reward to $[-1, +1]$.

If $a_t = stop$ then the corresponding reward is determined by the remaining error $||\theta^P_{GT,t} - \theta^P_{bsp,t}||$. Within one step of ground truth, the maximum reward is given. Larger errors result in a lower reward, with the reward decreasing proportional to error until it reaches zero at some user specified maximum error. Additionally, the maximum reward starts at zero and linearly increases until reaching a plateau after some number of training steps. This serves a similar purpose to using an ϵ-greedy policy, encouraging exploration early in training. Note that to prevent early stopping the maximum $a = stop$ reward after reaching the plateau should be no higher than the lowest possible reward that could be given following perfect action selection.

There are two end conditions for an image pair, after which the next pair is loaded. The first is that "stop" action is selected for all control points simultaneously. The second is that the number of actions taken with the current image pair has exceeded the per-pair limit. This second condition is most useful early in training, as untrained agents are unlikely to work in tandem to trigger an intentional stop, and are more likely to oscillate or wander far from any informative region. Limiting the number of actions per pair ensures training can continue without imposing any pre-defined constraints on agent behaviour.

When reading the state, W is the pixel/voxel window surrounding P. When $I(W)$ is sampled, if the window would reach outside the image it is zero padded. During DQN training, after every 5000 actions the agent is given a number of pairs from the validation set with which to test its performance. This has the same framework as the agent training, although without touching the ERB or updating the DQN. The final error for each test run (i.e., the distance from ground truth when the agent chooses to stop) is recorded. The distribution of this validation error during training, and its change as training progresses, is used to evaluate performance during training.

4.3 Registration and Evaluation

Registration using the trained agents is very similar to the training example, with the exception that there is no reward signal, no ERB, no network updates, and epsilon is zero. Additionally, the coarse-to-fine registration path is used. This process is outlined in Algorithm 8 with a visualization shown in Figure 4.3.

Input: *Image pair* I_f, I_m*, trained DQN model parameters for all scales*
Output: *Transform parameters* θ_{bsp}
foreach *scale* $k \in 1..K$ **do**
  Resample $I_{f,k}, I_{m,k}$ *from* I_f *and* I_m*;*
  Load agent Q-network for current scale;
  repeat
    foreach *control point* $p \in P$ **do**
      Resample $s_t^p = \{I_{f,k}(W), I_{m,k}(W) \circ T(\theta_{bsp,k}, \phi_k)\}$*;*
      Select best $a_t^p = \underset{a}{\text{argmax}} Q_k(s_t^p, a)$*;*
    end
    Update θ_{bsp} *according to all* a_t^p*;*
    Save new θ_{bsp} *in buffer;*
    Examine θ_{bsp} *buffer for revisited values;*
  until $a_t^p =$ "*stop*" $\forall\, p \in P$*, or oscillation detected;*
  Update $I_m = I_m \circ T(\theta_{bsp,k}, \phi_k)$*, scaling* $\theta_{bsp,k}, \phi_k$ *to match full scale;*
end

Algorithm 8: Registration process using trained agents.

A buffer of user-determined length records recently seen values of θ_{bsp}, to be used as a safeguard against infinite oscillation. If the same θ_{bsp} appears in the buffer more than once, the algorithm has begun repeating and is assumed to be oscillating around some local optimum that it cannot quite reach. This secondary stop condition is not expected to be triggered with any regularity, as by this point the agent has been trained when to stop correctly, and so it is simply included to prevent a potential mode of failure that would lock the algorithm in an infinite loop should it occur.

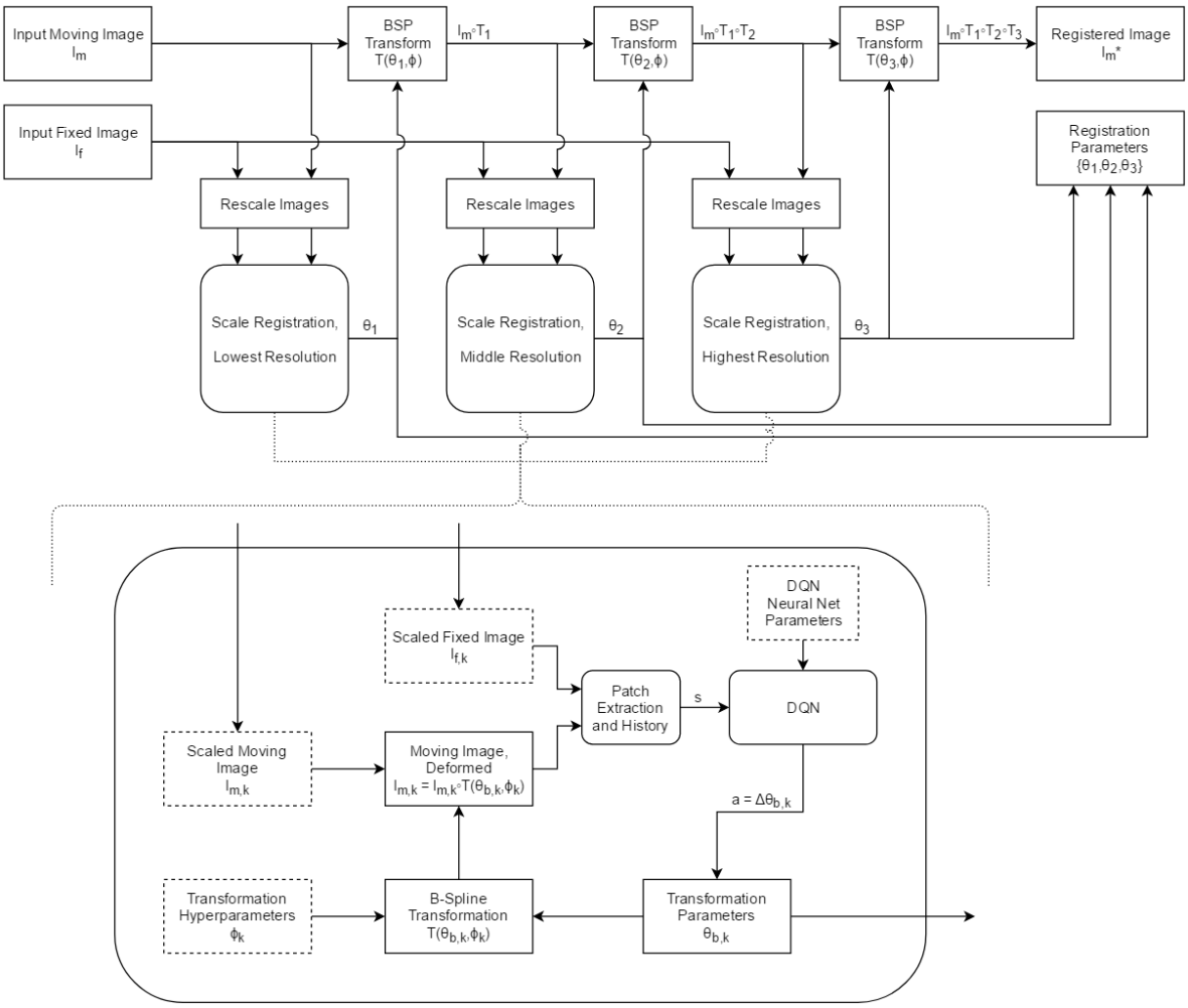

Figure 4.3 Visualization of the coarse-to-fine registration process described by Algorithm 8. Three scales are shown here for explanatory purposes, but the actual number is determined by the user prior to agent training.

4.4 Summary

This chapter has described the nature and structure of the system under investigation in this thesis. Details are provided regarding the application of b-spline transforms, the architecture of the DQN, and how agents using that DQN are used to update the transform. Training is described in detail, including what data is used, the meaning of the predictions made during action selection, and how the reward signal determines action quality. Post-training system execution is discussed briefly, as it is nearly identical to the training process after removing the subsystems related to sample generation and network updating.

The next chapter will examine the performance of this system, showing results for brain MRI registration.

Chapter 5

Experiments and Analysis

5.1 Overview

Before evaluating agent performance, it was first necessary to perform a number of experiments examining the effect of individual hyperparameters on agent performance, with a focus on selecting optimal values for a specific application. These tests focused on agents operating at a coarse scale, as the reduced hardware requirements allowed more tests to be performed within the available timeframe. The results are still informative, however, as the purpose of these initial tests was not to evaluate the performance of the full and final system but rather to examine relative agent performance and to choose a training environment expected to produce the best results. The test format and final parameter selection are described in Section 5.3, and the full results are discussed in Appendix B.

Having completed this first set of tests and selecting suitable hyperparameters, the tests discussed here were used for more in-depth investigation of agent performance. The primary concern was to investigate whether, at each scale before 1:1 in the coarse-to-fine process, the associated agent is capable of consistently registering images at that scale with error low enough that it is within the capture radius of the next agent. This is necessary to demonstrate reliable function of the coarse-to-fine process, as the agent of each layer after the first must rely on previous layers to supply transformed images with starting error within that agent's capture radius. With each scale change resolution doubles and capture radius is approximately halved. As such, each agent for a scale coarser than 1:1 will be deemed functional if it is able to produce a registration with error below half of that scale's starting error. The accuracy of the 1:1 scale agent is also examined, as that is what

will most strongly influence the final error of the full system. Monomodal performance is explored in Section 5.4, and then compared to multimodal performance in Section 5.5. Agent performance on the training and validation sets is examined in Section 5.6, addressing the risk of overfitting. Finally, Section 5.7 discusses some of the problems encountered and their potential solutions.

Aside from these main tests, some additional results stemming from specific events during development are discussed in Section 5.2.

5.1.1 Test Structure

For all tests discussed here, agent training is performed as described in Section 4.2.2, with periodic accuracy evaluations performed throughout the training process. Each evaluation consists of multiple test registrations at that specific scale using the validation dataset. Initial deformations follow the same distribution as in the training algorithm, for an average initial per-control-point deformation of 6 steps. The exact number of tests performed during each evaluation is not constant; rather, 2500 steps are taken with a new test beginning each time the previous test concludes, generally resulting in several hundred test registrations for each evaluation phase. This prevents poor agent performance from dramatically slowing the training process with overly long tests.

5.1.2 Target Data

Training and testing were performed using MRI data from the Human Connectome Project [8]. While such images are naturally 3-dimensional, processing full 3D MRI volumes was impractical in the available testing environment. GPU memory limitations combined with the larger network size of 3D convolutional layers resulted in insufficient space for an appropriately sized experience replay buffer, and applying transforms was slow enough that it was infeasible to perform the appropriate hyperparameter exploration in the time available. Therefore a 2D version of the system was implemented to allow testing of the core concept. If performance for the 2D case is promising, that may inspire future testing of the 3D case on more powerful hardware.

With this change, it became necessary to create 2D images from the available 3D volumes. This was accomplished by extracting the central transverse planes from each of the 1113 T1/T2 pairs, producing two aligned 256x320 8-bit/pixel greyscale images per volume

pair. The central transverse plane was selected as it had a consistently high ratio of brain to empty space content, and the brain regions featured a variety of visually distinct regions and features. For coarse-to-fine training copies of these images were generated at 1:2, 1:4, and 1:8 scale, as shown in Figure 5.1. For each scale 890 images were used for training and 223 were used for validation.

5.2 Extracts from the Development Process

During development there were some challenges and events of note that will be described here briefly. While not conclusive results of particular import on their own, they influenced design and implementation decisions and are included here for the sake of completeness.

Image Processing ITK libraries were used for spline-based image deformation early on, but both the transformation calculation and image resampling were found to be too slow. Specifically, they were limited to running on the CPU, which both resulted in slower calculation and introduced significant overhead as image data was copied from system RAM to GPURAM with every ERB update. Therefore it was necessary to reimplement nearly all image handling code.

Even when fully performed on the GPU, calculating the dense optic flow transformation from the B-spline parameters introduced a severe speed limitation. The solution developed here was to store several partially computed large matrices related to the calculations, set up according to all predetermined hyperparameters, and then to use the transform parameters to index and combine those results. Specifically, u and v in Equation 2.12 are unique only for $0 \leq x < n_x$ and $0 \leq y < n_y$, and are periodic with regards to x and y in general. Therefore the $B_l(u)B_m(v)$ products are pre-calculated for all integer values of x, y within that range, for each l, m pair. Every time the dense transformation field is required these precomputed values are then multiplied by the corresponding local parameters for each area, which is highly parallelizable and relatively quick to perform on the GPU.

Double DQN As mentioned in Section 3.7.1, a double DQN [34] modification may be used to address the overestimate of action Q-values observed with DQN networks applied to certain tasks. There does not yet appear to be a clear way to predict for which tasks this overestimation will occur, and when it occurs that can reduce the performance of

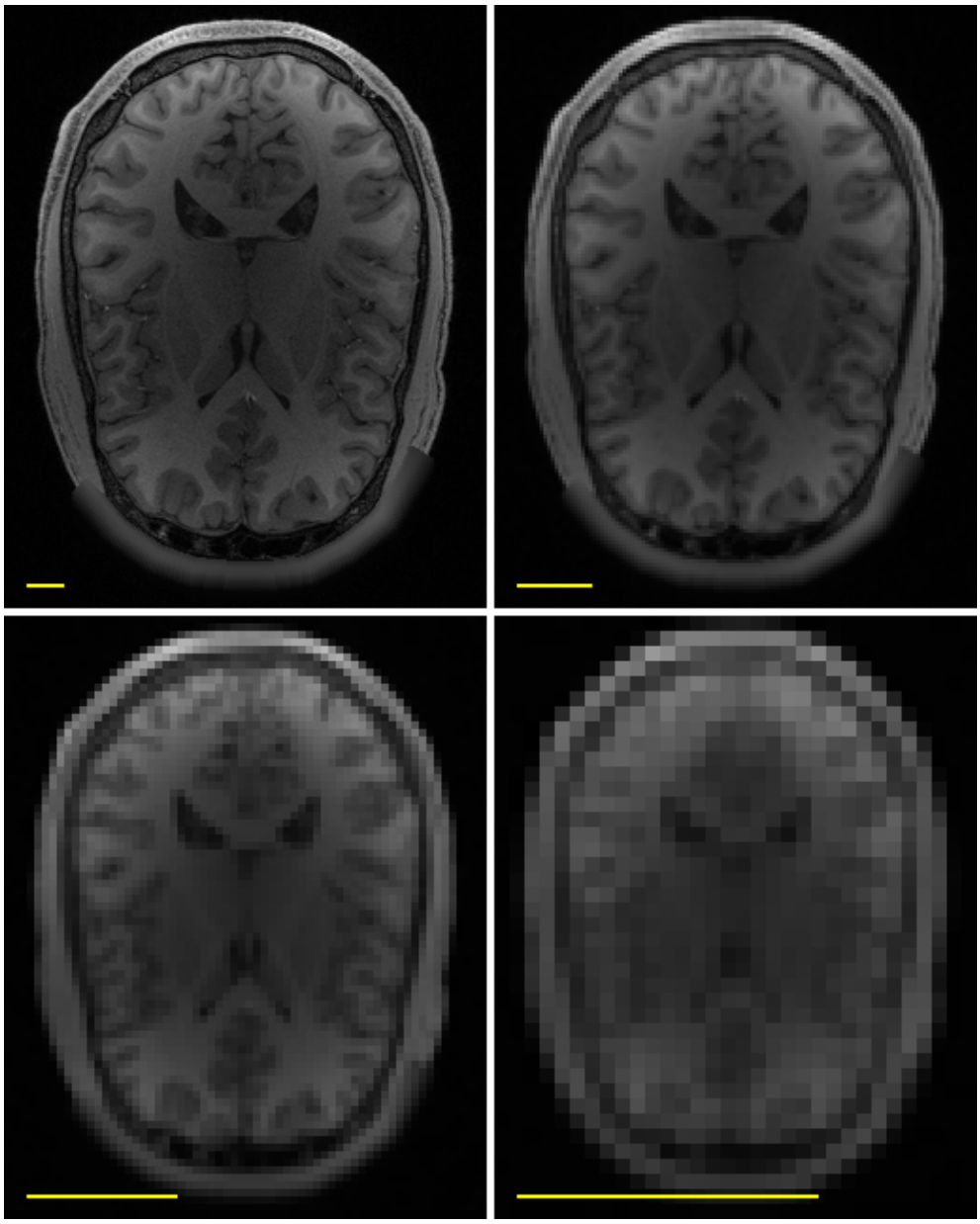

Figure 5.1 Example of scaled T1 images. Upper left is 1:1, upper right is 1:2, lower left is 1:4, and lower right is 1:8. The yellow line in each image represents the distance spanned by 20 steps at that scale. As with all experiments in this chapter, one step corresponds to a change in control point position of one pixel at that agent's associated scale. Base image from the Human Connectome Project [8].

trained agents. A double DQN modification was therefore implemented here to examine the effect on performance, and to ensure this potential overestimation would not affect this application.

Early tests (following the process in Section 5.3) showed no change in accuracy, however, and so it was concluded that this was unlikely to be a rewarding avenue of inquiry. The modification was therefore removed to eliminate the associated minor speed and memory costs. While the double DQN approach did not affect accuracy in this case, it is unclear whether that is because this approach to registration does not trigger Q-value overestimation or whether double DQN modification does not noticeably reduce such overestimation.

Parallelization Early work considered a non-parallel training regime. For each run, one control point would be randomly selected to be the "active" point, and no other points would move. This reduced overhead and allowed for larger state vectors and longer runs to be stored in the same amount of ERB memory. However, when actual registration was attempted and all points were free to move then the accuracy would degrade to the point of uselessness.

Error from the singular point prediction was expected, especially in low-information image regions, partially due to lack of communication between agents. The initially intended solution to this was to augmenting the network, adding additional layers to regularize the control point actions. For each step the original network would predict all action Q-values, which would then be collected and treated as a $n \times i \times j$ image, for $i \times j$ nodes with n possible actions each. A second network would then be trained to predict the correct $i \times j$ action set from this, allowing less certain control points to be influenced by their more certain neighbours and reducing single-point-errors.

Preliminary testing was performed, but the mean reduction in registration error was most often indistinguishable from zero. It seemed that either accuracy could not be improved by the second network, or that the training time required to observe improvement was far beyond what was feasible or acceptable. The system was therefore redesigned as explained in Chapter 4.

The subsequent approach to parallelization more closely resembled the SoC methods used by Seijen et al. [35], but it was found that significant simplification was necessary to accommodate the volume of data in use. With that simplification many other aspects of

the original became superfluous or detrimental and were removed, eventually resulting in the system described in Section 4.1.2.

5.3 Hyperparameter Testing

The hyperparameter values shown in Table 5.1 are used for all tests, unless otherwise noted. These values were determined by varying one hyperparameter at a time and performing a simple hillclimbing optimization. For each step training was repeated multiple times using different values for the hyperparameter of interest, in each case selecting the value that resulted in the best performance. The process was repeated until performance was no longer improved. Appendix B contains the results of a number of tests recreating the final cycle of this refinement process, and analyses the influence of each individual hyperparameter.

Parameter	Notation	Default Value
Reward discount	γ	0.3
Learning Rate	α	10^{-5}
Experience replay buffer size	$\|ERB\|$	$4 \cdot 10^4$ ($\{s_t, a_t, r_t\}$ records)
Patch size	$\|P\|$	25x25 (pixels)
Action History Length	$\|H_{act}\|$	18 (actions)
Neighbour Action History Length	$\|H_{nghbr}\|$	14 (actions)
Lowest random action probability	ϵ_{final}	0.15
Number of steps before reaching ϵ_{final}	N_ϵ	$8 \cdot 10^4$ (steps)
Maximum reward for "stop" action	$r_{sm,final}$	0.3
Number of steps before max "stop" reward	$N_{r,stop}$	$8 \cdot 10^4$ (steps)
Radius for max "stop" reward	Rad_{max}	1 (pixel)
Radius for any "stop" reward	Rad_{any}	6 (pixels)
Training NR deformation std. deviation*	σ_{train}	32; 16; 8; 4 (pixels)
Training global rigid deformation range*	x_{train}	16; 8; 4; 2 (pixels)
Control point grid size*	-	4x4; 8x8; 8x8; 16x16 (# points)

Table 5.1 Default experiment hyperparameters. Entries marked with * are scale specific, and show in order the values for 1:8, 1:4, 1:2, and 1:1 scale agents.

5.4 Monomodal Performance

Using the hyperparameters found as described in Section 5.3, an agent was trained for $5 \cdot 10^5$ steps in accordance with Section 5.1.1. Figure 5.2 shows the results of the validation tests performed throughout training for each agent, demonstrating that accuracy improvement continues throughout the entirety of training for every scale. The rate of improvement continually decreases as training progresses, but performance at each scale is still clearly improving throughout the whole process, even at the end of the training sequence. This suggests that the errors observed in this chapter should be taken as upper bounds, and that with sufficient time for further training one should expect better performance.

Increased resolution seems to consistently result in worse performance during early training, lower error once trained (if given sufficient time), and lower variation of error. Comparing the most disparate resolutions: the 1:8 scale agent achieved an average error of 3.8 steps with 95% of results falling between 2.3 and 5.4 steps, while the 1:1 scale agent's average error rate was 2.5 steps with 95% of results falling between 2.9 and 2.0 steps.

While coarser agents must simply reduce error below the subsequent agent's capture radius, the 1:1 scale agent determines final error of coarse-to-fine registration. It is fortuitous that it is observed to have the greatest accuracy and consistency.

It is worth reiterating here that the error for agents is measured in steps, which are not equal between scales. Step size may be altered during initial configuration, but in all training examples examined here one step changes the control point position by one pixel at the relevant scale. Therefore, converting the previously mentioned examples and comparing them in the context of the full-size image, the 1:8 scale agent has a mean error of 30.4 pixels while the 1:1 scale agent has a mean error of 2.5 pixels. The initial regional misalignments of the images shown to agents at the coarsest scale have an average displacement of 15% of the image's longest axis.

Considering the context of the full registration framework, we unfortunately find that the 1:8 and 1:4 scale agents do not improve enough to consistently reach the capture radius of the subsequent agent, preventing full coarse-to-fine registration from working with this degree of training. Given that agent accuracy was observed to have not reached a maximum in the allotted training time, as evidenced by performance for these agents continuing to improve at a steady albeit slow rate, it is plausible that with further training it may reach the necessary accuracy.

The 1:2 scale agent was observed to reliably produce registrations that reach the capture radius of the 1:1 scale agent. This means that, while full coarse-to-fine registration is not achievable with this level of training, partial coarse-to-fine registration may be performed by starting at 1:2 scale. This two agent system would be capable of reducing initial misalignments with an average error of 12 pixels down to an average error of 2.5 pixels.

An example registration sequence of the 1:2 scale agent is shown in Figure 5.4. The fixed, initial moving, and final moving images are shown in greater detail in Figure 5.3. This registration was the final evaluation registration performed during the 1:2 scale training sequence seen in Figure 5.2.

5.5 Monomodal vs Multimodal Performance

Multimodal performance was observed to be significantly worse than that of corresponding monomodal tests. At 1:8 scale, as shown in Figure 5.5, behaviour was similar to the monomodal case though with worse accuracy. Error continually decreased with no period of prolonged consistent error being observed and so, as with the monomodal case, one may predict that with further training this accuracy would continue to improve. However, the much slower rate of improvement relative to the monomodal case suggests it would take a very long time.

In a departure from the results of all other tests, higher resolution multimodal agents offer no useful contribution to registration. At higher resolutions the agents' error was nearly identical to the error distribution used during initial image deformation generation, as over 95% of control points were observed to stop within 1 step of their starting position. This is despite each point taking an average of at least 12 steps. Figure 5.6 highlights this lack of useful improvement, showing that during validation the agents do not move towards or away from their goal. This behaviour emerged after approximately $8 \cdot 10^4$ training steps and changed very little throughout the remainder of training. While there was very minor improvement, the rate of change is low enough that it is infeasible to pursue testing with currently available hardware.

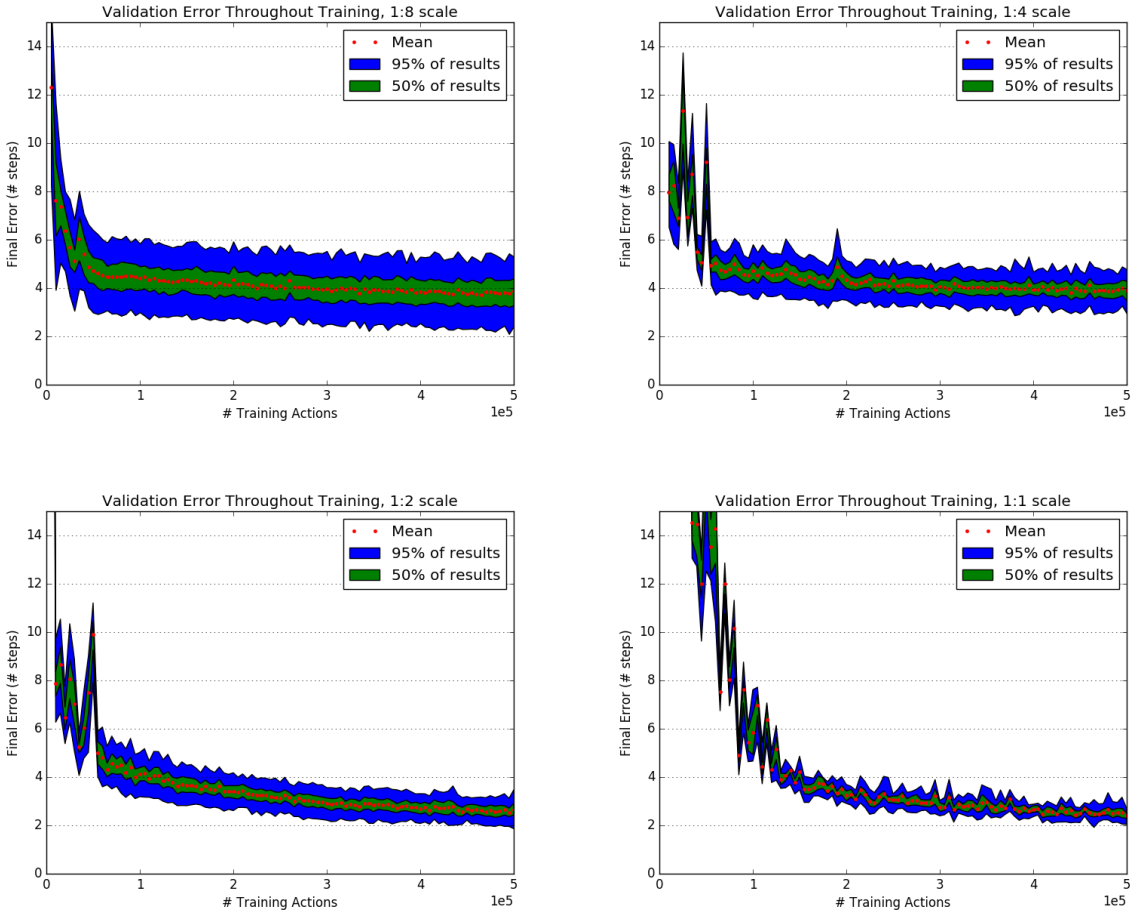

Figure 5.2 Extended training for monomodal T1. Graphs are for 1:8 scale (upper left), 1:4 scale (upper right), 1:2 scale (lower left), and 1:1 scale (lower right). For each sub-graph vertical axis measures per-attempt error, where the error of each attempt is calculated by taking the mean of the per-agent differences between final transform parameters and ground-truth transform parameters. Step size = 1 pixel/step, so given error is in both steps and pixels. Horizontal axis measures the number of actions taken before the corresponding validation test was performed, where each such action is the aggregate of all agent actions at a given timepoint. More training actions also result in more SGD updates to the network weights, due to the structure of training detailed in Chapter 4. The red dots indicate the mean value of all measured errors for validation tests performed after the indicated amount of training. The green region indicates the range of errors observed in the 50% of tests with error levels closest to the mean test error. The blue region is as the green region, but for 95% of the results.

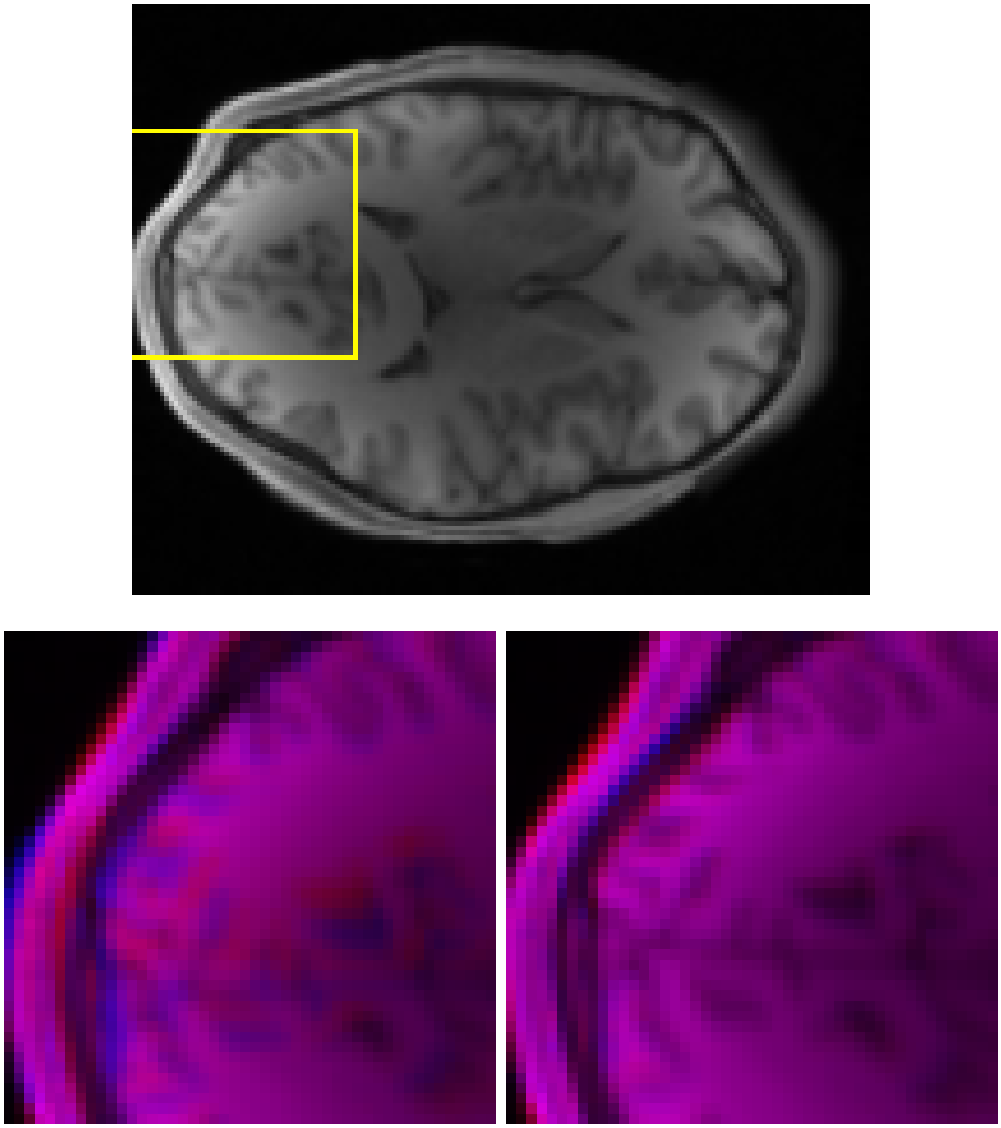

Figure 5.3 Example monomodal registration results for 1:2 scale agent. The upper image is the full fixed image, I_f, with a highlighted segment. The lower images are expanded views of that segment in red, overlaid with the corresponding portion of the moving image I_m in blue. The lower left uses I_m prior to registration, showing many clear misalignments. The lower right uses I_m after registration, and shows only minor misalignment in part of the skull. The full registration sequence is shown in Figure 5.4. Base images are from the Human Connectome Project [8]

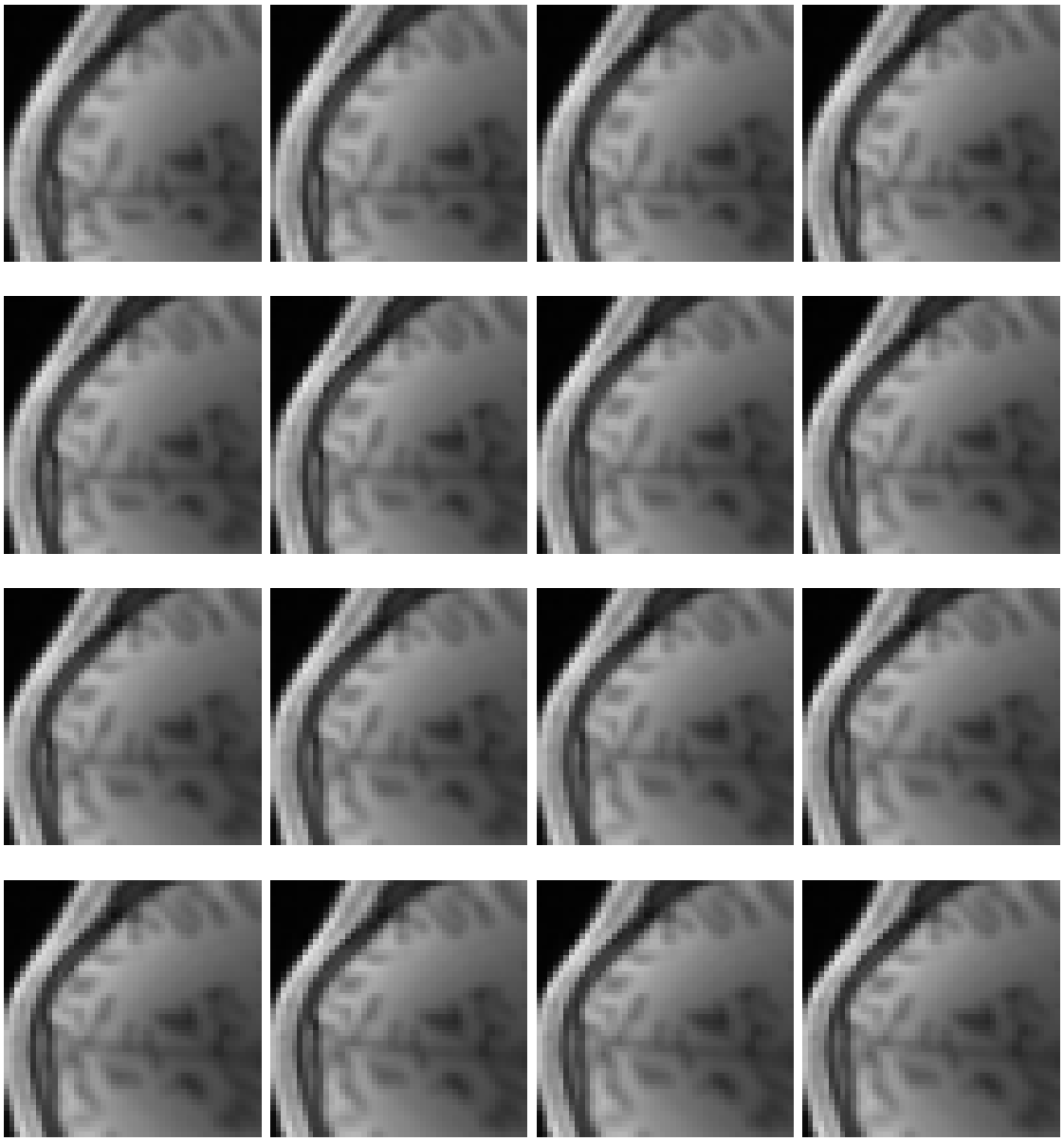

Figure 5.4 Example monomodal registration sequence for 1:2 scale agent. Shows detail view taken from I_m, corresponding to the highlighted section in Figure 5.3. The first image, in the upper left, is from the initial moving image. Images proceed in a sequence, left to right and then top to bottom, showing all intermediate states occurring throughout the registration process. The extract from the final moving image, the result deemed registered by the agents, is shown in the lower right. One step per agent is executed for each image in sequence. The full fixed image I_f and detail alignments are shown in Figure 5.3. Base images are from the Human Connectome Project [8].

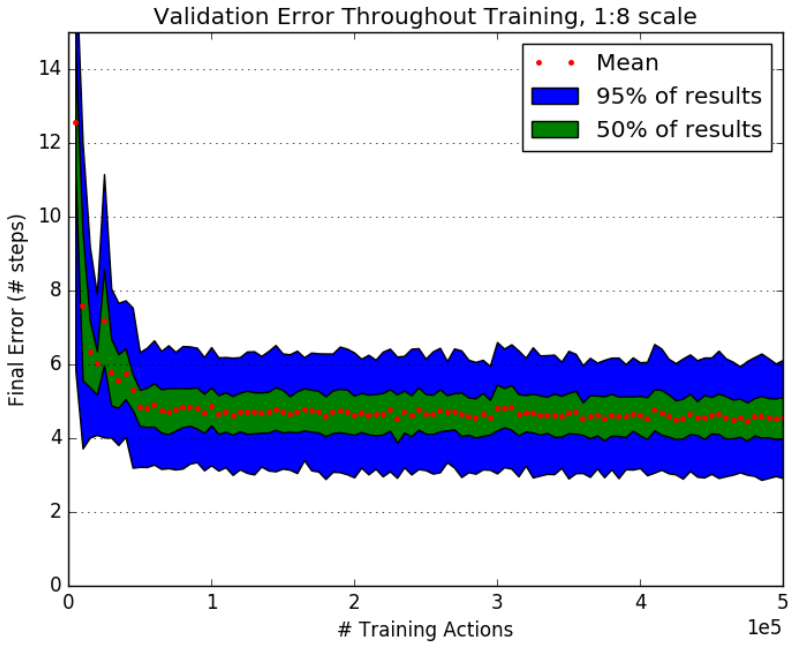

Figure 5.5 Extended training for 1:8 scale multimodal T1/T2. Axes and legend definitions are the same as in Figure 5.2.

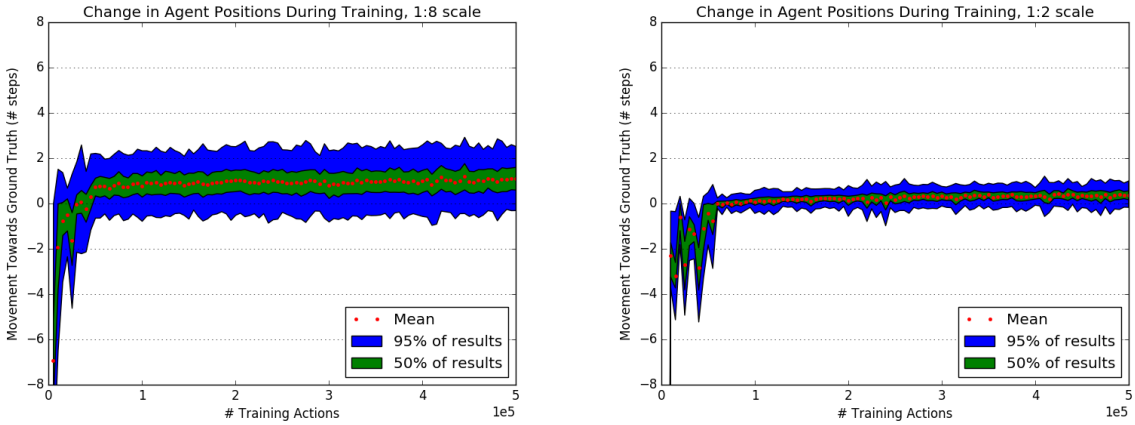

Figure 5.6 Extended training for multimodal T1/T2. Shows the distribution of the change in agent positions throughout validation for 1:8 scale (left) and 1:2 scale (right). It may be observed that, aside from ceasing early negative performance, throughout most of the training process the 1:2 scale agent causes little to no net change in error and performance improves at an incredibly slow rate relative to other tests. In contrast, the 1:8 scale agent causes consistent reduction of error and, while improvement is still slower than the monomodal case, as training progresses said improvement is faster than with the finer scale. Vertical axis measures the total change in error between the beginning and end of each registration attempt, where the error is calculated by taking the mean of the per-agent differences between final transform parameters and ground-truth transform parameters. Step size = 1 pixel/step, so given error is in both steps and pixels. Horizontal axis measures the number of actions taken before the corresponding validation test was performed, where each such action is the aggregate of all agent actions at a given timepoint. More training actions also result in more SGD updates to the network weights, due to the structure of training detailed in Chapter 4. The red dots indicate the mean change in error for all attempts in the validation tests performed after the indicated amount of training. The green region indicates the range of changes observed in the 50% of tests with change-in-error levels closest to the mean change. The blue region is as the green region, but for 95% of the results.

5.6 Training vs Validation Performance

The sequence of changes in agent performance that occur throughout the learning process varies depending on scale and modality tested, but in each case it was observed to follow the same pattern for both the training and validation sets. The transition from rapid improvement during early training to slower improvement would occur at the same time for both datasets, and if a plateau of consistent error was observed it would in every case occur for both datasets. Some example graphs are shown in Figure 5.7.

This is noteworthy, as it shows overfitting is unlikely to be occurring. When the agent learns a policy from the training set it is reflective of the underlying problem and generalizes to the validation set. Further, with the higher resolution multimodal agents for which a functional policy is not learned, it is difficult to argue that the algorithm is overfitting the training data when it is hardly fitting that data at all. The failure of the algorithm to learn anything useful in that exact context, despite functioning when given the same task with reduced image detail (lower resolution multimodal) or given a simpler variation of the task with the same level of image detail (high resolution monomodal), may suggest that the failure is related to insufficient network capacity. In any case, such errors are not caused by overfitting.

In discussing these results, it is important to explain the seemingly counter-intuitive error relation in Figure 5.7, in which training set error is most often higher than validation set error. As explained in Section 4.2.2, epsilon-greedy action selection is used during training set registrations as part of the learning process, while greedy action selection is used for validation set registrations to monitor performance. This increases the measured error for training set registration, as seen in the results, but would not change the behaviour noted above regarding improvement or lack thereof.

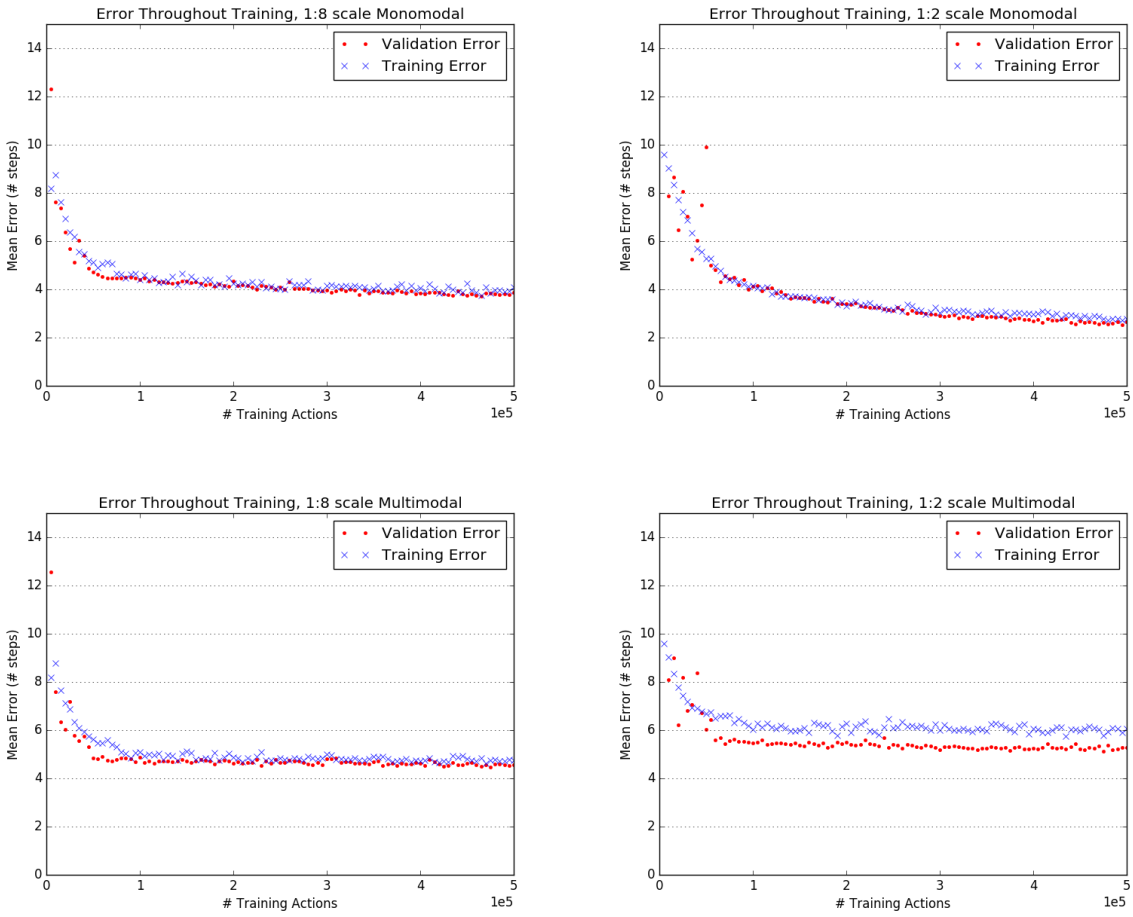

Figure 5.7 Accuracy for training and validation sets during extended training. Top row shows results for monomodal T1, from two training sequences also shown in Figure 5.2. Bottom row shows results for multimodal T1/T2, from the training sequences shown in Figure 5.6. The red dots indicate the mean value of all measured errors from validation tests performed after the indicated amount of training. The blue crosses indicate the mean value of all measured errors during the training registrations undertaken between the indicated step number and and the previous validation test loop. Axes definitions are the same as in Figure 5.2.

5.7 Limitations and Possible Redress

5.7.1 Training Time

As training was performed on a shared GPU with inconsistent background load, the time required for any given attempt would vary significantly. Tests of the duration shown in Figures 5.2 and 5.5–5.7 were observed to take between 20 and 110 hours to finish training, with a median length of approximately 40 hours. Inconsistent training times were observed both between different tests, and between subsequent executions of the same test. This large variation in training times reduces the utility of estimates of the real-time duration of future training.

As mentioned in Section 5.4, the monomodal agents for 1:8 and 1:4 scale did not reach the desired level of accuracy, but neither did their performance reach a plateau. Consistent improvement was observed during the latter portion of their training, and at the rate they were improving it seems likely that extending that training to five times the original duration would result in the desired accuracy. One must recall that the agents of these scales do not need to be exceptionally accurate, just "close enough" to consistently be within the next agent's capture radius. Therefore with 160–880 hours of further training time it should be possible to perform coarse-to-fine monomodal registration.

It is less clear how much additional time training the 1:8 scale multimodal agent may take. While improvement was observable, the small magnitude and slow rate of change are such that it would be unreasonable to extrapolate the trend far enough to make any meaningful time estimate. Observing that the agent was able to improve is promising, given the performance of the other multimodal agents, but more significant changes may be necessary to achieve desirable results.

5.7.2 Network Capacity

For the multimodal agents it would be worthwhile to investigate the effects of increased network capacity. As discussed briefly in Section 5.6, the system is capable of training monomodal agents and low resolution multimodal agents with at least some degree of success. This suggests there is no specific and singular component of the task that is incompatible with the training method, and that the failure to train higher resolution multimodal agents comes from combining the complexity of increased resolution and multimodality.

This combination results in the most complex visual relationships of all test cases examined here, and therefore requires that the network store the most information. At a minimum it becomes necessary for the agent to learn twice as many low level feature representations as it would in the monomodal case. That failure occurs when the amount of information to store increases, while the basic nature of that information does not change, suggests that the network is incapable of storing the additional information.

The most direct method of increasing the network's capacity would be to increase its size and number of connections. That most of the increased complexity is related to low-level visual representation suggests that increasing the number of kernels in the first few convolutional layers would be most relevant. One may estimate that learning feature representations of two modalities will require approximately twice as many kernels as learning the same information for one modality, assuming both modalities are of a similar level of visual complexity. However, due to the structure of the network (outlined in Section 4.1.2), doubling the number of kernels in the first three convolutional layers increases the memory used by the network to 195% of its prior requirements. This change was unfortunately impractical to implement given the other requirements of the algorithm. Between the agent's network, the Experience Replay Buffer, and the images involved, in addition to inconsistent background GPU use, there was precious little spare GPU memory to work with. Any enhancement made to the network would be at the expense of another component, and each such reduction would require testing to ensure it did not degrade performance more than the bolstered network might aid it. This testing is simple, but would have required more time than was available.

Chapter 6

Conclusions and Future Work

The results in Chapter 5 suggest that the overall framework discussed in Chapter 4 could become a useful method of medical image registration, but further work is necessary before reaching that point.

The success of higher scale monomodal tests shows that subdividing global registration into a collection of local registration problems is a reasonable approach to simplifying the challenge of non-rigid registration. A single deep-RL agent was found to be capable of registering any arbitrarily selected region, and multiple instantiations of that agent operating in parallel and communicating were able to achieve complete registration. The inherently parallel nature of the method should also scale well with additional hardware resources.

Coarser scale tests for both monomodal and multimodal are less positive, but still suggest a path forward. Coarse scale agent training functions, and continues improving throughout all observed training without reaching a performance plateau. This suggests failure to reach the capture radius of subsequent scales is not necessarily due to an inherent limitation of the method, and may simply be due to insufficient available training cycles. With more time, or more powerful hardware, it is plausible that adequate performance could be achieved for these scales. The coarse scale monomodal cases in particular are close to functioning, and should reach adequate performance with a feasible increase in training time. As final accuracy is determined by the accuracy of the highest scale agent, merely "adequate" performance of these coarse layers should not be a problem for overall system performance.

Higher scale multimodal registration is the most concerning result, due to the near lack of observable improvement. However, the more comparable performance of monomodal and multimodal agents at coarser scales suggests that the failure of less coarse multimodal registration is not due to an essential difference between multimodal and monomodal registration that these methods cannot overcome, but rather that at high detail levels the additional complexity of the multimodal case may surpass what this exact parameterization of the model is capable of handling. Given that training agents to handle full scale monomodal registration is already quite taxing on available resources, if multimodal registration is possible it is likely unachievable without more powerful hardware. It is not certain that the method can work for this application, but it merits further investigation. And, while potentially expensive, said investigation should be relatively simple from a research perspective.

For similar reasons, 3D image registration remains unaddressed. At higher resolutions GPU memory limitations prevent an experience replay buffer of sufficient size if it must accommodate 3D image patches and 3D convolutional networks. However, there is no part of the proposed methods that are inherently tied to a specific dimensionality, and neural-net image analysis has been shown to adapt well to 3D data. With sufficient hardware resources it should be viable to attempt this method for 3D registration.

If one were to make a final conclusion summarizing these results, it would be that this application of reinforcement learning to medical image registration seems to have potential, and merits further investigation. It was demonstrated to work in a very limited circumstance (partial coarse-to-fine monomodal registration), and the tests performed suggest that simply allocating more time may be enough to resolve the majority of those limitations. Multimodal registration is less certain, but should more powerful hardware be available it is similarly worth investigating. The results shown here serve less to prove this exact implementation of the method is superior to other approaches, but rather to argue in favour of further attention.

Bibliography

[1] A. Sotiras, C. Davatzikos, and N. Paragios, "Deformable medical image registration: A survey," *IEEE transactions on medical imaging*, vol. 32, no. 7, p. 1153, 2013.

[2] L. G. Brown, "A survey of image registration techniques," *ACM computing surveys (CSUR)*, vol. 24, no. 4, pp. 325–376, 1992.

[3] T. Arbel, X. Morandi, R. M. Comeau, and D. L. Collins, "Automatic non-linear mri-ultrasound registration for the correction of intra-operative brain deformations," *Computer Aided Surgery*, vol. 9, no. 4, pp. 123–136, 2004.

[4] M. Simonovsky, B. Gutiérrez-Becker, D. Mateus, N. Navab, and N. Komodakis, "A deep metric for multimodal registration," in *International conference on medical image computing and computer-assisted intervention*, pp. 10–18, Springer, 2016.

[5] G. Haskins, U. Kruger, and P. Yan, "Deep learning in medical image registration: A survey," *arXiv preprint arXiv:1903.02026*, 2019.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[7] F. C. Ghesu, B. Georgescu, T. Mansi, D. Neumann, J. Hornegger, and D. Comaniciu, "An artificial agent for anatomical landmark detection in medical images," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 229–237, Springer, 2016.

[8] D. C. van Essen, K. Ugurbil, E. Auerbach, D. Barch, T. Behrens, R. Bucholz, A. Chang, L. Chen, M. Corbetta, S. W. Curtiss, *et al.*, "The human connectome

project: a data acquisition perspective," *Neuroimage*, vol. 62, no. 4, pp. 2222–2231, 2012.

[9] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens, "Multimodality image registration by maximization of mutual information," *IEEE transactions on Medical Imaging*, vol. 16, no. 2, pp. 187–198, 1997.

[10] D. De Nigris, D. L. Collins, and T. Arbel, "Multi-modal image registration based on gradient orientations of minimal uncertainty," *IEEE transactions on medical imaging*, vol. 31, no. 12, pp. 2343–2354, 2012.

[11] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool, "A comparison of affine region detectors," *International journal of computer vision*, vol. 65, no. 1-2, pp. 43–72, 2005.

[12] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," *Image and vision computing*, vol. 22, no. 10, pp. 761–767, 2004.

[13] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2, pp. 1150–1157, IEEE, 1999.

[14] L. Wilkinson, *The Grammar of Graphics*. Springer Science+Business Media, 2005.

[15] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. Hill, M. O. Leach, and D. J. Hawkes, "Nonrigid registration using free-form deformations: application to breast MR images," *IEEE transactions on medical imaging*, vol. 18, no. 8, pp. 712–721, 1999.

[16] B. Lorensen, "Example B-spline deformation of a liver CT, from https://www.slicer.org/wiki/file:bsplexample_grid_animgif.gif," 2010.

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*. Cambridge University Press, 1992.

[18] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 295–307, 2016.

[19] B. Fischer and J. Modersitzki, "Ill-posed medicine—an introduction to image registration," *Inverse Problems*, vol. 24, no. 3, p. 034008, 2008.

[20] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*, pp. 7–15, Springer, 1987.

[21] E. I. George and R. E. McCulloch, "Variable selection via gibbs sampling," *Journal of the American Statistical Association*, vol. 88, no. 423, pp. 881–889, 1993.

[22] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[23] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[24] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, p. 947, 2000.

[25] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[26] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, 2013.

[27] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[28] G. Litjens, T. Kooi, B. Ehteshami Bejnordi, A. Setio, F. Ciompi, M. Ghafoorian, J. van der Laak, B. van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical Image Analysis*, vol. 42, 02 2017.

[29] G. Wu, M. Kim, Q. Wang, Y. Gao, S. Liao, and D. Shen, "Unsupervised deep feature learning for deformable registration of mr brain images," in *International Confer-*

ence on Medical Image Computing and Computer-Assisted Intervention, pp. 649–656, Springer, 2013.

[30] Q. V. Le, W. Zou, S. Yeung, and A. Y. Ng, "Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis," 2011.

[31] D. Shen, "Image registration by local histogram matching," *Pattern Recognition*, vol. 40, no. 4, pp. 1161–1172, 2007.

[32] S. Miao, Z. J. Wang, and R. Liao, "A cnn regression approach for real-time 2d/3d registration," *IEEE transactions on medical imaging*, vol. 35, no. 5, pp. 1352–1363, 2016.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[34] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, vol. 2, p. 5, Phoenix, AZ, 2016.

[35] H. van Seijen, M. Fatemi, J. Romoff, and R. Laroche, "Improving scalability of reinforcement learning by separation of concerns," *arXiv preprint arXiv:1612.05159*, 2016.

[36] R. Liao, S. Miao, P. de Tournemire, S. Grbic, A. Kamen, T. Mansi, and D. Comaniciu, "An artificial agent for robust image registration," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[37] K. Ma, J. Wang, V. Singh, B. Tamersoy, Y.-J. Chang, A. Wimmer, and T. Chen, "Multimodal image registration with deep context reinforcement learning," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 240–248, Springer, 2017.

[38] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[39] J. Krebs, T. Mansi, H. Delingette, L. Zhang, F. C. Ghesu, S. Miao, A. K. Maier, N. Ayache, R. Liao, and A. Kamen, "Robust non-rigid registration through agent-based action learning," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 344–352, Springer, 2017.

[40] S. Miao, S. Piat, P. Fischer, A. Tuysuzoglu, P. Mewes, T. Mansi, and R. Liao, "Dilated fcn for multi-agent 2d/3d medical image registration," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[41] L.-J. Lin, "Reinforcement learning for robots using neural networks," tech. rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[42] D. Ziou, S. Tabbone, *et al.*, "Edge detection techniques-an overview," *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii*, vol. 8, pp. 537–559, 1998.

[43] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A MATLAB-like environment for machine learning," in *BigLearn, NIPS workshop*, 2011.

Appendix A

Implementation Details

A.1 Code Structure

The implementation of the above algorithms is carried out using a modified and extended version of the Google DeepMind deep Q-network [33]. Code is written primarily in Lua, using Torch 7 [43] as a backend. Certain speed-critical image resampling code is written in C to create a Torch module.

An overview of the code structure is shown in Figure A.1. It illustrates the entry points, as well as library dependencies.

A.2 Code Description

Entry Points

ex_train_one_agent.lua This executable script is invoked when training an agent to process images at a single scale. It uses `train_agent.lua` to perform this training.

ex_train_scale_agents.lua This executable script is invoked when training multiple agents to process images at each scale configured within `opts_agent.lua`. It iteratively uses `train_agent.lua` to perform this training, calling it with different options each time.

ex_test_reg.lua After the necessary networks have been trained, this script is used to test registration. It registers the specified images in a coarse-to-fine manner and evaluates the system's performance.

Figure A.1 Code structure, with entry points at top and denoted by triangular component

ex_scan_gamma.lua This is an executable testing script which trains a separate network for each of many possible γ values. It is used to observe the effect of γ on performance.

ex_scan_LR.lua Similar to `ex_scan_gamma.lua`, this script examines learning rate.

High-Level Control

initenv.lua This is an initialization function which is run once at startup to perform initial setup. It loads the appropriate libraries, configures the GPU, and initializes the DQN.

opts_agent.lua Configuration options are stored in this file. It controls all training, agent, and network options not defined in `convnet_2d_a.lua` or `convnet_2d.lua`.

train_agent.lua The main agent training loop is found in this library. It is used to train an agent, evaluate its performance, and log relevant training metrics. Via `initenv.lua`, an agent is instantiated from `NeuralQLearner.lua` and an environment from `splinewrap.lua`, after which this library manages the interactions between them.

c2f_reg.lua After the required agents have been trained, this library is used to perform coarse-to-fine registration. It loads the pre-trained networks and the provided images, and then performs registration at each scale in order of increasing resolution, using the end results of each as the starting point for the next.

Neural Net Control

NeuralQLearner.lua This is the main interface for the Q-value predicting neural network. Its main role is to predict action values from state information and to update network weights during training.

parallel_TT.lua This acts as a multiplexer, dividing state/action/reward data from parallel actors into sequences of individual agent actions, and sampling those action histories to provide training data. Memory management is perfomed by `TransitionTable.lua`.

TransitionTable.lua The Experience Replay Buffer is managed by this library, which is used to store state, action, and reward history during training. It provides sampling functions for `NeuralQLearner.lua` to use when updating network weights.

convnet_2d_a.lua A network shape is defined by this configuration file. It controls the number and size of convolutional and fully connected layers used when agents are configured to use local and neighbour action history as a component of the state vector.

convnet_2d.lua This configuration file is similar to `convnet_2d_a.lua`, but defines the network shape when not considering action history.

convnet_acthist.lua When setting up the network defined by `convnet_2d_a.lua`, this initializes and connects the neural network layers using Torch.

convnet_orig.lua This is as `convnet_acthist.lua`, though for when using `convnet_2d.lua`

Rectifier.lua This is an implementation of a Rectified Linear Unit activation function for use with Torch.

nnutils.lua This library provides functions for examining a neural net. It is mainly used for logging network weight and gradient statistics during training.

preproc_affineScale.lua `NeuralQLearner.lua` uses preprocessing filters to perform basic operations on the image component of the DQN's input state vector. This filter may be applied to image patches that would be the wrong size, and scales them such that they have the correct dimensions.

preproc_blur.lua This preprocessing filter blurs the input patches.

preproc_passthrough.lua This preprocessing filter does not modify the input data at all. This is the default filter.

scale_2channel.lua Performs the calculations for `preproc_affineScale.lua`

gblur_2channel.lua Performs the calculations for `preproc_blur.lua`

Image Management

splinewrap.lua A reinforcement learning friendly interface to the registration process is provided by this library. It does this by converting actions selected by the agent into motion commands for `gpu_img_pair.lua`, calculating appropriate rewards, and

tracking action histories. It also handles re-initializing the registration environment as appropriate during training.

datasplit.lua File management is abstracted by this library. It reads images from folders specified in the configuration, pairs them together, and divides them into testing/training/validation sets. It is then invoked by `splinewrap.lua` to randomly select images from the appropriate sets.

gpu_img_pair.lua This library performs image-pair management. It is used to load image pairs into memory, extract patches for the DQN to process, and apply transforms to the moving image via `gpu_bspline_T.lua` or `gpu_affine_T.lua`.

gpu_bspline_T.lua This is a b-spline transform management library. It tracks parameters as they are updated and calculates dense transform matrices as necessary, applying them to images using the `gimg` module.

gpu_affine_T.lua This library is the equivalent of `gpu_bspline_T.lua` for affine transforms.

gimg module (external) This Lua module uses the GPU to quickly apply a dense transform to an moving image.

A.3 Libraries and Environment

The code was developed and tested on an Ubuntu platform. LuaJIT 2.0.4 was used to execute the main body of code, and Torch 7 was used to for neural net implementation. CUDA 8.0.61 was used for GPU acceleration, and execution was performed on an Nvidia TITAN Xp.

The following Lua modules were included as part of Torch or were otherwise required for its operation: cunn scm-1, cutorch scm-1, nn scm-1, nngraph scm-1, cwrap scm-1, dok scm-1, env scm-1, gnuplot scm-1, graph scm-1, image 1.1, luaffi scm-1, luafilesystem 1.7.0, moses 1.6.1, paths scm-1, penlight 1.5.4, sundown scm-1, sys 1.1, xitari, xlua 1.1.

The following additional software packages were required by LuaJIT, Torch, or a Lua module: build-essential 12.1, gcc 5.3.1, g++ 5.3.1, cmake 3.5.1, curl 7.47, libreadline-dev 6.3, git-core 2.7.4, libjpeg-dev 8c, liblua5.1-dev 5.1.5, libpng-dev 1.2.54, ncurses 6.0, imagemagick 6.8.9.9, unzip 6.0 .

Note that while this list covers the notable requirements, it is not a comprehensive list of all software and libraries used on the development machine.

A.4 Modifications

The core library for managing the Q-value predicting neural net, NeuralQLearner.lua, was updated in a number of ways. Interaction with the experience replay buffer was altered to support multiple parallel agent instances, both during training and execution. Recent action history of each instance was added as part of the state considered by the network, as well as the action history of agent instances responsible for neighbouring control points. A function was also added to monitor action scores (rather than simply recording the best action's index) to aid in development and hyperparameter tuning. A double DQN was tested, but later removed as it did not have a noticeable impact on performance.

To facilitate these changes, the neural net structure defined in convnet_acthist.lua was also altered. Net input is split into the state's image history component, consisting of patches surrounding the control point's position extracted from the fixed and moving images, and the state's action history component, a one-hot encoding representation of the most recent actions taken by that control point and its neighbours. The image history is fed through a configurable number of convolutional layers. The output of these convolutional layers is then concatonated with the encoded action history and fed through a configurable number of fully connected layers. The number of previous steps considered is set independently for each history type (image patches, actions, and neighbour actions), and as they determine the net's size they cannot change once training begins.

As additional state information was being used, it was necessary to augment TransitionTable.lua to also track action history. Due to the very small size in memory of the action history data relative to image data, the full action history was stored for each state entry. This increased buffer sampling speed relative to reconstructing the action history from stored single actions, while negligibly increasing the size per state entry. However, the original method of assembling history from sequential entries is still used for the state's image history component. This is somewhat slow, as all potential entries are tested for run termination to ensure a run shorter than the desired history length isn't causing the buffer to accidentally incorporate state data from a distinct run that is stored nearby in RAM.

It is necessary, though, due to the relative size of the image data and the impracticality of guaranteeing minimum run length.

The agent training loop provided by train_agent.lua has been altered to no longer be the main and only entry-point for execution, as post-training registration is a desired goal. Additionally, a maximum run length has been introduced to prevent a single training run from getting "lost" in improbable regions, stuck in a loop and never self-terminating, and giving undue importance to useless data by filling the whole buffer. Improved logging was added to track training/validation accuracy, and optionally step-by-step saving of the moving image, per-agent state, and action Q-scores. Control-point masking was also added, to allow ignoring data from agents far away from the region of interest (e.g., in the empty background of an MRI).

A.5 New Code

As the original DQN system was designed to interact with Atari games, it was necessary to fully replace the "environment" simulation with an image deformation and comparison system, as well as to parallelize interaction to allow multiple agents to update it simultaneously. This primary agent/environment is performed by splinewrap.lua, which instantiates, loads, and interfaces with image management module (gpu_img_pair.lua); tracks action history; collates state/reward information returned by said module; and tracks action history.

Image management is overseen by gpu_img_pair.lua, which handles loading image files, extracting pixel patches to represent a specific agent's "state", and calculating action rewards. Reward values are only used during agent training, in which case the pre-aligned training images have an initial deformation applied when loaded, and the reward is calculated based on whether the action reduces or increases that initial error. Image deformation is performed by gpu_affine_T.lua or gpu_bspline_T.lua, depending on the type of transform being used. B-spline is the default, with affine used primarily as a point of comparison.

With the parallel execution of agents during training it was necessary to create parallel_TT.lua. This largely serves as a wrapper for TransitionTable.lua, creating and managing a separate TransitionTable buffer for each agent instance. When a Q-network update is required, parallel_TT.lua is responsible for sampling agent experiences from the separate buffers and merging them appropriately.

File management is handled by datasplit.lua. The class pairs images of the same subject under different modalities, and splits the resulting pair list into training and testing sets.

While the original codebase set all training options in a short startup script, the increased number of parameters made this unwieldy, and therefore configuration is now handled by opts_agent.lua. The startup script was replaced with ex_train_one_agent.lua and ex_train_scale_agents.lua, to train an agent for a single scale or for all targeted scales, respectively. After training is complete, c2f_reg.lua may be used to perform registration.

While developing the system, the ex_scan_*.lua scripts were used to tune hyperparameter values. Each script examines one hyperparameter, and performs agent training multiple times while varying that parameter. Performance of the trained agents is then compared to select the best hyperparameter value.

Appendix B

Hyperparameter Analysis

The tests discussed in this chapter show results recreating the final cycle of the hyperparameter refinement process mentioned in 5.3. One test is performed for each hyperparameter, training agents using different values for that hyperparameter covering its valid range and examining the changes in agent performance. The structure of each test is as described in Section 5.1.1. Due to training time constraints, this full sweep is only performed for the 1:8 scale agents. Results may be compared to tests in Section 5.4 which show the effect of extended training and the diminishing returns in performance improvement as training time increases.

Training and execution time is discussed for tests in approximate and relative terms, and the number of training steps taken is often used in place of exact time measurements. This is because the nature of the testing environment makes it impossible to reliably measure precise times, as parallel testing and multiple users result in varying background load between tests. Large and obvious changes in required time were noted, but more precise analysis would require different experimental context.

B.1 Reward Discount

Within the valid range of γ, from 0 to 1, for most applications one may reasonably expect the resulting agent's performance to be poor at either extreme and to have some ideal value between the two. Excessively low values of gamma tend to result in a short-sighted hill-climbing algorithm which take actions prone to getting stuck in a shallow local optimum,

Test	Values tested
γ	From 0.1 to 0.9, increments of 0.1
α	10^x for $x = \{-7, ..., 0\}$
$\|ERB\|$	$\{256, 10^3, 4 \cdot 10^3, 10^4, 2 \cdot 10^4, 4 \cdot 10^4, 8 \cdot 10^4, 2 \cdot 10^5\}$
$\|P\|$	11x11, 15x15, 25x25, 41x41, 61x61
$\|H_{act}\|$	From 2 to 20, increments of 2
$\|H_{nghbr}\|$	From 2 to Len_{AH}, increments of 2
ϵ_{final}	From 0 to 0.5, increments of 0.1
N_ϵ	20% to 100% of full training time, increments of 20%
$r_{sm,final}$	From 0 to 0.5, increments of 0.1
$N_{r,stop}$	Same as $Nstep_\epsilon$
Rad_{any}	From 2 to 12, increments of 2

Table B.1 Experiment summary.

while excessively high values may have the opposite effect and result in overlooking a desirable solution.

These tests serve to scan the range of possible values and find an appropriate γ for this application.

B.1.1 Effect on Registration Error

Lowest error rates were observed at $\gamma = 0.3$, though performance was similar for $\gamma \leq 0.8$. Somewhat surprisingly low values of γ do not significantly degrade performance, as might be expected from a "short-sighted" agent strongly focused on immediate rewards.

As well as having the lowest mean error, $\gamma = 0.3$ has the lowest 95% upper confidence bound. Certain higher γ values have similar mean errors, but all have higher upper confidence bounds, suggesting that while performance is often similar between the options it is less consistently achieved with higher γ. Due to the risks associated with unexpected errors in a medical context, this lower upper bound is most likely to be preferable.

B.1.2 Effect on Training and Execution Speed

For lower γ values, validation accuracy during training is observed to more rapidly approach its final performance level, and fewer late-training accuracy fluctuations occur. This is readily apparent when comparing the results in Figure B.1.

Similarly, low values of γ result in a lower mean number of steps for registrations, with

γ	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
0.1	4.49	5.95	3.07	9.78
0.2	4.45	5.95	2.93	9.33
0.3	4.27	5.71	2.86	8.73
0.4	4.39	5.93	2.83	9.51
0.5	4.41	5.97	2.91	9.58
0.6	4.37	5.92	2.89	9.60
0.7	4.43	5.95	2.88	10.46
0.8	4.35	5.77	2.94	10.11
0.9	4.62	6.27	3.00	12.18

Table B.2 Effect of γ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error" is the mean of the individual registration attempt errors, where the error of each attempt is calculated by taking the mean of the per-agent differences between final transform parameters and ground-truth transform parameters. The 95% of registration errors falling closest to the median error were extracted, an the range of errors within this subset is shown by "95% Confidence Upper/Lower Bound" columns. The measures in these first three columns are all given in number of pixels. "Mean # Steps per Registration" is the mean number of global actions taken while performing each of these final validation registrations, where each global action consists of the aggregate of all agents' local actions. For registrations of equal accuracy, a higher number of steps implies a more circuitous and less efficient route was taken.

$\gamma = 0.3$ resulting in the lowest number of steps and therefore the fastest registration. The difference is not particularly large in most cases, though, with most values differing by less than one step. The difference becomes greater at higher values, and $\gamma \geq 0.7$ is notably slower.

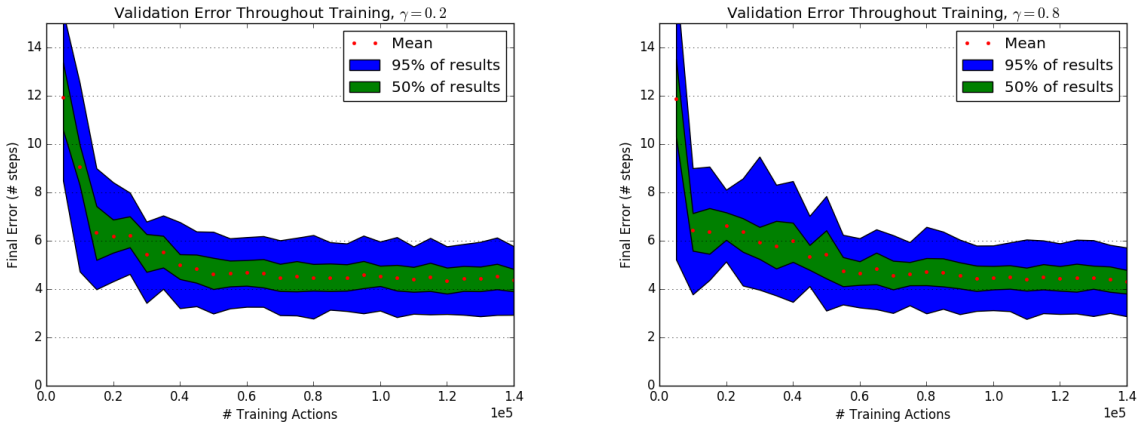

Figure B.1 Validation error logs for γ tests, showing the faster learning rate associated with lower values ($\gamma = 0.2$, left) as compared to higher values ($\gamma = 0.8$, right). Vertical axis measures per-attempt error, where the error of each attempt is calculated by taking the mean of the per-agent differences between final transform parameters and ground-truth transform parameters. Step size = 1 pixel/step, so given error is in both steps and pixels. Horizontal axis measures the number of actions taken before the corresponding validation test was performed, where each such action is the aggregate of all agent actions at a given timepoint. More training actions also result in more SGD updates to the network weights, due to the structure of training detailed in Chapter 4. The red dots indicate the mean value of all measured errors for validation tests performed after the indicated amount of training. The green region indicates the range of errors observed in the 50% of tests with error levels closest to the mean test error. The blue region is as the green region, but for 95% of the results.

B.2 Learning Rate

The effects of varying learning rate are fairly predictable. Low α values lead to insufficient change with each learning step, as shown in Figure B.2. While these examples may eventually converge, they waste time by unnecessarily prolonging training.

As α increases training occurs more quickly, but eventually the system becomes unstable. For values slightly too large the network parameters will fluctuate around their ideal solution, reducing accuracy, and at even higher levels the fluctuations will increase in amplitude at each step until they are arbitrarily far from a solution.

α	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
10^{-7}	5.60	7.52	3.67	1.00
10^{-6}	5.41	7.23	3.65	5.98
10^{-5}	4.27	5.71	2.86	8.73
10^{-4}	4.51	6.21	2.84	48.06
10^{-3}	4.42	6.17	2.87	72.46
10^{-2}	-	-	-	-
10^{-1}	-	-	-	-
1	-	-	-	-

Table B.3 Effect of α on validation error. "-" indicates a result that was not stable enough to reliably complete validation without calculation errors. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.2.1 Effect on Registration Error

For a fixed number of training steps, a value near $\alpha = 10^{-5}$ was found to be produce the most accurate results (Figure B.2). Lower values were slower to train, while higher values resulted in greater error and fluctuating performance throughout training. For $\alpha \geq 10^{-2}$ training is non-functional.

B.2.2 Effect on Training and Execution Speed

Increasing α reduces training time so long as $\alpha \leq 10^{-3}$. Greater values of α do not slow training, but rather prevent training from functioning at all. As α is a parameter that is only used during training, it does not directly affect execution time. However, higher values of α increase the number of steps taken during registration, indirectly slowing the process. This increase in step count is particularly high for $\alpha \geq 10^{-4}$, with agent performance degrading as α approaches the non-functional range of $\alpha \geq 10^{-2}$.

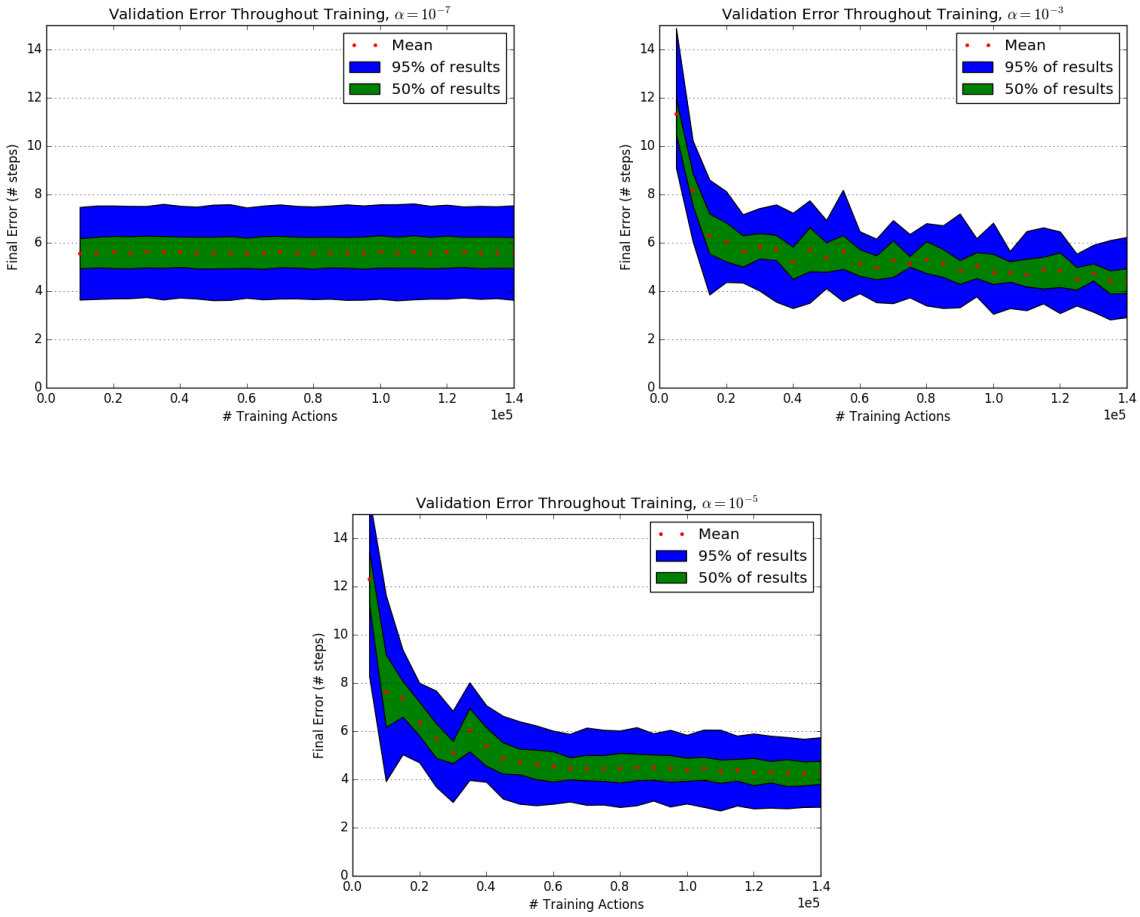

Figure B.2 Validation error logs for α tests. The best performance among tested values was observed with $\alpha = 10^{-5}$ (bottom). For lower values ($\alpha = 10^{-7}$, upper left) error did not improve much or was static throughout training, while for higher values ($\alpha = 10^{-3}$, upper right) there were recurring fluctuations and higher final errors. Axes and legend definitions are the same as in Figure B.1.

B.3 Buffer Size

The Experience Replay Buffer serves an important role in forcing the target value function to seem static, rather than varying with each new image pair. Too small of a buffer may lead to instability, as the learning algorithm may overfit to recent data, "forgetting" earlier experiences. Increased buffer size is expected to improve performance for all feasible values. It is possible for an excessively large buffer to slow training time, as old data would take a long time to be cycled out, but this requires more memory than is available in all but perhaps the most extravagant of modern hardware.

One purpose of these tests is to determine if there is a point of diminishing returns within the available memory limit, beyond which increasing the buffer is less useful than increasing other memory-intensive parameters. The second purpose is to determine if low values still allow training to progress.

$\|ERB\|$ (# records)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
256	5.03	6.71	3.29	12.23
10^3	5.02	6.76	3.32	10.78
$4 \cdot 10^3$	4.66	6.30	3.07	9.76
10^4	4.53	6.03	3.10	9.70
$2 \cdot 10^4$	4.61	6.30	2.98	10.40
$4 \cdot 10^4$	4.34	5.77	2.85	9.69
$8 \cdot 10^4$	4.44	5.95	2.95	9.50
$2 \cdot 10^5$	4.45	6.01	2.86	9.90

Table B.4 Effect of $|ERB|$ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.3.1 Effect on Registration Error

Early training with a small buffer is erratic and inaccurate, as shown in Figure B.3, but does eventually settle. Increasing $|ERB|$ improves performance for the majority of the range tested, though the highest values do show a very small increase in mean registration error, as seen in Table B.4. This may be due to larger buffers reducing the influence of new

data points by making them a smaller portion of the whole, as well as retaining "old" and possibly outdated data for longer. Both of these may potentially increase required training time, though it is not clear they will limit final accuracy if allowed to train to equilibrium.

B.3.2 Effect on Training and Execution Speed

A significant portion of the effect of buffer size on training time depends on the memory management capabilities of the hardware and low-level libraries used, and in this case it was not found to have an observable effect. Large buffers may potentially slow training as mentioned above, but within the memory limits of the test system the effect was small at most.

Execution does not use an Experience Replay Buffer and therefore is unaffected by this parameter.

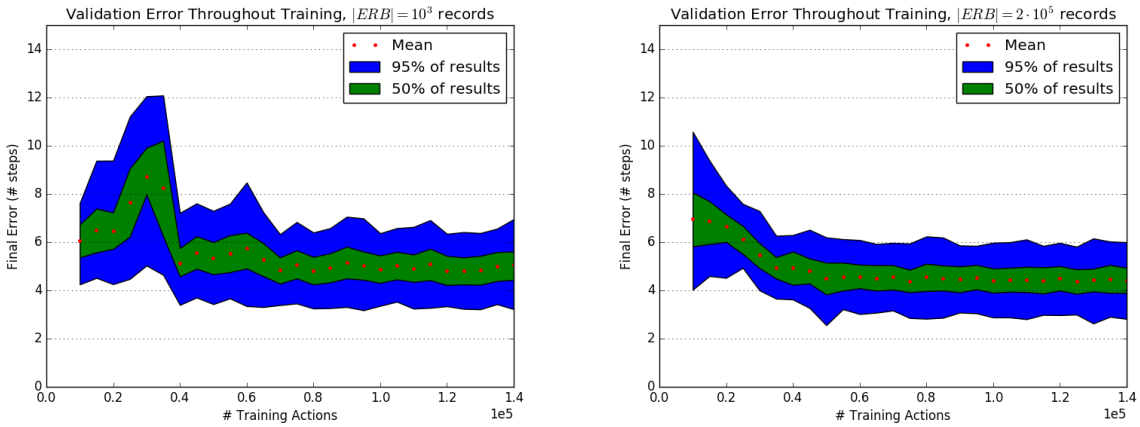

Figure B.3 Validation error logs for $|ERB|$ tests. Lower values ($|ERB| = 10^3$ records, left) show inconsistency in early training, while higher values ($|ERB| = 2 \cdot 10^5$ records, right) show smoother training results over time and improved final performance. Axes and legend definitions are the same as in Figure B.1.

B.4 Patch Size

"Patch size" refers to the size of the control-point centred sections of I_f and I_m extracted as agent inputs. It is an important parameter to balance, as larger patches use significantly more memory and result in many more network parameters that must be learned, while smaller sizes decrease the deformation range for which there is still overlap between the patches and thus likely reduces the capture radius.

The minimum patch size for these tests was 11x11, as smaller values would require reducing the kernel sizes within the convolutional layers and make the comparison less valid.

$\|P\|$ (pixels)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
11x11	4.92	6.70	3.07	8.18
15x15	4.34	5.98	2.70	9.78
25x25	4.27	5.71	2.86	8.73
41x41	4.49	5.91	3.10	9.53
61x61	4.64	6.32	2.93	11.41

Table B.5 Effect of $|P|$ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.4.1 Effect on Registration Error

Accuracy was found to be best with patch sizes of 25x25, with error increasing as patch sizes deviated further from this optimal value. The worst performance occurred with 11x11 patches. A comparison is shown in Figure B.4. Lower patch sizes were also observed to have lower error early in training, but to take more training steps before beginning to improve on this early accuracy. This seems to be caused by small-patch agents choosing "stop" far more often during early training, thereby reducing state-space exploration, combined with an increased need for exploration as each state contained less information about the environment.

B.4.2 Effect on Training and Execution Speed

As with other options that affect memory use performance will depend on the system's memory management capabilities. However, unlike ERB size, larger patches require larger sections of memory to be copied every time a buffer entry is added or a prediction is performed, slowing down execution. For example, when comparing 61x61 patches to 11x11 patches the time-per-step was found to increase by approximately 250% for training and 50% for execution. Additionally, the greater number of network parameters often requires more training steps to converge, further increasing training times.

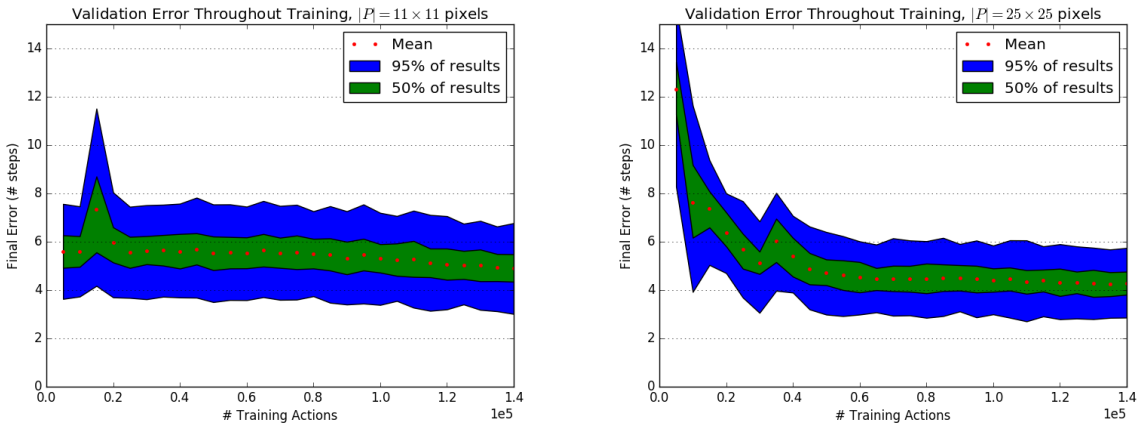

Figure B.4 Validation error logs for $|P|$ tests. Inferior performance and slower training is observed for $|P|$ = 11x11 pixels (left) compared to $|P|$ = 25x25 pixels (right). Note that for $|P|$ = 11x11 pixels accuracy does not begin to noticeably improve until approximately $8 \cdot 10^4$ training steps have passed. Axes and legend definitions are the same as in Figure B.1.

B.5 History Length

Including a history of an agent's actions in its input state data should allow it to, with sufficient training, avoid becoming trapped in action loops should some incorrect estimation of action value cause it to return to a previously visited position. Additionally, being aware of the general direction of control point movement should discourage erratic motions, which may reduce the number of suboptimal actions.

$\lvert H_{act} \rvert$ (# actions)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
2	4.60	6.23	2.97	9.62
4	4.59	6.09	3.09	8.96
6	4.37	5.84	2.89	8.47
8	4.51	6.05	2.98	8.91
10	4.34	5.82	2.90	10.07
12	4.50	6.01	3.05	9.41
14	4.39	5.84	2.94	9.78
16	4.45	6.00	2.90	9.35
18	4.35	5.87	2.78	9.48
20	4.44	5.96	2.96	9.52

Table B.6 Effect of $\lvert H_{act} \rvert$ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.5.1 Effect on Registration Error

Shorter histories where $\lvert H_{act} \rvert < 6$ were observed to have error rates that, while having a general downwards trend, would rise and fall during early training (Figure B.5). Throughout testing it was observed that the length of this early fluctuation-heavy training segment would vary with other hyperparameters. Many non-optimal combinations could greatly extend the unstable portion of training, to the point where low $\lvert H_{act} \rvert$ training was unstable for the entirety of the session. Low $\lvert H_{act} \rvert$ values were also found to learn more slowly, taking a longer time to reach similar performance.

These metrics improved as history length was increased, but after a point there was little to no change. All tested values where $\lvert H_{act} \rvert \geq 10$ showed nearly identical performance,

an example of which is shown in Figure B.5. As saved history has negligible effect on performance, $|H_{act}| = 18$ was selected rather than a value closer to that threshold. This excess should ensure that the parameter is of a sufficiently high value to maximize agent performance.

B.5.2 Effect on Training and Execution Speed

Higher values of $|H_{act}|$ were found to reduce required training time required to reach a given level of performance.

While managing long action histories may impose some burden, the tested values of $|H_{act}|$ showed little to no difference in execution time.

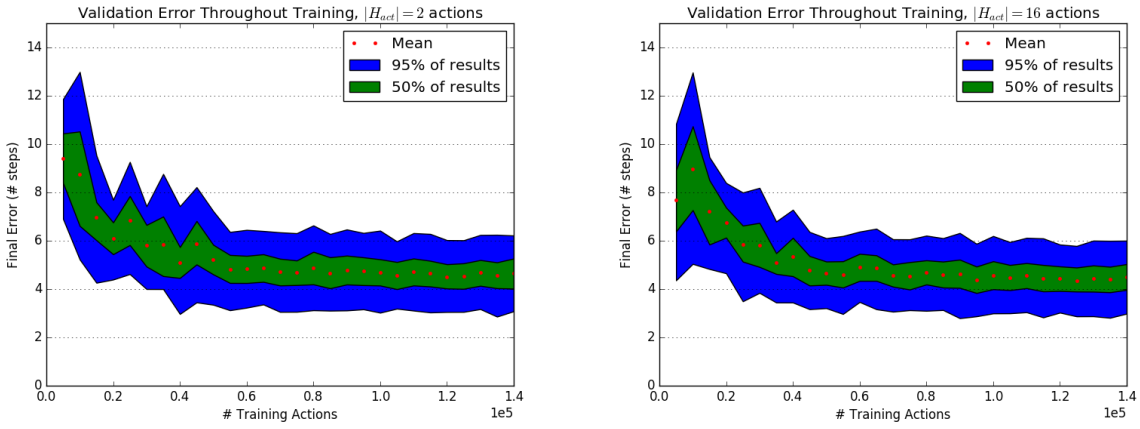

Figure B.5 Validation error logs for $|H_{act}|$ tests. Slower overall improvement and greater accuracy fluctuation early in training is observed for lower values ($|H_{act}| = 2$ actions, left) relative to agents with longer action histories ($|H_{act}| = 16$ actions, right), though overall performance does not differ dramatically. Axes and legend definitions are the same as in Figure B.1.

B.6 Neighbouring History Length

The action-history of neighbouring points is intended to improve performance when correcting for the global affine component of a misalignment. If a large region of the image containing several control points should move in a single direction, then even control points without a good action to take based on their local data should see the actions of the control points around them and be encouraged to follow the trend. This regularization should allow even low-information regions to improve to some degree.

The purpose of these tests was to determine what neighbouring history length, if any, is most productive.

$\lvert H_{nghbr} \rvert$ (# actions)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
0	4.49	6.07	2.99	9.50
2	4.46	6.04	2.91	9.04
4	4.41	5.88	2.90	8.70
6	4.29	5.83	2.73	9.68
8	4.45	5.94	2.94	9.24
10	4.43	5.85	3.01	9.36
12	4.46	6.04	2.87	9.95
14	4.27	5.71	2.86	8.73
16	4.46	5.93	3.06	10.23
18	4.40	5.95	2.84	10.11

Table B.7 Effect of $\lvert H_{nghbr} \rvert$ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.6.1 Effect on Registration Error

It was observed that changing $\lvert H_{nghbr} \rvert$ had very little effect on accuracy. There is a very minor drop in error when $\lvert H_{nghbr} \rvert$ is above zero, but the error varies to a greater degree between subsequent training runs of the same settings.

B.6.2 Effect on Training and Execution Speed

Varying $|H_{nghbr}|$ has a negligible effect on memory use and does not cause any noticeable change in training time. The number of steps taken during execution increases slightly with higher $|H_{nghbr}|$, but the change is minor and the correlation is poor.

B.7 Final Epsilon

The value of ϵ changes during training, starting at $\epsilon = 1$ and decreasing until it plateaus at ϵ_{final}. High ϵ_{final} encourages greater exploration of potential state spaces, but if too high then the explored states are far from the areas leading to a good solution and resources are wasted on useless data. These tests serve to examine the effect of these high ϵ_{final} values.

ϵ_{final}	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
0	4.45	5.91	3.01	8.17
0.1	4.46	5.96	2.93	9.01
0.2	4.42	6.02	2.83	9.25
0.3	4.44	6.10	2.76	9.70
0.4	4.38	5.90	2.88	9.35
0.5	4.36	5.86	2.85	9.80

Table B.8 Effect of ϵ_{final} on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.7.1 Effect on Registration Error

Higher values of ϵ_{final} were found to slightly increase accuracy, though the change was small and inconsistent between training sessions. This suggests either that by the time ϵ_{final} is reached the state space is adequately explored, or that any remaining inaccuracy in action prediction result in sufficient exploration via semi-random error without the need for an off-policy modification. Additionally, one may note greater fluctuation during early training for higher ϵ_{final} values, suggesting that more frequent random actions may cause the network

to deviate further from useful solution-adjacent states, "forgetting" then re-learning that information (Figure B.6). These fluctuations are noticeable for all $\epsilon_{final} > 0.3$.

B.7.2 Effect on Training and Execution Speed

Changing ϵ_{final} does not affect the per-step training time. However, for certain accuracies achievable by high ϵ_{final}, similar results may be found by training a lower ϵ_{final} agent for a smaller number of steps and therefore shorter time.

The off-policy component of the algorithm, and therefore everything directly relating to ϵ_{final}, is only used in training and therefore cannot directly affect execution.

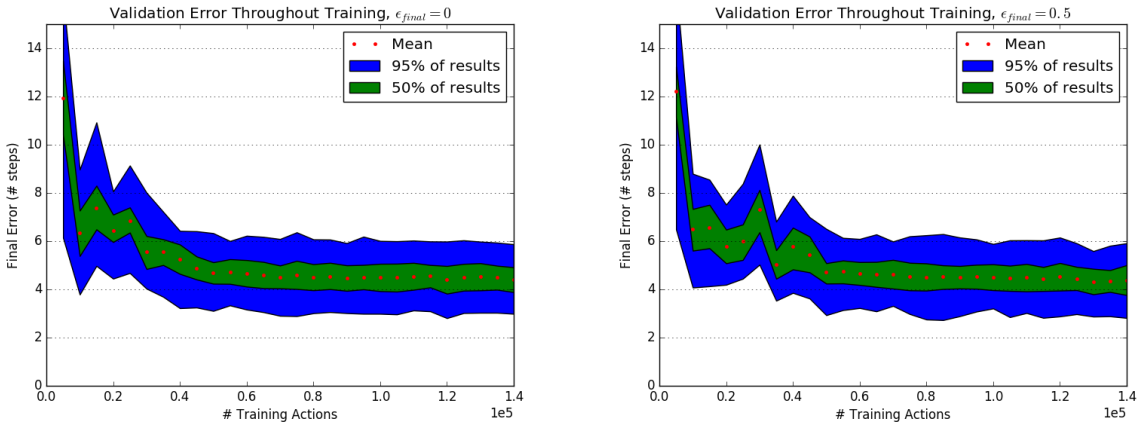

Figure B.6 Validation error logs for ϵ_{final} tests. Lower values ($\epsilon_{final} = 0$, left) show reduced fluctuation and better accuracy during early training relative to higher values ($\epsilon_{final} = 0.5$, right). Axes and legend definitions are the same as in Figure B.1.

B.8 Epsilon Ramp Length

Training begins with $\epsilon = 1$ and, over the course of N_ϵ steps,decreases linearly to $\epsilon = \epsilon_{final}$. This high initial ϵ value is intended to encourage increased exploration early on while agent performance is poor, and should prevent the agent from becoming stuck in a undesirable local performance maxima early on. Extending N_ϵ excessively is unnecessary and, at worst, may slow training. Theses tests serve to find a useful value.

N_ϵ (# steps)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
$2.8 \cdot 10^4$	4.41	5.91	2.94	9.60
$5.6 \cdot 10^4$	4.53	6.03	3.14	9.58
$8.4 \cdot 10^4$	4.52	6.09	2.95	9.18
$1.12 \cdot 10^5$	4.34	5.81	2.85	9.38
$1.4 \cdot 10^5$	4.37	5.87	2.98	9.11

Table B.9 Effect of N_ϵ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.8.1 Effect on Registration Error

N_ϵ was observed to have little effect on final registration accuracy, as the final error was similar in all tests. However, error was found to decrease much faster for lower values of N_ϵ (Figure B.7). This suggests that sufficient exploration is accomplished very early on, and keeping ϵ high for a greater portion of training simply distracts the learning algorithm with less relevant data.

It is important to note once more that $\epsilon = 0$ for all validation steps, and so ϵ-greedy action selection is not directly influencing the results. Consider the previously discussed results for ϵ_{final}, the increased error during early training is likely due to higher action randomness leading to training sample sequences with less useful information content.

B.8.2 Effect on Training and Execution Speed

Higher N_ϵ leads to longer training times due to prolonging the period of increased error associated with higher ϵ, with no observed benefit.

The off-policy component of the algorithm, and therefore everything directly relating to N_ϵ, is only used in training and therefore cannot directly affect execution.

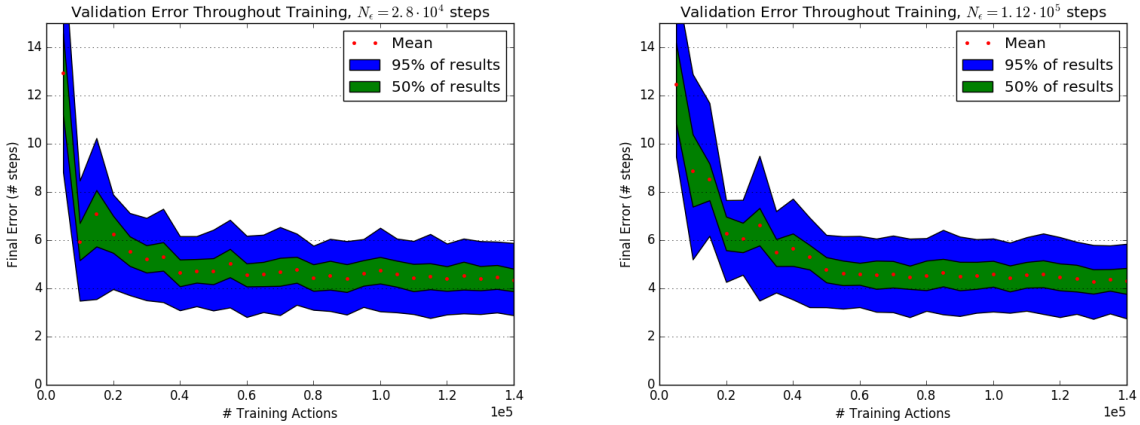

Figure B.7 Validation error logs for N_ϵ tests. Final performance is very similar for most values of N_ϵ, but early training is observed to progress more quickly for lower values ($N_\epsilon = 2.8 \cdot 10^4$ steps, left) compared to higher values ($N_\epsilon = 1.12 \cdot 10^5$, right). Axes and legend definitions are the same as in Figure B.1.

B.9 Maximum "Stop" Reward

The reward an agent receives for selecting the "stop" action during training is determined by the distance between the agent's control point and its ground-truth registered position, and when this error is smaller than the agent's step size the reward reaches its maximum value specified by $r_{sm,final}$. The value of $r_{sm,final}$ is expected to influence how aggressively the agent selects the "stop" action, with lower values encouraging more motion.

So long as the agent is more than one step away from ground truth, it is counterproductive for the "stop" reward to not be lower than the reward for making the best possible motion. This is especially true for low-γ systems. On a 2D grid with orthogonal motions the lowest optimal-motion reward occurs when the direction to ground-truth is at 45°to the motion axes, in which case $r = \sqrt{\frac{1}{2}} \approx 0.71$. High values are most likely to cause errors early in training, as an untrained network will choose actions in a nearly random manner, meaning that the expected value of motion is close to zero. Gradually increasing the stop rewards from zero to $r_{sm,final}$, similar to the ϵ ramp, may counter this to some degree.

$r_{sm,final}$	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
0	4.46	6.02	3.00	38.06
0.1	4.44	5.96	3.04	19.81
0.2	4.30	5.78	2.82	11.19
0.3	4.27	5.71	2.86	8.73
0.4	4.48	5.91	2.98	8.67
0.5	4.53	6.00	3.07	8.04
0.6	4.59	6.21	3.05	7.94
0.7	4.67	6.23	3.14	7.11
0.8	4.74	6.37	3.03	6.75
0.9	4.77	6.31	3.21	6.29
1	4.80	6.42	3.22	6.80

Table B.10 Effect of $r_{sm,final}$ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.9.1 Effect on Registration Error

As shown in Figure B.8, the optimal value of the hyperparameter was found to be approximately $r_{sm,final} = 0.3$, and final accuracy decreased if the value strayed higher or lower than this. Low values were observed to be more erratic throughout training process, especially the early stages, while high values quickly reached their final performance and improved little if at all with further training.

B.9.2 Effect on Training and Execution Speed

Below $r_{sm,final} = 0.3$ the training time required to reach a given level of accuracy is increased. Higher values seem to reach equilibrium faster, but their final accuracy is lower.

Reward values are only used during training, and therefore this parameter cannot affect per-step execution time. However, agents trained with high $r_{sm,final}$ values were more prone to early stopping and therefore ended after fewer steps, while low values resulted in an agent that chose the "stop" action much less frequently. When $r_{sm,final}$ was at or near zero, the agents would take over four times as many steps before finally stopping in the same area as agents for which $r_{sm,final} \geq 0.3$.

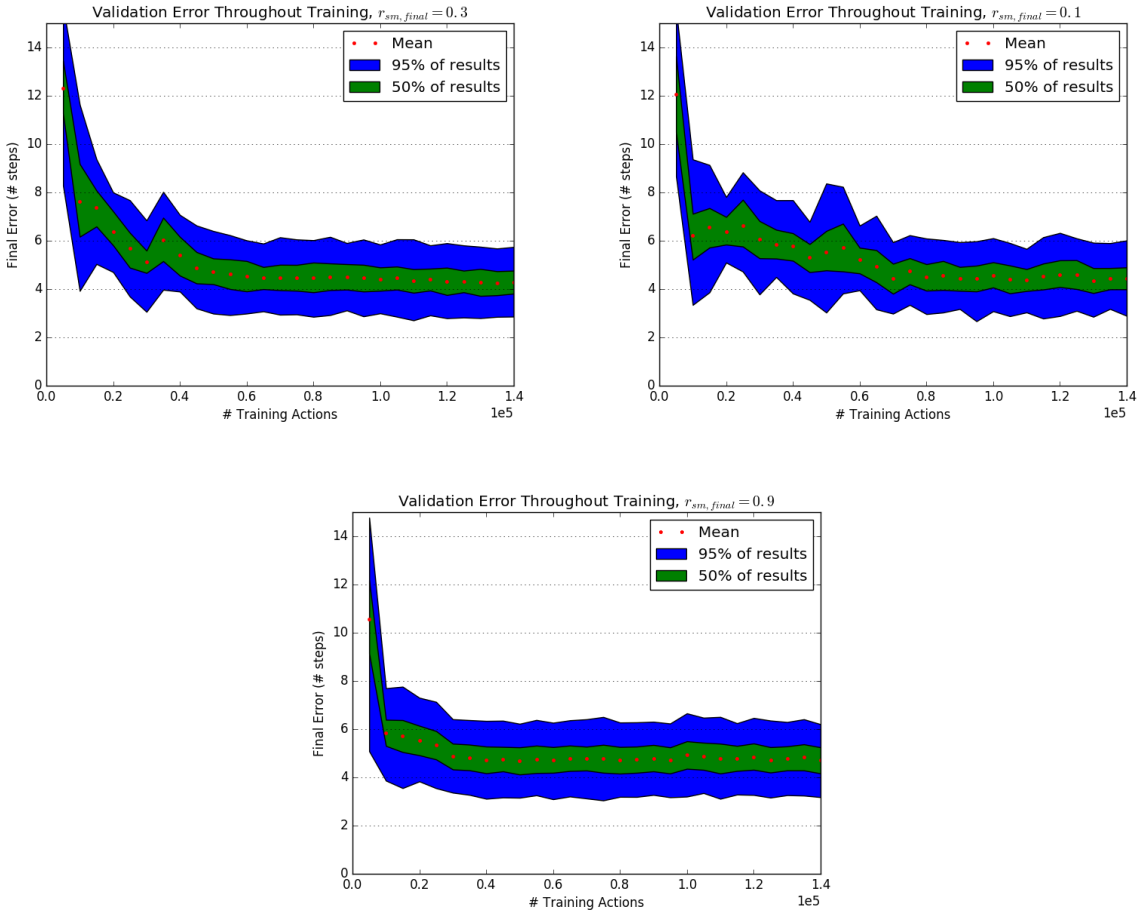

Figure B.8 Validation error logs for $r_{sm,final}$ tests. The best performance was observed with $r_{sm,final} = 0.3$ (upper left). Lower values ($r_{sm,final} = 0.1$, upper right) demonstrated greater fluctuation during training and slightly worse final performance, while higher values ($r_{sm,final} = 0.9$, bottom) demonstrated faster early learning but worse final performance. Axes and legend definitions are the same as in Figure B.1.

B.10 "Stop" Reward Ramp Length

During training r_{sm} mirrors ϵ, starting at $r_{sm} = 0$ and rising to $r_{sm,final}$ over the course of $N_{r,stop}$ training steps. This is to encourage early exploration by avoiding high "stop" rewards at a time when the under-trained agents do not have a very high expected value for non-"stop" actions.

$N_{r,stop}$ (# steps)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
$2.8 \cdot 10^4$	4.40	6.01	2.96	9.47
$5.6 \cdot 10^4$	4.43	5.99	2.87	9.56
$8.4 \cdot 10^4$	4.44	5.99	2.86	9.81
$1.12 \cdot 10^5$	4.34	5.81	2.90	9.19
$1.4 \cdot 10^5$	4.43	5.89	2.97	9.51

Table B.11 Effect of $N_{r,stop}$ on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.10.1 Effect on Registration Error

Varying $N_{r,stop}$ was not observed to affect final registration accuracy. However, lower values were found to approach this final performance after fewer training steps and to fluctuate less after reaching it, as seen in Figure B.9. Given the behaviour observed with $r_{sm,final}$, it seems likely that a significant contributing factor to this behaviour is the lower effective r_{sm} during the turbulent portion of the training history. There does not seem to be a noteworthy difference between slowly increasing r_{sm} over many steps or simply having a lower constant r_{sm}.

B.10.2 Effect on Training and Execution Speed

Agent performance was observed to plateau after fewer steps when trained with low $N_{r,stop}$ values, allowing equivalent accuracy with shorter training times when compared to higher $N_{r,stop}$.

As with other reward hyperparameters, $N_{r,stop}$ is not used during execution. No significant variation was observed in the number of steps taken by agents trained with different $N_{r,stop}$ values, and as such it does not affect execution times.

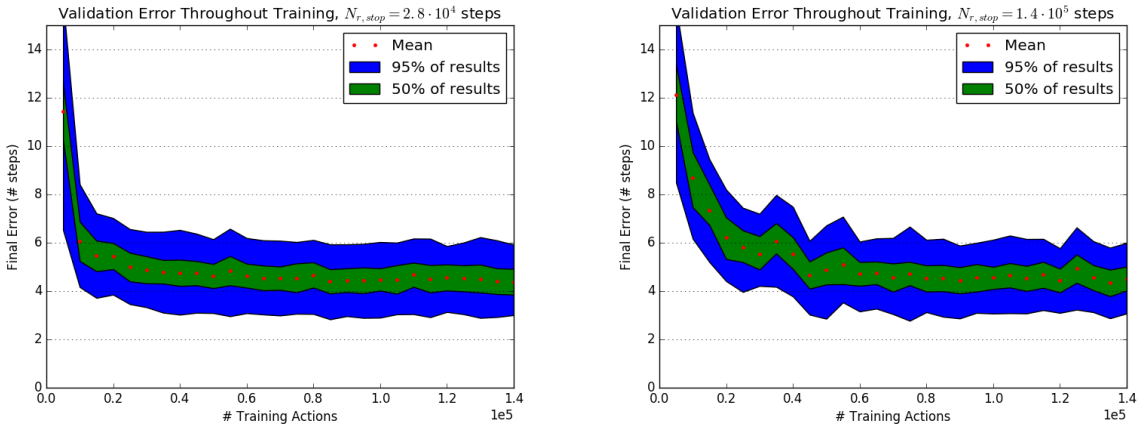

Figure B.9 Validation error logs for $N_{r,stop}$ tests. Lower values ($N_{r,stop} = 2.8 \cdot 10^4$ steps, left) show much faster early learning than higher values ($N_{r,stop} = 1.4 \cdot 10^5$ steps, right). Axes and legend definitions are the same as in Figure B.1.

B.11 "Stop" Reward Radius

Giving a reward for "stop" actions helps avoid infinite runtime by giving the agent incentive to stop after correct local registration, and giving a diminished but non-zero reward for imperfect registration extends this to lower-accuracy systems. The hyperparameter Rad_{any} determines the error above which "stop" gives zero reward, with a linear drop between $r = r_{sm}$ at zero error and $r = 0$ at Rad_{any} error.

Low Rad_{any} values may cause erratic performance and longer runtime, as the agent is slow to learn (or does not learn) when to stop. However, if Rad_{any} is too high it may negatively affect accuracy by decreasing the stop reward difference between adjacent locations, thereby lowering the relative penalty for stopping in an inferior location.

Rad_{any} (pixels)	Mean Error	95% Confidence Upper Bound	95% Confidence Lower Bound	Mean # Steps per Registration
2	4.29	5.74	2.88	21.82
4	4.40	5.87	2.89	13.16
6	4.27	5.71	2.86	8.73
8	4.46	5.96	3.00	7.94
10	4.53	6.13	3.03	7.96
12	4.42	5.95	2.87	7.58

Table B.12 Effect of Rad_{any} on validation error. All table values taken from the registration attempts made during the final validation phase, performed after training was complete. "Mean Error", "95% Confidence Upper/Lower Bound", and "Mean # Steps per Registration" columns have the same definitions as in Table B.2.

B.11.1 Effect on Registration Error

The lowest error was observed with $Rad_{any} = 6$, but when trained for sufficient time the final accuracy was found to be relatively insensitive to this hyperparameter. Accuracy throughout the training process showed similar behaviour to the examination of $r_{sm,final}$, with low values of the hyperparameter leading to slower improvement. This may be because increasing Rad_{any} causes greater "stop" rewards for low but non-zero error end positions, mimicking an increase of $r_{sm,final}$, rather than for its effect on stopping at greater distances.

B.11.2 Effect on Training and Execution Speed

Low values of Rad_{any} increase both training and execution time. The per-step training time does not change, but the time required to reach a given accuracy increases due to the slower rate of improvement as shown in Figure B.10.

Execution time is slower due to the increased number of steps taken before ending the registration, as selecting "stop" has a lower incentive.

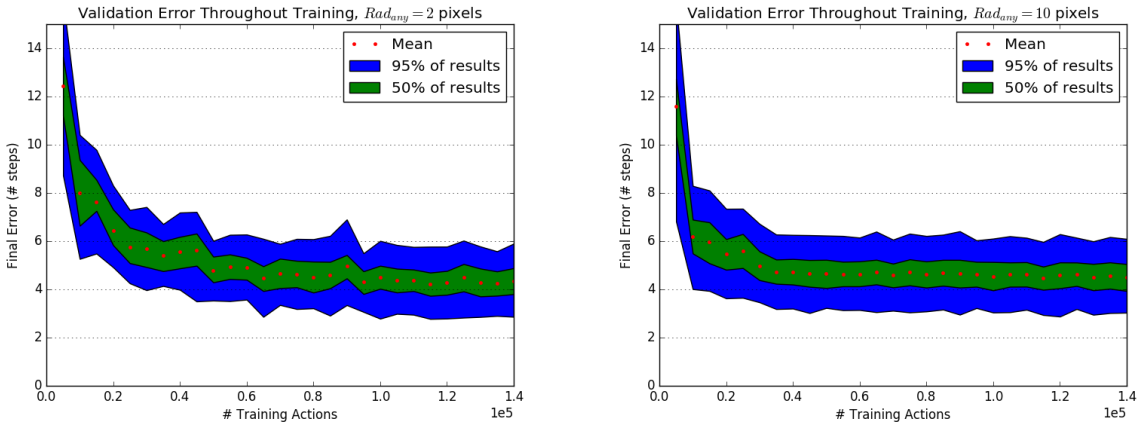

Figure B.10 Validation error logs for Rad_{any} tests. Lower values ($Rad_{any} =$ 2 pixels, left) showed slower improvement in accuracy, while higher values ($Rad_{any} = 10$ pixels, right) were observed to cause faster decrease in error during early training followed by nearly static accuracy throughout the rest of training. Axes and legend definitions are the same as in Figure B.1.