Deep Learning for Plasma Tomography

Francisco Duarte Pinto de Almeida Matos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Diogo Manuel Ribeiro Ferreira
Dr. Pedro Jorge de Paula Carvalho

Examination Committee
Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. Diogo Manuel Ribeiro Ferreira
Member of the Committee: Prof. Andreas Miroslaus Wichert

June 2016
Acknowledgments

First and foremost, I would like to express my utmost gratitude to my supervisor, Prof. Diogo R. Ferreira, for his steadfast dedication and support, for showing me how rewarding research work can be, and for the very positive experience I had in working with him while carrying out this thesis. I also want to thank him for introducing me to two fields of science – deep learning and fusion – which I only vaguely knew existed, and have now grown to admire for their potential to change our world.

I would also like to thank Dr. Pedro J. Carvalho, my co-supervisor, for taking the time to explain some of the intricacies of fusion diagnostics and for his assistance throughout this work.

Lastly, I would like to thank my parents for having always built up my sense of curiosity, for having always indulged my thirst for knowledge, and for having always supported my decisions. I have no doubt that that is what made me decide to study science and, ultimately, carry out this thesis.
Resumo

A tomografia é um diagnóstico frequentemente utilizado em experiências de fusão, nomeadamente para determinar a forma e posição do plasma. No entanto, os algoritmos tomográficos para plasmas trabalham com apenas algumas projeções, o que limita a sua precisão e aumenta a sua complexidade computacional. As redes neuronais têm sido apontadas como uma alternativa viável para concretizar a tomografia de plasmas, dado que uma rede neuronal treinada é potencialmente mais rápida do que algoritmos tradicionais, e é capaz de alcançar uma precisão semelhante. Esta tese propõe levar a cabo reconstruções tomográficas utilizando uma rede neuronal profunda, com uma topologia inversa à das redes convolucionais que têm sido foco de investigação nos últimos anos. Aplicando esta abordagem a dados de um diagnóstico de tomografia do JET (Joint European Torus), mostra-se que a rede é capaz de reproduzir as reconstruções com elevada fiabilidade, quando medida com várias métricas diferentes.

Palavras-chave: Diagnósticos de Plasmas, Tomografia, Redes Neuronais
Abstract

Tomography is a frequently used diagnostic in fusion experiments, namely to determine the shape and position of the plasma. However, plasma tomography algorithms work with only a few projections, which limits their accuracy and increases their computational complexity. Neural networks have been pointed out as a suitable alternative to carry out plasma tomography, since a trained neural network is potentially faster than traditional algorithms and capable of achieving similar accuracy. This thesis proposes to carry out tomographic reconstructions by using a deep neural network, with a topology that is the inverse of the convolutional networks that have been the subject of much research in the past few years. Applying this approach on data from a tomography diagnostic at the Joint European Torus (JET), we show that the network is able to reproduce the reconstructions with high accuracy, as measured by several different metrics.

Keywords: Plasma Diagnostics, Computed Tomography, Neural Networks, Deep Learning
Contents

Acknowledgments ......................................................... iii
Resumo ................................................................. v
Abstract ............................................................... vii

1 Introduction .......................................................... 1
  1.1 Motivation .......................................................... 1
  1.2 Main goal ........................................................... 2
  1.3 Thesis outline ....................................................... 3

2 Deep Learning .......................................................... 5
  2.1 Neural Networks ..................................................... 5
    2.1.1 Training, loss and learning rate ............................... 6
    2.1.2 Overfitting and regularization .................................. 9
  2.2 Problems with early networks ..................................... 10
  2.3 Towards Deep Neural Networks .................................... 11
  2.4 Convolutional Networks ............................................ 12
    2.4.1 Features and filters ........................................... 13
    2.4.2 Pooling .......................................................... 14
  2.5 Conclusion .......................................................... 16

3 Plasma Tomography .................................................... 17
  3.1 Principles of tomography .......................................... 17
  3.2 Applications in fusion diagnostics .............................. 19
    3.2.1 Traditional methods .......................................... 19
    3.2.2 Reconstruction using neural networks ......................... 20
    3.2.3 Shape and density of a fusion plasma ......................... 21
  3.3 A simple experiment ................................................ 23
    3.3.1 Network topology .............................................. 25
    3.3.2 Deep learning framework ..................................... 25
    3.3.3 Training parameters .......................................... 26
    3.3.4 Results ........................................................ 26
  3.4 Conclusion .......................................................... 29
4 Experiments on JET Data

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The KB5 diagnostic</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Data preprocessing</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Filtering by campaigns</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Principal Components Analysis</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>The Up-convolutional Network</td>
<td>36</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Symmetry of convolution and deconvolution</td>
<td>37</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Unpooling</td>
<td>37</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Topology</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Training and results</td>
<td>40</td>
</tr>
</tbody>
</table>

5 Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Contributions</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>46</td>
</tr>
</tbody>
</table>

Bibliography
Chapter 1

Introduction

Research has been ongoing for several decades to develop nuclear fusion as a viable and clean source of energy. Achieving self-sustaining fusion reactions requires the heating of gas until it becomes a plasma. This process is usually done inside devices such as tokamaks and stellarators [7].

Computed Tomography [26] has many practical applications, from medical imaging [27] to plasma diagnostics [12]. In nuclear fusion devices [51], tomography can be used to determine the shape and position of the plasma inside the device [1]. This can be used either for control of the fusion experiment, or for post-experimental analysis of the plasma [7].

It has been suggested that neural networks [17] can achieve tomographic reconstruction with the same or even higher accuracy [2, 44] than traditional tomographic algorithms. Neural networks are a model for machine learning which has seen tremendous progress in the past few years, in particular with the advent of deep learning [5]. One of the main successes of deep learning is in achieving (and even surpassing) human-level accuracy on image classification [18].

1.1 Motivation

The extremely high temperatures (in the order of millions of degrees) required for the occurrence of nuclear fusion mean that any contact between plasma and its containing vessel must be avoided, so as to prevent damage to the container walls. It is this that makes it necessary to have real-time knowledge, and control, of plasma position and shape. To know these properties, tomography is a useful diagnostic because it relies only on emitted radiation by the plasma; thus, it is, in principle, immune to influences from outside the vessel.

In 2010, at the ISTTOK tokamak [7], tomographic reconstruction was achieved through the use of classical tomographic algorithms. However, an experiment was made with a simple neural network, trained with a small sample of phantom plasma profiles. The phantoms simulated the actual expected plasma profiles from the ISTTOK tokamak and Wendelstein 7-X stellarator. The experiment showed that, despite its simplicity, the network achieved better reconstructions than traditional algorithms in all tested cases. This was in line with an experiment carried out at the Joint European Torus (JET) tokamak in 2009 [44], which had also achieved good results with tomographic

\[1\text{https://www.euro-fusion.org/jet/}\]
reconstruction based on neural networks.

1.2 Main goal

Tokamaks and stellarators have a toroidal shape. It is inside that toroid that hydrogen, tritium or deuterium gas are heated and turned into plasma.

In fusion plasmas, tomography is used to reconstruct a 2D cross section of the plasma inside its containing vessel. If the vessel is shaped like the one in figure 1.1, the reconstruction will be as if we took a slice of the toroid (a poloid) along the direction of one of the D-shaped blue hoops, and captured the plasma distribution in that plane.

The goal of this thesis is to investigate how Computed Tomography can be carried out using the latest developments in neural networks and deep learning, as opposed to traditional tomographic algorithms. In particular, we measured the accuracy that a deep neural network can achieve when reconstructing the cross section of a plasma.
contained in a tokamak. For this purpose, data from JET, which is currently the largest fusion experiment in Europe, was used.

1.3 Thesis outline

Chapter 2 introduces the principles behind traditional neural networks and their particular problems. It then presents the most recent solutions to those problems, which in the past few years brought about the field of deep learning. Within the specific context of deep learning, a particular type of deep network – the convolutional network – is also introduced.

Chapter 3 begins with a brief overview of traditional tomographic techniques and their main principles. It follows by describing the relevance of tomography when specifically applied to fusion plasmas, as well as the particular problems with this diagnostic. Then, previous experiments with tomographic reconstruction using neural networks are summarized. Finally, a simple experiment which attempts to use a neural network to reconstruct some phantom plasma emissivity profiles is explained. Some simple formulae are deduced to allow us to measure the accuracy of that experiment; our idea was to use this as a proof of concept before carrying out experiments on actual data.

Chapter 4 enumerates the experiments we did using data from JET. Firstly, we describe the diagnostic existing at JET that collects the data we use for tomographic reconstruction. Then, we explain the first steps – analysing and pre-processing – we took with the data. We then describe the topology of the deep neural network used for tomographic reconstruction, as well as the network training process. Finally, the results of using our approach are shown.

Chapter 5 sums up our conclusions and describes additional ideas which we believe may be of use to improve our approach.
Chapter 2

Deep Learning

The concept of machine learning has existed since the advent of modern computers and, in general, refers to creating computer models capable of recognizing, or learning, patterns in a given type of input data. Once a pattern has been learned, it can be used to make predictions regarding new inputs of the same type. Mathematically, recognizing a pattern essentially means finding a function that maps specific inputs to specific outputs as correctly as possible. Over the years, several models for machine learning have been proposed; we will focus on neural networks and, more specifically, deep neural networks.

2.1 Neural Networks

Neural networks [17] are inspired by the functioning of biological brains, where each individual neuron can be active or inactive and performs a simple task; it is the composition of many millions of neurons, and the links between them, that makes the brain capable of performing complex tasks.

Similarly, an artificial neural network has many neurons (or nodes). Each node receives an input, performs a calculation on it, and outputs a result, and has several incoming and outgoing links that connect it to other nodes.

In terms of topology, neural networks are a composition of layers of nodes. Typically, there is one input layer, one output layer, and at least one “hidden” layer in between. In the input layer, each node simply receives an element of input data which is to be processed by the network. In the hidden and output layers, each node receives the outputs from nodes in the previous layer.

The output $o_i$ of a node $i$ is computed by an activation function $f$ that is applied over the weighted sum of the components of $x$, the input vector of that node (the set of incoming values that were yielded by the previous layer). Activation functions must be non-linear to allow a network to capture non-linear dependencies between its inputs and outputs.

Since a node $i$ computes the weighted sum of its inputs (a quantity generally denoted as $net_i$), those weights can be tuned to influence the output of the node. A bias $b_i$ can also be used to displace the result of that weighted sum by a certain amount. The general rule for the output of a node $i$ preceded (pointed to) by $h$ nodes is given by:

$$o_i = f \left( \sum_{h} x_h w_{h,i} + b_i \right) = f \left( net_i + b_i \right).$$  \hspace{1cm} (2.1)
where \( x_h \) is the component of input vector \( x \) that comes from node \( h \) (i.e., the output of that node) and \( w_{h,i} \) is the weight of the link originating in node \( h \) and pointing to \( i \).

Because they influence node (and consequently, network) behavior, weights and biases are usually called the network parameters. When a network is first created, its parameters are usually initialized with random values; those values can be drawn from distributions with some specific mean and variance [15].

Figure 2.1: Sample of 2 network layers with a total of 4 nodes. \( w_1, w_2 \) and \( w_3 \) are the weights of the links between nodes A, B and C (in the first layer) and node D (in the second layer). \( b_D \) is D’s bias. \( o_D \) is (as per equation (2.1)) the result of applying D’s activation function \( f \) on its net input and bias, which is \( x_A w_1 + x_B w_2 + x_C w_3 + b_D \).

Although each node, by itself, merely calculates a weighted sum, the mathematical function rendered by the composition of many nodes with non-linear activations creates a powerful model that can describe highly complicated patterns. This power stems from the universality theorem [5], which states that one single hidden layer of nodes is enough for a network to be able to approximate any arbitrarily complex continuous function. Thus, when it comes to machine learning, no matter how complicated a pattern may be, a neural network with one layer is, at least in theory, capable of learning it [20].

2.1.1 Training, loss and learning rate

To learn a pattern or function, a neural network must be trained, that is, it must be tuned so that it maps inputs to outputs as accurately as possible. One common method for training is supervised learning, which requires a set of training samples in which the actual, correct output is known. This method consists in repeatedly feeding the network with those samples (either individually or in batches) so as to progressively reduce the error of its outputs.

The error of a network output is calculated by what is usually referred to as the loss function. Several types of loss functions exist – namely, mean squared and absolute error. Though their actual value may be different, their purpose is the same: to obtain a measure of how accurate a network output is when compared to the correct value.

Since the behavior of a network is determined by its node biases and weights, the error of its output depends on the values of those parameters. Thus, the immediate goal of the training process is to find a set of network parameters \( \theta \) (weights and biases) that minimize the loss function \( E(\theta) \) with respect to the data \( x \) contained in the training set \( X \). In particular, if \( E(\theta) \) is calculated through the mean squared error, then we wish to minimize

\[
E(\theta) = \frac{1}{n} \sum_{x \in X} (y(x) - f(\theta, x))^2 \tag{2.2}
\]

where \( n \) is the size of the training set, \( y(x) \) is the true (known) output corresponding to a training sample \( x \), and \( f(\theta, x) \) is the output computed by the network for that same sample.
Looking at equation (2.2), one can see that reducing network error – that is, finding a minimum for its loss function – requires tuning the values of \( \theta \), i.e., the network parameters. Different methods exist to find those values: namely, gradient and non-gradient based [29].

In gradient based methods, which are the ones we used in this thesis, the training process searches for the ideal parameter values by iteratively updating the parameters themselves; that is, the training process repeatedly performs the update

\[
\theta_{t+1} = \theta_t + \Delta \theta
\]

in search of values for \( \theta \) that minimize network error. In the example illustrated in figure 2.2, the network has only one parameter \( x \), thus the problem is only about finding the value of \( x \) that minimizes the error function \( f(x) \). This is done by iteratively searching for values of \( x \) that minimize \( f \) – a process illustrated by the green arrows. At each step, the slope of the tangent (red) line to the blue function is calculated, with respect to the current value of \( x \). If the slope is positive, the next step must move to the left; if the slope is negative, it must move to the right. Eventually, the algorithm should converge to the function minimum.

\[ f(x) \]

\[ x \]

Figure 2.2: Simple illustration of the error minimizing process in a neural network with only one parameter.

Naturally, the key to making this process work is to, in each iteration, find a \( \Delta \theta \) that indeed makes \( \theta \) approach its optimal value. While in the previous example this was done by calculating the slope of the tangent at each step – in effect, the derivative of the blue function – the more general rule is to compute \( \Delta \theta \) as

\[
\Delta \theta = -\eta \frac{\partial E(\theta)}{\partial \theta}
\]

i.e., the product between the gradient of the loss function with respect to the network parameters, and the network learning rate \( \eta \), which scales the size of each update. Intuitively, a learning rate controls how fast a network learns. The learning rate can be chosen and its ideal value is sometimes difficult to determine; too small a value means the network will take very long to find error minima, while too large a value will actually make the loss function become unable to converge. A good guiding principle is to choose a learning rate close to the largest value which does not cause the loss function to diverge [4].

Ultimately, this process uses the gradient of the loss function to attempt to minimize its value; for this reason, it is known as gradient descent. One particular variation of gradient descent is to calculate the gradient (and hence update the parameters) not on the loss function of the entire training set (as defined in equation (2.2)), but only on a batch of training samples, or even on a per-sample basis. In this case, the process is known as stochastic gradient descent (SGD). SGD is useful because it allows for more frequent parameter updates (thus speeding up the training
process) without significant loss of accuracy in the computed gradients [32].

A pass over all the samples in the training set – that is, using all training samples to compute parameter updates – is known as a training epoch. Usually, whether through simple or stochastic gradient descent, a full training process requires several epochs, i.e. several passes over the training set.

![Figure 2.4: Simple network with three nodes, each in its own layer.](image)

The actual calculation of the loss function gradient as defined in equation (2.4) must be done on a per-parameter basis – that is, for each weight and bias. In the case of the network depicted in figure 2.4, weight $w_{i,j}$ links a hidden layer node ($i$) to an output layer node ($j$). Thus, by applying the chain rule, the gradient of the error with respect to that particular link becomes

$$
\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}}
$$

(2.5)

where, as in equation (2.1), $o_j$ denotes the output of $j$, computed as the activation of its total input $net_j$ (which, in this case, is merely the input that it receives from $i$).

The component $\frac{\partial o_j}{\partial net_j}$ is actually the derivative of the activation function $f$ of node $j$ with respect to its input. This is important because it means that activation functions must be differentiable. In early neural networks, popular options for activation functions were sigmoid functions and the hyperbolic tangent. They have very simple derivatives, and have the added bonus of behaving similarly to the activations of the biological neurons on which neural networks are inspired.

In particular, if a sigmoid such as the logistic function $f(x) = \frac{1}{1+e^{-x}}$ is used for the activation, then the previous equation (still assuming a MSE loss function), for the node $j$ in figure 2.1, becomes

$$
\frac{\partial E}{\partial w_{i,j}} = (y_j - o_j) o_j (1 - o_j) o_i = \delta_j o_i.
$$

(2.6)

where $y_j$ is the actual (known) target value that $j$ should output, and $o_j$ and $o_i$ are the outputs of $j$ and $i$. $\delta_j$ is a quantity called the error of node $j$.

Since the loss function can only be calculated with respect to the output nodes – that is, $y_j$ is only defined if $j$ is an output node – calculating the error gradient with respect to a weight linking two hidden layer nodes is trickier.
It can still be done, however, by backpropagating the errors in higher layers back to the lower ones. In this case, because node $i$ is followed by node $j$, to calculate the error gradient with respect to weight $w_{h,i}$, we would have to backpropagate the error of node $j$ – that is, $\delta_j$, which we already know from equation (2.6). Thus, the error gradient with respect to the weight $w_{h,i}$ in figure 2.4 would be

$$\frac{\partial E}{\partial w_{h,i}} = \delta_i o_h = \delta_j w_{i,j} o_i (1 - o_i) o_h.$$  (2.7)

This means that the error in layers closer to the input depends on the error of the layers that come after them. If other nodes existed before $h$, that is, in previous layers, this expression would still be valid; at each layer, the gradient would always depend on the one after it, and so forth, recursively, until the output layer. As we will see later, this became one of the main problems of early neural networks.

Once the error gradient has been computed for all weights and biases in the network, the parameters for each node are updated accordingly. Ideally, with each new training iteration, the error should decrease, which corresponds to the network getting better at recognizing patterns – and at predicting results.

### 2.1.2 Overfitting and regularization

Given sufficient training samples, a well-trained network should be capable of generalizing from its training set so as to be able to make correct predictions on previously unseen input data [5] – that is, data which was not used for training the network. In practice, this does not always happen – a problem known as overfitting, that occurs when weights and biases have values that make the network output very correct results for its training set but, equivalently, very incorrect results for new data of the same type.

A set of strategies collectively known as regularization [5] attempt to prevent the occurrence of overfitting. One example of regularization is to limit the extent to which the network parameters are allowed to vary with each training iteration, so that they do not become too adjusted to the training set. This is done by introducing a new term in the loss function of equation (2.2), which becomes

$$E(\theta) = \frac{1}{n} \sum_{x \in X} (y(x) - f(\theta, x))^2 + \lambda R(\theta)$$  (2.8)

where $R$ is the regularization function applied on the network parameters. Several regularization functions exist, such as the L1 or L2 (Euclidean) norm of $\theta$; each yields potentially different results. The parameter $\lambda$ controls the strength of the regularization factor. If it is too low, the regularization may be too weak to prevent overfitting. If it is too large, the network will actually begin to underfit – that is, it will be unable to make its parameters converge to their ideal values. Underfitting can also arise when the network is too simple to fit a function with a particularly complicated pattern.

Another form of regularization is to set aside some training samples for the purpose of validation – that is, create a validation set. Validation prevents overfitting by checking that, as the loss function over the training set decreases with more training iterations, a similar reduction is occurring in the loss function over the validation set. The key difference here is that the validation set samples are not used to compute network parameter updates; thus, they can be used to measure how well the network is generalizing to new data while training progresses.
Figure 2.5: Examples of overfitting (a), underfitting (b) and the real function to approximate (c). In (a), the network parameters are too well adjusted to the training set. In (b), the network parameters cannot adjust themselves to discover the desired function. In both cases, any new inputs will be mapped to very wrong outputs. In (c), the parameters are tuned well enough that a pattern is found, but not too adjusted to the training set. Thus, while new inputs may be mapped to outputs with some error, it will generally be relatively small.

When the loss function on the validation set does not improve anymore, training can be stopped – hence, this strategy is known as early stopping. In practice, it is common to use e.g. 20% of the available training data for validation. In cases where a validation set is used, one should look to the validation loss, rather than training loss, for a measure of network accuracy while training is carried out.

After training ends, it is common to use an additional set of samples – the test set – to assess the predictive power of the network. The difference between this and the validation set is that the latter is used to stop training at a specific time, and therefore it ultimately has some influence (even if indirectly) on the rendered network. On the other hand, the test set is not used either for parameter updates, nor for early stopping, and thus it can provide a final, objective measurement of network accuracy without ever having been responsible for network optimization.

2.2 Problems with early networks

According to the universality theorem, a neural network with a single hidden layer is capable of approximating any continuous function. However, attempting to build a network with only one layer to approximate complex functions often requires a very large number of nodes and is therefore computationally prohibitive.

The immediate solution to this is to build networks with more hidden layers. Indeed, deep networks with more layers, even with less nodes in total, are capable of achieving the same, if not better, results than shallow networks [5], at a comparatively smaller computational cost. This solution, however, introduced a problem in neural network research – that of vanishing gradients [29].

When training a network, computing the updates for network parameters requires obtaining the error gradient at each node, which, as we saw in Subsection 2.1.1, depends on the derivative of the activation function at each node in the network. We also saw that early networks used activations like sigmoids and hyperbolic tangents. The problem is that far away from the origin (i.e., far from \( x = 0 \)), these functions converge to 1, and their derivatives correspondingly converge to zero.

Consequently, when backpropagation happens with these activation functions, a situation might occur where the error gradient reaches a layer with nodes in saturation (i.e., with outputs close to 1 and derivatives close to 0). If that happens, the gradient is backpropagated to the next, lower-level layer as a very small value, and approaches zero as it reaches the input layer. As a result, weight and bias updates in lower-level layers are almost insignificant. Ultimately, this phenomenon means that it is impractical to train networks with many layers when using sigmoid
or hyperbolic tangent activations.

The vanishing gradients problem remained unsolved for many years. An unrelated, yet additional problem with training neural networks was the need of having datasets large and diverse enough to achieve a good generalization capacity (i.e., to prevent overfitting).

In any case, neural networks have been successfully deployed and used since the 1990’s on many applications. However, for most purposes, the lack of computing power, the vanishing gradients problem, and the absence of adequate training sets led to disappointing results which stalled neural network research throughout the 90’s and into the beginning of this century. The field only gained renewed interest in the past few years.

2.3 Towards Deep Neural Networks

Computational power has been steadily increasing for decades. Recently, however, a new hardware paradigm has come about: that of GPU (Graphical Processing Unit) computing. GPUs excel at parallel computation, which refers to processing data in parallel rather than sequentially [42]. They are ideal for scientific problems which require applying the same operation on large datasets. This is exactly the sort of operations that are done on neural networks. Therefore, we are now capable of performing neural network calculations with much greater speed, allowing for greater numbers of neurons and thus, larger, more powerful, networks, with potentially better accuracy.

On the other hand, a solution to the vanishing gradients came with the introduction of the Rectified Linear Activation Function (ReLU) [16], which takes the form

\[ f(x) = \max(0, x). \]  

(2.9)

Unlike traditional sigmoid functions, the ReLU’s derivative takes only two values: 1, for positive inputs, and 0, for negative inputs. Thus, gradients are no longer exponentially decreased after they pass through a node in saturation.

ReLUs have the added advantage of tending to induce sparsity in networks; sparse networks are much less entangled than dense ones. This means that small variations in the inputs do not generate potentially large changes in most of the network weights and biases, thereby facilitating parameter convergence to correct values.

Finally, on the problem of training datasets, one new factor has come about in the recent past: the explosion of
the amount of digital data, and the ease with which it is now shared through the Internet. Whether it be watching movies, reading books, listening to music, or purchasing items online, everything we do is more and more digitized. The fact that even greater amounts of data are expected in the future, as a result of the miniaturization of sensors and their application on our everyday devices, means that this trend is set to continue [43]. From a neural network point of view, this information bonanza is ideal for training models with greater capacity for generalization, and therefore, achieving better results.

On the whole, these solutions spurred renewed interest in neural network research in the past few years. For the first time, it has became possible to efficiently train deep networks with many nodes and layers. It is this fact that has brought about the term deep learning.

### 2.4 Convolutional Networks

Until now, we have described fully connected (FC) neural networks, i.e., networks where all nodes in a layer are connected to the previous one. This type of network topology has a drawback: when network input size increases even by a small amount, the number of total network connections increases greatly, quickly rendering them cumbersome even for the fastest computers. This is particularly problematic, for instance, in image recognition tasks, because images are two dimensional and thus their total number of pixels increases very easily.

Convolutional Neural Networks (CNN’s) are a particular type of deep networks whose architecture is inspired by the functioning of the animal visual cortex. They are typically used for image classification tasks, in which they have had considerable success[28]. Crucially, unlike fully connected networks, their number of links does not increase exponentially with the size of the inputs; therefore, these networks scale easily to images of larger sizes. CNN’s are usually composed of alternating convolutional and pooling layers, with several fully connected layers at the end. The use of fully connected layers may seem to defeat the purpose of a CNN (because it apparently brings back the problem of increasing network size). However, the convolutional and pooling layers are designed in such a way as to make the FC layers in a CNN be much smaller than the network inputs, and hence, the problem does not arise.
2.4.1 Features and filters

When used for image classification, the idea behind convolutional networks is that images are composed of several distinguishing features. While the first level layer of a fully connected network would simply combine all pixels of an image, a convolutional layer in a CNN will instead have several feature maps, each one specialized in detecting a given feature in the input. A feature map (also called a filter or kernel) is essentially a group of weights shared by several nodes. It is this parameter sharing that prevents the number of network links from bursting like in fully connected networks [30].

Filters can be much smaller than the images entering the network; typical feature map sizes are $3 \times 3$ or $5 \times 5$ weights. Each filter slides horizontally and vertically across all the input, mapping (or more accurately, convolving – hence the name convolutional neural network) certain regions of the input into a particular node of the following layer. As is the case with fully connected networks, the weights and biases of convolutional layers in CNN's must be tuned through a training process. Intuitively, when a CNN trains, the parameters of its convolutional layers will adjust themselves to detect those features that the network “judges” to be the ones most important to output a correct result.

Each convolutional layer in a CNN can be thought of as having a 3D spatial arrangement. In particular, a convolutional layer at the beginning of an image processing network will have an input volume of dimensions $(im\_width, im\_height, im\_channels)$, corresponding to the width, height and number of channels of the input, and an output volume of $(new\_width, new\_height, number\_of\_filters)$, corresponding to the output width and height, and to the number of filters that processed the input. If no zeros are padded to the outermost columns and rows of an input, output width and height will be smaller, because filters map regions to single nodes. This may be clearer in

Figure 2.8: Fully connected (left) vs convolutional layer (right). Notice how, in the FC layer, if the input size increases by even a small amount, the number of links between the nodes will increase much more. In the convolutional layer, however, no matter how large the input, the feature maps will always have the same size, so the number of links does not increase. In this convolutional layer, 2 feature maps, with a width and length of $3 \times 3$, are convolving an input of depth 1; thus, the output will have a depth of 2.
Figure 2.9 illustrates the behavior of a convolutional layer with an input whose dimensions are $5 \times 5 \times 1$ – for example, a $5 \times 5$ image with only one channel. Since the input depth is 1, filters $w_1$ and $w_2$ also have a depth of 1; their width and length are 3. Furthermore, since there are two filters, the output will have a depth of 2 (represented by matrices $o_1$ and $o_2$ on the right). Output width and height are $4 \times 4$ because $w_1$ and $w_2$ can only be applied a maximum of 4 times horizontally and vertically on the input. In this case, preserving input width and height would require padding it with 2 rows and 2 columns of zeros.

As in fully connected layers, the output of a node is given by equation (2.1). In this case, $o_{1,1}$ will be equal to applying that equation on all nodes of the input convolved by $w_1$ - that is, the area covered by the large blue square - which renders

$$o_{1,1} = f(x_{1,1}w_{1,1} + x_{1,2}w_{1,2} + \ldots + x_{3,3}w_{1,3} + b_1)$$

where $f$ is the applied activation function (in this case, let us suppose, a Rectified Linear Unit - recall equation (2.9)). Thus,

$$o_{1,1} = \max(0, (x_{2,3}w_{1,2,3} + x_{3,2}w_{1,3,2} + b_1)) = \max(0, (-2 - 1 + 1)) = \max(0, -2) = 0.$$  

which is the value in the small blue square in figure 2.9. Similarly, $o_{2,3}$ is the result of $w_2$ convolving the region of the input surrounded by the large cyan square, followed by adding the bias and computing the activation.

This example illustrates the behavior of a single convolutional layer with two filters; a convolutional network, as a whole, would have several of these layers, each with potentially different numbers of filters.

### 2.4.2 Pooling

Another key idea behind the architecture of CNN’s is that when an image is to be classified, the relative position of its features is irrelevant compared to what features are actually present. This sort of translational invariance
is usually obtained through the use of a sub-sampling or pooling layer. This type of layer essentially performs a reduction in the size of its input which eliminates the information on the spatial location of a feature detected by a previous convolutional layer.

Pooling is usually done with the \textit{max} mathematical function. In this case, the operation done by the layer is usually called \textit{max pooling}. Remember that in a convolutional layer, a feature map will output a series of nodes with varying activations; some of those nodes, however – the ones corresponding to the convolved regions most similar to the feature being looked for – have the greatest activation values. The max pooling layer receives non-overlapping regions of the input and selects the maximum value of each of those regions, outputting only single, scalar results.

Typical max pooling operations act on $2 \times 2$ regions of an input; in that particular case, the max pooling operation will cut total input width and height by half. Pooling operations preserve input depth.

The end effect of the max pooling layer is the detection of some feature of the input, irrespective of its position,
as well as a reduction of the input size. In particular, this reduction in size is ideal to combat overfitting, as it decreases the number of free parameters to which the following network layers will have to adjust.

Ultimately, in a convolutional network, inner layers attempt to find small, localized image features, while layers closer to the output will identify higher-level features that describe the image content. After several successive convolutional and max pooling layers, total input size should shrink to a certain minimum, while most high-level features will have been extracted. At that point, one or more fully connected layers are used to combine those features and output a result, which may correspond, for example, to the probability that some image belongs to a certain class.

2.5 Conclusion

This chapter described fully connected and convolutional neural networks. CNN’s, in particular, are usually used for image classification and recognition tasks. The goal of this thesis, however, was not to feed a network with an image and classify it. Rather, our goal was the opposite: to feed a network with plasma data and get it to output the corresponding plasma profile. The next chapter begins by describing the tomographic problem and how it has been solved up to this point. It then describes a simple experiment carried out to evaluate the potential accuracy that neural networks can achieve with regards to tomographic reconstruction.
Chapter 3

Plasma Tomography

In general, tomography refers to obtaining (more accurately, reconstructing) a 2D cross section of an object without having to slice it open. Though commonly associated with the medical industry due to its application in non-invasive imaging of living bodies, tomography has a wide range of applications in other fields [8]. In this work, we were interested in its applications in plasma physics, and in particular, in using a variation of traditional tomography to reconstruct the 2D cross section of a plasma.

The basic mathematical formulae necessary for tomographic reconstruction of an object have existed since at least 1917, through the work of Johann Radon. In practice, effective tomographic reconstruction required computational capabilities that only came about with the advent of modern computers in the 1950’s and 60’s; for that reason, it is frequently designated as Computed Tomography (CT) [26].

3.1 Principles of tomography

The reconstruction of the 2D cross section of an object requires the acquisition of several of its projections. In medical imaging, acquiring a projection means illuminating the object with electromagnetic radiation (typically, X-rays) from a certain direction or angle $\theta$ in the cross section plane, and measuring the resulting radiation attenuation along the opposite side. Due to the density and shape of the object, the radiation, after traversing it, is reduced by different amounts at each point of measurement in projection space; the actual projection is the set composed of those measurements. Intuitively, rays that traverse denser or deeper areas will suffer more attenuation.

Mathematically, this process is equivalent to evaluating the line integral of the function $g(x, y)$ (that describes the point-wise absorbency of the 2D cross section of the medium) along each line of measurement $L(\theta, t)$:

$$ P_\theta(t) = \int_{L(\theta, t)} g(x, y) \, dL $$

(3.1)

where $L(\theta, t)$ is the line in the direction of $\theta$ that points to the position of $t$.

The cross section of the object can be reconstructed by combining multiple projections at different angles. In classical CT, and when projections with parallel rays are being used, the key principle behind reconstruction is the Fourier Slice Theorem [26]. The theorem states that the 1D Fourier Transform of each projection is a “slice”
Figure 3.1: Example of a projection, $P_\theta(t)$, measured along an angle $\theta$. The blue area $g(x, y)$ is the cross section that we wish to image, and is being traversed by radiation. Because different rays traverse different areas of the object, the value at each point $t$ in projection space will be different.

of the 2D Fourier Transform of the cross section of the object. With several projections along different angles and their respective 1D Fourier Transforms, it is possible to obtain several slices of the 2D transform of the cross section. Each slice will be rotated by an angle which corresponds to the angle of projection. Those slices are then combined, and an Inverse 2D Fourier Transform is applied on them, rendering the desired cross section – that is, the original function $g(x, y)$ that we saw earlier:

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} G(u, v) e^{i2\pi(ux+vy)} \, du \, dv.$$  \hspace{1cm} (3.2)

In this case, $G(u, v)$ denotes the 2D Fourier Transform of $g(x, y)$. Intuitively, this equation tells us that the more projections one has, the more accurate the reconstruction can be; in fact, given an infinite number of projections, it would be theoretically possible to obtain an exact 2D transform of the desired cross section and therefore to reconstruct it with complete accuracy.

In practice, obtaining an infinite number of projections is impossible; nevertheless, that condition implies that tomographic algorithms work ideally when fed with many projections. For that reason, in medical applications, CT relies on as many as $10^5$ measurements, in projections taken over angle spans of $360^\circ$, to obtain good reconstructions [7]. The computer implementation of this process is usually done with the filtered backprojection algorithm [26].

Naturally, it is essential to have a way to check the accuracy of tomographic algorithms. This is done by testing the algorithms on objects whose cross sections are known, and whose projections can be calculated analytically. By feeding the algorithm with those projections, it is then possible to compare the reconstructed image to the actual, known cross section of the object.

In medical applications, a commonly used benchmark for tomography algorithms is the Shepp-Logan phantom [9]. This phantom has a cross section that resembles a human head, but is in fact the superposition of 10 ellipses with different parameters. It is not by chance that ellipses are used; rather, their usage is useful because they are simple geometrical objects with known shapes, and whose projections can be calculated analytically.
3.2 Applications in fusion diagnostics

Tomography can also be used to determine the 2D cross section of the plasma contained inside a fusion device. In contrast with medical imaging, where a body is placed in front of a radiation source, the projections of the cross section of a plasma are measured as a function of the radiation emitted by the plasma itself. Thus, in this case, the projection value at each point in projection space is the total radiation emitted by the plasma along the corresponding line, rather than the total attenuation; essentially, equation (3.1) still holds, but in this case, \( g(x, y) \) corresponds to the point-wise plasma emissivity. In general, emitted radiation intensity depends on plasma density and temperature; therefore, it can convey information regarding plasma position [35].

Plasma emissivity can be detected by a bolometer system [21, 22]. A bolometer is a radiation measuring instrument based on temperature. Essentially, it has an absorber that captures the total radiation (the emissivity line integral) along a path, and a thermometer that measures the resulting temperature change in the absorber [25]; this variation in temperature is a marker of how much radiation was emitted. One or more bolometers can be grouped in a camera which, for tomographic purposes, acquires one projection. Multiple cameras can be used to capture different projections (that is, at different directions).

3.2.1 Traditional methods

Aside from the fact that it measures emitted (rather than absorbed) radiation, plasma tomography has an additional particularity: sparsity of projection data. The technical constraints inherent to fusion experiments (that is, poor accessibility of plasma-containing vessels) mean that obtaining a large number of projections over many different directions is impossible. Concurrently, the existing projections typically have few line integral measurements. As a result, tomography, when applied to fusion plasmas, is an ill-posed problem: typically, many solutions exist that satisfy existing projection data [36, 35]. For the same reason, methods used in medical tomography such as the filtered backprojection algorithm are unsuitable.

Consequently, plasma tomography must make use of alternative algorithms that can adapt to the small number of projections. However, the trade-off is that even small errors in projection measurements are enough to make those algorithms generate major artefacts – i.e., areas with wrong emissivity values – in the reconstructed plasma profiles [40].
To overcome this issue, plasma tomography algorithms work with the aid of regularization. Several types of regularization methods exist for plasma tomography, such as the Tikhonov regularization, maximum likelihood, and maximum entropy [10]; essentially, they refer to making a priori assumptions about the profile of the plasma to reconstruct, and thus restrict the number of solutions for the existing projection data. Examples of those assumptions include the smoothness of the plasma surface [24], its thinness (if the plasma is thin enough, it is safe to assume that all emitted radiation escapes it to reach the sensors) [7], and boundary conditions such as non-negativity of plasma emissivity, and zero emissivity outside the plasma edge [40, 35].

In any case, the inherent difficulty in the task of plasma tomography means that the algorithms traditionally used are computationally intensive and, consequently, relatively slow. For this reason, in fusion experiments, older algorithms (and computer hardware) were used essentially for post-experimental study of natural events such as sawteeth and disruptions, or induced phenomena such as impurity injections [24] – and only on the most relevant plasma discharges.

On the other hand, controlling plasma position is one of the main challenges in fusion research [41]. Plasmas in nuclear fusion devices are heated to extremely high temperatures; in the JET tokamak, for instance, it is heated to about 150 million degrees Celsius. These temperatures induce turbulence and instabilities in the plasma; should it touch the container walls, they would melt or evaporate.

Prevention of contact between plasma and the container walls is achieved through the use of externally generated magnetic fields that attempt to keep the plasma in its prescribed position. When instabilities occur, the external magnetic field must be adapted so that it can push the plasma back to its ideal position. However, they happen extremely fast – in the order of mili to microseconds [7]. Consequently, any control system must be able to react very quickly to any detected instability or shift in position. It is for this reason that tomography – with its high computational cost – has, for the most part, been difficult to use as a control mechanism. Nevertheless, its potential usefulness as a diagnostic for real-time study (and control) of fusion experiments has long been understood [2, 36].

### 3.2.2 Reconstruction using neural networks

Proposals have long been made to use neural networks as a tool for carrying out plasma tomography [11]. In 2000, Ma et al. [33] developed simple networks where the number of layers varied from 3 to 5 and the total number of nodes never surpassed 63. Curiously, for the output layer, they chose an activation \( f(x) = x + \log(1 + e^{-x}) \), whose shape is very similar to that of the softmax function – which is not very different from the ReLU – because it allowed them not to normalize the data. For the other layers they used the logistic function. They concluded that neural networks were useful for solving ill posed tomographic problems (i.e., where the number of projections is small) but the approach was hampered by computational cost. To mitigate this issue, they suggested the introduction of parallel computation.

In 2002, Barana et al [2] also used simple networks, with only one hidden layer, and used the hyperbolic tangent as activation function. They used 5 separate networks to reconstruct different parts of a plasma cross section, because some areas had different radiation characteristics. Their results showed that neural networks approximated the results of traditional algorithms satisfactorily and were fast enough to be a good candidate for real-time control of fusion experiments.

More recently still, in 2010, Ronchi et al. [44] also used a simple network with sigmoid activations for neu-
tron emissivity tomography (the KN3 diagnostic) at JET. They achieved conclusions similar to previous research, though they also observed that larger networks did not seem to yield better results. In hindsight, their use of sigmoid activations might have contributed to this issue.

Despite these experiments with neural networks, in 2010, research was still ongoing on improving traditional tomographic algorithms to make them accurate and fast enough for the purpose of real-time plasma study and control. This feat was indeed achieved at the ISTTOK tokamak, though algorithm effectiveness was restricted to some particular (small) types of fusion vessels, and to projections measured by reduced numbers of bolometers [7]. Other advances in the speed of traditional tomographic algorithms were also achieved at JET [40].

In any case, the surge of deep learning in the past few years means we now have much better tools and hardware to attempt to carry out plasma tomography using neural networks as opposed to classical tomographic techniques. In the previous attempts described above, the networks used were relatively small, as were the training sets. Furthermore, they were developed with code running on CPUs. In principle, one can expect to achieve faster and better results with more recent, optimized frameworks which generate GPU-executable code, and using larger and deeper networks, and larger training sets.

### 3.2.3 Shape and density of a fusion plasma

In section 3.2.1 we saw that, in medical applications, it is common to measure tomographic accuracy using objects composed of ellipses, because their projects can be analytically calculated. An ellipse of radii $a$ and $b$, centered at the point $(x_0, y_0)$, can be expressed as:

$$
\frac{(x - x_0)^2}{A^2} + \frac{(y - y_0)^2}{B^2} = 1.
$$

(3.3)

From here, we can deduce the expression that allows us to calculate the $y$-projection of an ellipse (i.e. vertical-wise projection towards the top), which we denote $P_y(x)$ because its point-wise value depends on the $x$ coordinate.

By solving equation (3.3) for $y$, one gets

$$
y = \pm b \sqrt{a^2 - \frac{(x - x_0)^2}{a^2}} + y_0 \Leftrightarrow y = b \sqrt{1 - \frac{(x - x_0)^2}{a^2}} + y_0 \vee y = -b \sqrt{1 - \frac{(x - x_0)^2}{a^2}} + y_0. \tag{3.4}
$$

These are the $y$-coordinates of the points belonging to the ellipse, as a function of $x$. Assuming that the $y$-projection is obtained through parallel rays, we can calculate it by evaluating the integral specified in equation (3.1). In this case, the emissivity function $g(x, y)$ to integrate is a constant $\rho$ (corresponding to some value of ellipse emissivity) while the line (path) to integrate is the path limited by the two values of $y$ that are found by solving equation (3.4), which we denote $y^+$ and $y^-$. Thus,

$$
P_y(x) = \int_{y^-}^{y^+} \rho \, dy = \rho \int_{y^-}^{y^+} \, dy = \rho (y^+ - y^-) = 2b\rho \sqrt{1 - \frac{(x - x_0)^2}{a^2}}. \tag{3.5}
$$

Naturally, this condition is only true if the $x$ values for which we calculate it are inside the ellipse; this is the same thing as saying that the expression is only valid for $\frac{(x-x_0)^2}{a^2} \leq 1$. Through an analogous process, the
projection (i.e. horizontal-wise projection towards the right) of the ellipse (for points whose \( y \) coordinate is contained in the ellipse) is

\[
P_x(y) = 2a\rho\sqrt{1 - \frac{(y - y_0)^2}{b^2}}. \tag{3.6}
\]

In short, the projections are

\[
P_y(x) = \begin{cases}
2b\rho\sqrt{1 - \frac{(x - x_0)^2}{a^2}} & \text{if } \frac{(x-x_0)^2}{a^2} \leq 1 \\
0 & \text{if } \frac{(x-x_0)^2}{a^2} > 1
\end{cases} \tag{3.7}
\]

\[
P_x(y) = \begin{cases}
2a\rho\sqrt{1 - \frac{(y - y_0)^2}{b^2}} & \text{if } \frac{(y-y_0)^2}{b^2} \leq 1 \\
0 & \text{if } \frac{(y-y_0)^2}{b^2} > 1
\end{cases} \tag{3.8}
\]

These two projections are illustrated in figure 3.3.

Interestingly, the shape of a plasma cross section inside a tokamak also resembles an ellipse. An important difference, however, is that its emissivity (and therefore its density) is far from being constant. Instead, it grows from a minimum at the edge (the outermost elliptical contour) to a maximum at the center. The actual shape and density (or, more precisely, pressure) of the plasma is described by the Grad-Shafranov equation [51], a well-known differential equation that describes the magneto-hydro-dynamic (MHD) equilibrium of a plasma. The simplest known solution to the Grad-Shafranov equation is Solovev's solution which involves a polynomial of fourth degree [46].

Therefore, rather than using a constant value, the plasma emissivity \( \rho \) can be better approximated by a fourth degree function such as:

\[
\rho(r) = \begin{cases}
    c(1 - 2r^2 + r^4) & \text{if } r \leq 1 \\
    0 & \text{if } r > 1
\end{cases} \tag{3.9}
\]

where \( c \) is an arbitrary constant corresponding to the maximum emissivity, and \( r \) is a measure of distance to the
center of the plasma, assuming it has an elliptical shape:

$$r = \sqrt{\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2}} \quad (3.10)$$

The function $\rho(r)$ has the property of being zero when $r = 1$, which yields the equation of an ellipse with radii $a$ and $b$, and has a maximum value of $c$ at the point $(x_0, y_0)$. Figure 3.4 shows two plots of this function.

For an ellipse with the emissivity function of equation (3.9), and through a process analogous to calculating the projections with constant emissivity, the $y$- and $x$-projections are given by

$$P_y(x) = \begin{cases} \frac{16}{15} a c \left( 1 - \frac{(x-x_0)^2}{a^2} \right)^{\frac{3}{2}} & \text{if} \quad \frac{(x-x_0)^2}{a^2} \leq 1 \\ 0 & \text{if} \quad \frac{(x-x_0)^2}{a^2} > 1 \end{cases} \quad (3.11)$$

and

$$P_x(y) = \begin{cases} \frac{16}{15} a c \left( 1 - \frac{(y-y_0)^2}{b^2} \right)^{\frac{3}{2}} & \text{if} \quad \frac{(y-y_0)^2}{b^2} \leq 1 \\ 0 & \text{if} \quad \frac{(y-y_0)^2}{b^2} > 1 \end{cases} \quad (3.12)$$

This result is interesting because it allows us to easily measure the accuracy that a neural network can achieve in reconstructing plasma cross sections, if we assume that they are elliptical in shape and have an emissivity approximated by equation (3.9). Thus, before applying neural networks on actual data, a preliminary experiment was carried out which consisted in generating elliptical phantoms and training a neural network to recover them from their $x$- and $y$-projections. We then evaluated the accuracy of that network in recovering previously unseen phantoms from their projections. The idea was to use this as a proof of concept before using neural networks on actual plasma data.

### 3.3 A simple experiment

The experiment presented in this section attempted to use a neural network to reconstruct phantom plasma profiles from their projections. For simplicity, we supposed that there were only two cameras, which corresponded to the top and side views of the plasma — i.e., they were positioned at the projection angles $\theta = 0$ and $\theta = 90^\circ$ — and that the projections were obtained through parallel rays.

To be able to use the formulae derived in section 3.2.3, we assumed that the plasma had an elliptical shape.
and was contained in a square box with normalized dimensions – that is, the coordinates of both the \( x \) and \( y \) axes were in the range \([0, 1]\). Using vertical- and horizontal-wise projections allowed us to calculate them using \( P_y(x) \) as defined in equation (3.11) and \( P_x(y) \) equation (3.12), respectively. Each projection had 11 equally spaced projection lines, at intervals of 0.1. Therefore, the number of inputs to the network was 22, corresponding to the total number of projection values.

The output of the network was the 2D cross section of the plasma with a certain resolution, where each output node gave the density/emissivity \( \rho \) of the plasma at a certain position \((x, y)\) in the normalized square box. To decide on the image resolution, we took into account the experiments carried out at the ISTTOK tokamak [7], where 3 projections, of 8 values each, were used to generate images on a \( 15 \times 15 \) grid – roughly, doubling the number of values in each dimension. Therefore, we opted for an output resolution of \( 21 \times 21 \), which is approximately the same proportion, and also allows us to have an evenly spaced grid at intervals of 0.05 in both dimensions. This resolution is also similar to the image sizes found in well-known datasets for deep learning, such as the MNIST database [31], where images are \( 28 \times 28 \) pixels. Thus, there will be a total of \( 21 \times 21 = 441 \) network outputs.

Figure 3.5 illustrates the resolution of the projections in each direction, and the resolution of the cross section which was to be reconstructed by the network.

![Figure 3.5](image_url)

Train and test datasets were generated with a Python script that produced samples based on randomizing the ellipse parameters \((x_0, y_0, a, b, c)\) subject to two restrictions: the ellipse must fit within the normalized square box, and the value of \( c \) was in the range \([0, 1]\). With these parameters we calculated, for each ellipse/phantom, the 22 values of the projections that would serve as input to the network, and the 441 values of emissivity that the network should produce as output. These are calculated based on equations (3.9), (3.10), (3.11) and (3.12). Since we had the actual density values for each example in the test set, we could measure the network accuracy in recovering the density values of those phantoms. By measuring this accuracy, we could get an estimate on the effectiveness of using deep networks for recovering actual plasma cross sections.
3.3.1 Network topology

The main inspiration for the network used in this experiment was the autoencoder [19], a type of deep network that is based on the idea that data dimensionality can be reduced (encoded) into a small set of features. In essence, an autoencoder is a deep network which outputs the same image that is given as input, even though the intermediate layers of the network have far less nodes than either its input or output layers. The first part of the network can be seen as an encoder (of the image data into a small set of features), and the second part can be seen as a decoder (of those features back into the original image).

In this experiment, we used only the second part – the decoder. We regarded the projections as the encoded data, and we feed these data as input to the network. At the output, we expect to have the cross section of the object (that is, the plasma density/emissivity at each pixel) and we regarded this as the decoded data. In other words, we regard the projections as an encoded description that needed to be “decoded” into an image of the cross section.

We opted for a network topology where each layer had twice the number of nodes of the previous layer, except for the last layer, which had as many nodes as required in the output (21 × 21). This progression in powers of 2 was similar to the ones that we found in existing literature on the autoencoder [19], and it rendered a network with six fully connected layers, as shown in figure 3.6.

![Network Topology Diagram](image)

Figure 3.6: Network topology used in the experiment

3.3.2 Deep learning framework

As a consequence of the rising interest in deep learning over the past few years, there has been a steady increase in the number of software frameworks which allow for fast development of deep neural networks. For this experiment, and the remaining work carried out in this thesis, we used Keras\(^1\), which is built on top of Theano [6, 3]. Both are

\(^1\)https://github.com/fchollet/keras
deep learning frameworks that can generate both CPU and GPU-executable (in this case, CUDA [38]) code, and are Python-based.

Building a neural network with Keras is a simple process. In the following lines of code, we built the network topology of figure 3.6:

```python
n_inputs = 11+11
n_outputs = 21*21
model = Sequential()

n = n_inputs
model.add(Dense(n, init='glorot_uniform', activation='tanh', input_dim=n))
while n*2 < n_outputs:
    n *= 2
    model.add(Dense(n, init='glorot_uniform', activation='tanh'))

n = n_outputs
model.add(Dense(n, init='glorot_uniform', activation='tanh'))
```

Listing 3.1: Python instructions to generate the network used in simple experiment.

### 3.3.3 Training parameters

With regard to network training, we used batch sizes ranging from 1 (network updates on every training example) to 100. The number of training epochs ranged from as few as 100 to as many as 200000, where larger batches were used. We generated two datasets, each with 10000 samples, and where each sample includes the projections and the emissivity values on the output grid. One dataset was used exclusively in the training phase, and the other in the test phase. In the training phase, we used 80% of the data for training, and 20% for validation, to be able to detect overfitting.

For weight initialization, we used Keras’s built-in Glorot-uniform distribution [15]. We used both sigmoid (more specifically, hyperbolic tangent) and ReLU activation functions, and defined the network error through a mean-square error loss function. Parameter updates were done through stochastic gradient descent (SGD). We also tested different values for the network learning rate, ranging from 0.01 to 10. Learning momentum and learning decay were not used.

Training time ranged from as little as 10 minutes, in cases where larger batches were used (and with fewer epochs), to as much as 9 hours, and was conducted on a laptop with an Intel Core i3 3217U processor at 1.8 GHz (as CPU) and an NVIDIA GeForce GT 710M with 96 cores at 775 MHz (as GPU).

### 3.3.4 Results

Experimenting with different combinations of the configurations described above, we achieved good results with a training batch size of 10, and by allowing the network to train for 10000 epochs. With these values, the total network training time was 9 hours and 10 minutes. We found that a smaller batch size (for example, 1) meant that the validation loss converged to a minimum initially faster, but often produced large fluctuations in later epochs. It also meant that we were not taking advantage of the GPU, because greater parallelization can be achieved with larger batches [4]. Specifically, for a batch size of 1, we saw that epochs ran faster on the CPU; however, with a batch size of 10, each epoch was already running significantly faster on the GPU – on average, 3 seconds, versus approximately 8 seconds on the CPU.
Figure 3.7: Validation loss per training epoch. At epoch 1, validation loss had a value of 0.00481; at epoch 10.000, its value was 0.00036.

For larger batch sizes (e.g. 100) the performance gain on the GPU was even higher, but the loss function converged much more slowly. In cases where we trained the network for much longer (i.e., 200000 epochs), we did not see significantly better results; in fact, we observed the occurrence of overfitting, since, at a certain point, the validation loss began to increase.

For our best results (10000 epochs with a batch size of 10), we do not believe that overfitting occurred, as we saw no indication that the validation loss was increasing. It did fluctuate from epoch to epoch, but its overall trend was one of decrease, as shown in figure 3.7.

Similarly, small learning rates such as 0.01 slowed down the convergence of the loss function, while large rates above 10 actually caused the loss function to diverge as epochs progressed. For that reason, we found 1.0 to be an optimal value for the learning rate, which allowed for a reasonably fast convergence (in terms of epochs) without large fluctuations.

Curiously, we found that using a network with a traditional activation function – the hyperbolic tangent – gave us better results than the ReLU, probably because the output of the hyperbolic tangent is limited to 1.0, which is also the maximum possible value of emissivity in the phantoms. The fact that we did not note the occurrence of vanishing gradients can probably be explained by the very large number of training epochs. We also found that weight initialization, in the short term, influences the loss function, but this effect becomes negligible as the network trains for more epochs.

In summary, the training parameters that we settled on in this experiment were: 10000 training examples, batch size of 10, learning rate of 1.0, 10000 epochs, hyperbolic tangent as activation function, Glorot-uniform weight initialization, and a validation split of 0.2.

After the network was trained, to obtain a measure of its accuracy we calculated the root mean square error (RSME) on the test set, which yielded a value of 1.87%. This means that, on average, the error of each pixel in the reconstruction is below 2%, which makes most reconstructions virtually indistinguishable from the original phantoms. This fact is illustrated in the sample results presented in figure 3.8.
Figure 3.8: Sample reconstructions from experiments with phantom profiles.
3.4 Conclusion

This chapter described the traditional uses of tomography and then presented its applications in fusion experiments. The inherent problem of plasma tomography is the very sparse projection data, which requires the usage of regularization methods to achieve correct reconstructions. These methods can easily generate artefacts in the reconstructed density profiles and tend to be computationally slow, even if that problem has been mitigated in the past few years with advancements in hardware and algorithms. Thus, their applicability for real-time plasma control remains rather limited.

With these problems in mind, neural networks have for some time been seen as a suitable alternative for plasma tomography, though previous experiments were done before the developments in the field of deep learning in the past few years. Thus, we carried out an experiment where we reconstructed phantoms using a fully connected decoder, which was significantly larger and deeper than the networks used before, and with a framework optimized for neural network development. The results lead us to believe that deep learning was indeed an effective tool for tomographic reconstruction. In the next phase of this thesis, we attempted to replicate this approach on actual data from JET.
Chapter 4

Experiments on JET Data

The results of the experiment conducted with a fully connected decoder on randomly generated plasma profile phantoms were promising. Therefore, it was safe to assume that our approach was, indeed, effective for the purpose of tomographic reconstruction. Our next step was to test the approach on actual data from plasma discharges carried out at JET.

The data was collected over a period of several years – from September 2011 to December 2015 – and consisted of a total of 18357 samples, where each sample contains two projections and the image reconstructed from those projections. In each sample, the projections were collected by the bolometer system existing at JET, while the image was the corresponding plasma cross section obtained through traditional tomographic methods. We wished to train a network using the projections as inputs, and the reconstructions as outputs.

Each data sample corresponded to an instant in time of a certain pulse, which is a plasma discharge in the JET tokamak. Pulses were identified by a number in the range 80176, 89152, where larger values are more recent. Some pulses contained only one measurement/image pair; others, such as pulse 84888, had as many as 98.

4.1 The KB5 diagnostic

KB5 is the bolometer system installed at JET, which measures radiation emitted by the plasma inside the tokamak [21, 22]. It is composed of two subsystems (cameras), KB5H and KB5V, whose lines of sight are shown in figure 4.1. KB5H and HB5V both have a total of 24 active channels, that is, they both measure radiation along 24 lines of sight, though KB5V also has 8 reserve channels [34].

With this configuration, we initially expected our network to have an input array of 48 values, corresponding to the 24 views of each camera. However, the actual number of bolometer measurements per sample in the data turned out to be 52. The 4 extra values per sample can be explained based on the fact that there are 8 reserve channels of KB5V which are typically used for calibration purposes [34]. In any case, this number of inputs contrasts with the 22 received by the fully connected decoder in our simple experiment (section 3.3).

Another obvious difference between the data and the phantoms we used for our experiment was the size of the plasma profile reconstructions: we had generated images of 21 × 21 pixels, while the actual plasma profiles in the JET data had 115 × 196 pixels. It quickly became apparent that using a fully connected decoder would be
unwise due to the size of the required network and the corresponding number of links. We therefore needed a new approach regarding our network topology, which we describe in detail in section 4.3.

Before carrying out any experiments with the JET data, we first attempted to discover any existing anomalies, as we did not have any up to date information on the state of the KB5 system. We did this by developing Python scripts that allowed us to thoroughly visualize the projection and image data we possessed. Essentially, the anomalies we found were:

- The data on the last 4 channels of KB5V (those that we did not expect to exist in the inputs), displayed a strange behavior: 3 of the channels sometimes had radiation measurements with negative values, while the measurements of the last channel were always 0.

- Another anomaly we observed in KB5V were occasional spikes in intensity so great – on the order of $10^8$ – that the measurements of all the other channels became insignificant (i.e. reduced to the level of noise) next to it. This problem was restricted to bolometers 6 and 16. Furthermore, bolometer 16, in addition to...
The behavior of KB5H seemed somewhat less anomalous, yet it also had one defining feature: the measurements of its bolometer 24 were always 0.

In all cases, these anomalies were most likely due to bolometer malfunctions, some of which we confirmed.

Though we knew these anomalies existed, we nonetheless decided to keep the samples in which they occurred. Our intuition was that since they constituted only a small fraction of the total data we possessed, the network should be able to learn to ignore them. On the other hand, failure to ignore those samples would be proof that the network was having trouble generalizing – i.e., it would be a symptom of overfitting.

### 4.2 Data preprocessing

Since the data was collected over several years, hardware and components of the JET tokamak may sometimes have been changed or upgraded. Thus, even ignoring anomalous situations, plasma measurements and image reconstructions could have different values and behavior as a result of changes in equipment.

For this reason, we decided to calculate several statistics – average, minimum, maximum, and median – over the samples in each pulse. Though by itself this was a somewhat coarse overview of the data, it nonetheless allowed us to check if any discernible patterns existed in samples from different times. In particular, we wished to verify if recent pulses differed significantly from the older ones. This was important to know before the training process because, if the data did indeed show great divergence over time, our network would attempt to fit very different samples, which would probably degrade its capacity for accurate reconstruction. If that were the case, it might make more sense to conduct separate training processes for different time frames.
We opted for a visual representation of the calculated values for the KB5 measurements, which can be seen in figure 4.4. To further simplify the analysis, we grouped pulses in their corresponding campaigns. A campaign is essentially a time frame along which pulses were carried out. This grouping is useful because equipment changes mostly occur in the time between two successive campaigns; therefore, large variation in data values, if it existed, was expectable mainly in transitions between campaigns.

<table>
<thead>
<tr>
<th>Campaign</th>
<th>Start pulse</th>
<th>Start date</th>
<th>Finish pulse</th>
<th>Finish date</th>
</tr>
</thead>
<tbody>
<tr>
<td>C28a</td>
<td>80176</td>
<td>2011/09/01</td>
<td>80372</td>
<td>2011/09/20</td>
</tr>
<tr>
<td>C28b</td>
<td>80653</td>
<td>2011/10/10</td>
<td>80976</td>
<td>2011/10/28</td>
</tr>
<tr>
<td>C28c</td>
<td>81264</td>
<td>2011/11/25</td>
<td>81643</td>
<td>2011/12/21</td>
</tr>
<tr>
<td>C29</td>
<td>81726</td>
<td>2012/01/10</td>
<td>82905</td>
<td>2012/04/10</td>
</tr>
<tr>
<td>C30b</td>
<td>82944</td>
<td>2012/05/15</td>
<td>83620</td>
<td>2012/07/14</td>
</tr>
<tr>
<td>C30c</td>
<td>83621</td>
<td>2012/07/16</td>
<td>83794</td>
<td>2012/07/27</td>
</tr>
<tr>
<td>C31</td>
<td>84442</td>
<td>2013/07/17</td>
<td>85355</td>
<td>2013/09/27</td>
</tr>
<tr>
<td>C32</td>
<td>85356</td>
<td>2013/09/30</td>
<td>85457</td>
<td>2013/10/04</td>
</tr>
<tr>
<td>C32a</td>
<td>85900</td>
<td>2014/01/08</td>
<td>85978</td>
<td>2014/01/10</td>
</tr>
<tr>
<td>C33</td>
<td>86452</td>
<td>2014/06/19</td>
<td>87583</td>
<td>2014/09/05</td>
</tr>
<tr>
<td>C34</td>
<td>87584</td>
<td>2014/09/10</td>
<td>87958</td>
<td>2014/10/09</td>
</tr>
<tr>
<td>C35</td>
<td>88941</td>
<td>2015/11/09</td>
<td>89472</td>
<td>2015/12/18</td>
</tr>
</tbody>
</table>

Table 4.1: Timeframe of JET campaigns and their respective pulses. Notice that campaigns skip pulses; for example, campaign C28a ends at pulse 80372, but the next campaign, C28b, starts at pulse 80653. For that reason, figure 4.4 have a white background for pulses that have no corresponding campaign.

Figure 4.4: Top to bottom: minimum, maximum, average and median of the radiation measured by KB5. Each blue dot represents the computed statistic for the KB5 measurements in a single pulse. The alternating green and brown backgrounds denote changes in campaigns. To the left are older pulses (and campaigns); to the right are more recent ones.
4.2.1 Filtering by campaigns

Looking at figure 4.4, in particular through the prism of average and maximum values of KB5 measurements, one can clearly see that our initial assumption – that data values potentially varied significantly over time – was correct: notice how the values in more recent campaigns tend to be larger. Regardless of this fact, some patterns were clearly distinct over time, in particular within groups of consecutive campaigns. Specifically, we noticed the existence of two campaign groupings – C29+C30b+C30c and C31+C32+C33 – containing data that showed overall similar trends. Also interestingly, the minimum values of all measurements in more recent campaigns are always 0, which means that one of the anomalies we mentioned in section 4.1 (that some bolometers had negative measurements) was restricted to the older campaigns.

Ultimately, we concluded that four main sets of data existed which might be useful to conduct separate training processes. Those sets were respectively composed by campaigns:

- C29 – 3238 projection/image samples;
- C30b + C30c – 2845 projection/image samples;
- C31 + C32 – 4162 projection/image samples;
- C33 – 4044 projection/image samples.

We grouped C30b + C30c and C31 + C32 because those campaigns were carried out at very close dates, and also so that the number of total data samples did not vary significantly from one set to another.

In each of these 4 sets, bolometer measurements and image pixel intensities have overall similar values, yet are still sufficiently dispersed so as not to be too similar – if they were, we would potentially be training a network too prone to overfitting, with no capacity for generalization. We judged the data in the other campaigns as being either too anomalous compared to its immediate neighbors (for example, C32a and C34), or as having too little variation (for example, C28b and C28c) and too few samples to be of use for training.

Though this process of filtering by campaigns implied throwing away the samples we deemed not useful (in essence, a form of data waste), the 4 chosen sets, taken together, had 14289 image/reconstruction samples out of a total 18357 in the raw data. This meant that the number of samples we threw away was only a fraction of the total. In our opinion, all 4 sets had sufficient data samples to allow for an efficient training process.

4.2.2 Principal Components Analysis

The projections we possessed each had, individually, a total of 52 dimensions/bolometer measurements. Even though we were now filtering data samples by campaigns, the individual network inputs could have noise, or their 52 dimensions might be correlated. In particular, if the input dimensions were correlated, the network might have had difficulty in extracting information from its inputs to generate their corresponding emissivity profiles.

Had we possessed unlimited resources, this, by itself, would have constituted no problem – we would merely have enlarged network power by increasing the number of nodes, and hoped that the network used that extra capacity to extract more information from the data. In practice, we had no a priori guarantee that this would occur
when the network trained; and, in any case, we were indeed limited by the amount of memory and GPU power, so this was not an option.

In short, our intuition was that we could use some form of pre-processing step that allowed us to decorrelate the network input dimensions. This should ease the task of discovering the most relevant information to reconstruct the emissivity profiles, while using with the same computing power.

We decided to achieve this by using Principal Components Analysis (PCA), because of its common usage as preprocessing technique in deep learning tasks [48, 37]. Using PCA was also in line with the work done by Ronchi et al. [44], who used the technique in experiments with neural networks for tomographic reconstruction. We performed PCA by using Python’s scikit-learn library.

Apart from decorrelating the data, PCA also allowed us to see that, in the dataset composed by campaigns C30b + C30C, 6 of the 52 input data dimensions had no variance whatsoever, which probably means that the corresponding KB5 channels were not being used. This phenomenon also occurred in the other 3 campaign sets, though not on as many channels; in those cases, only 2 input dimensions had no variance. These probably corresponded to the bolometers whose values were always 0, which we had already detected (section 4.1). We therefore decided to remove these 2 dimensions from all the samples in the network input data, and use only the remaining 50, in all campaign sets.

One other modification we performed on the data was the scaling of data samples. There were two reasons for this. Firstly, PCA is usually combined together with some form of normalization or scaling of the data on which it is applied. Furthermore, since our raw network outputs had very large and dispersed values (section 4.1), the validation loss in the training process would be potentially hard to interpret; scaling the outputs would make loss values smaller and easier to understand.

### 4.3 The Up-convolutional Network

Section 2.4 described convolutional neural networks and their typical uses – image recognition and classification. The chapter also explained one of the main benefits of CNN’s: that the number of total network links does not grow significantly even with larger inputs.

As we saw in section 4.1, our existing plasma profile reconstructions are $115 \times 196$ images. The ideal network to output results that large would be one where the number of parameters could be kept small, as in a CNN. Usually, a CNN receives as input an image, and outputs some encoding of that image, which in most cases is a probability of the image belonging to a certain class.

On the other hand, the data we would be processing (the projections measured by the JET bolometer system) could be considered an encoding of the plasma profile, as we have seen in section 3.3. Thus, our problem was the opposite of the one solved by a CNN: we required a network topology that did not increase in size regardless of the output dimensions, and with the ability to reconstruct an image given an encoding of that image.

For this reason, we used the opposite of a CNN – an up-convolutional neural network (uCNN) – where the typical progression of convolutional, pooling and fully connected layers is turned around, as done by Dosovitskiy et al [14]. The key in the functioning of that network is the “up-convolution” operation, which can be thought of

---

as the opposite of the convolution + pooling operations typical in CNN’s.

4.3.1 Symmetry of convolution and deconvolution

Intuitively, an up-convolution, having essentially the opposite effect of a convolution + pooling operation, would also be built on top of the inverses of those operations: that is, unpooling and deconvolution. In practice, deconvolution and convolution are symmetrical [53, 39] and therefore, an up-convolution can be done by an unpooling followed by a regular convolution [14, 45].

The symmetry of convolution and deconvolution can be better understood in figure 4.5. Though we have so far talked only about multi-dimensional convolution, the operation can also be done in only one dimension, which, to simplify our demonstration, is the case we use here.

Figure 4.5: Symmetry of convolution and deconvolution in 1 dimension. Both filters have a length of 3 and are sliding along the dimension, as they would along the width and height of a 2D input. The blue filter is currently convolving the top nodes into the middle lower node (B), while the cyan filter is convolving the lower nodes into the middle top node (A).

Suppose that the original network is top-down. When the blue filter passes by A, its output is successively fed to the three bottom nodes, albeit to each with different weights – that is, A contributes to the output of those three nodes. Suppose now that the network was then flipped, and analysed bottom-up. Intuitively, if we wished to obtain A’s original input, we would have to make some combination (weighted sum) of the outputs of the nodes to which A originally contributed. That combination would be the operation performed by the cyan convolution – which is symmetrical to the blue one. In effect, the operations are the same – though the weights used in each convolution would be different. For this reason, we ultimately opted to carry out de-convolution by using the already-existing convolution operation implemented by the Keras framework.

4.3.2 Unpooling

The same way a pooling operation will cut down on its input’s size, an unpooling operation will have the opposite effect – given an input, it will output a result larger in width and height. Regular max pooling operations are irreversible: only the maximum value of an input is propagated to the next layer (2.4.2), while all other information is discarded. Therefore, in an unpooling operation, we intuitively know what value to output – the same value as the input. A more difficult question to answer, however, is where to place that value in the output – and what values will fill the rest of the enlarged output region.

Some networks[54, 52] use so-called switch variables that, in each regular pooling operation, store the input position that contains the pooling output value. Thus, when an unpooling operation is to be executed, those variables are retrieved and an input value can be mapped to the correct output location, while the remainder of the
output is filled with zeros (the original information is irretrievably lost in the pooling operation). However, those networks are significantly different from the one we wish to use. Though they use deconvolution and unpooling, their goal is to study the inner working of vanilla convolutional networks by attempting to use their feature maps to regenerate their inputs, and thus get an idea of what features the network filters are actually looking for.

Alternatively, one can create an unpooling operation that always maps its input to a particular location of the output\cite{14}, while leaving the remainder of the output also filled with zeros.

Our work uses the unpooling operation currently implemented by the Keras framework (UpSampling2D, as named in the Keras API)\cite{14}. This particular implementation of unpooling merely copies the input to all locations of the output region, as shown in figure 4.6.

Our work uses the unpooling operation currently implemented by the Keras framework (UpSampling2D, as named in the Keras API)\cite{14}. This particular implementation of unpooling merely copies the input to all locations of the output region, as shown in figure 4.6.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.6.png}
\caption{Inner working of the Keras unpooling operation. In this case, output has a depth of 1 (the same as the input). This particular unpooling operation has a range of $2 \times 2$ – that is, it doubles its input width and length.}
\end{figure}

### 4.3.3 Topology

The topology of our up-convolutional network is thus a sequence of fully connected layers and up-convolutions. Inputs are a 1D array containing the JET bolometer signals of one data sample. Those inputs are fed to two FC layers with 7500 nodes each. This number was chosen because our experiments showed that larger FC layers yielded significantly better results than smaller ones – which leads us to believe that a major part of the network learning process occurs in these layers. We did not use even more FC layers because we were operating at almost the limit of available GPU memory. In any case, an experiment was done where we pushed the number of FC layers nodes to 9000 (roughly the limit at which no more memory was available), and this yielded essentially the same results.

The output of the second FC layer is reshaped from a sequence of 7500 values to a $(20 \times 15 \times 25)$ matrix, where each dimension, respectively, corresponds to depth, width, and height. That matrix can already be thought of as a rough “draft” of the final network output, but one where each of the 20 samples along the depth dimension is a particular representation (channel) emphasizing certain features of the final image. It will then be up to the up-convolutions to focus, zoom-in, smoothen and combine those channels to generate the final output, which will be a 2D plasma profile. All layers in the network use ReLU activation functions.

The reshaped data is fed to three sequential up-convolutions, each consisting of one $(2 \times 2)$ un-pooling operation and two $5 \times 5$ convolutions with 30 feature maps (the double convolution is based on the work of Dosovitskiy et al. \cite{14}, which concluded that this generated better results than a single convolution). Similarly to what we observed in the FC layers, increasing the number of feature maps in the up-convolutions, for example, to 50, did not yield better results. Both convolutions at each level have a stride of 1 (i.e., they slide trough all their inputs).

\footnote{http://keras.io/layers/convolutional/}
One final, single convolution with only 1 feature map combines the features coming from the lower-level layers, and gives the network output. All convolutions are padded to preserve input size. The unpooling/upsampling operations are responsible for the actual zooming in on the FC layer output; the convolutional layers can be thought of as performing a search for a particular feature in the unpooling operation output, and performing a smoothing of the image.

With this progression of up-convolutions, the final output shape of the network is $120 \times 200$. However, the existing tomographic reconstructions in the JET data had a size of $115 \times 196$ (section 4.1). This meant that the network would not be able to train with the existing reconstructions because their shapes did not match with the network output. To solve this problem, we decided to pad all images used in network training with the additional columns and rows entirely filled with zeros, thus equalizing their shape with that of the output. We did not judge this as being problematic because the first and last rows and columns of the existing images were, themselves, already filled with zeros.

The lines of code in listing 4.1 give a better overview of the data shape as it flows through the network, from the bottom to the top layers.

```python
model.add(Dense(7500, input_dim=50))
model.add(Activation('relu'))
model.add(Dense(7500))
model.add(Activation('relu'))
model.add(Reshape((20,15,25)))  # Output shape = (20*15*25)
model.add(UpSampling2D(size=(2,2)))  # Output shape = (30*30*50)
model.add(Convolution2D(30, 5, 5, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(30, 5, 5, border_mode='same'))
model.add(Activation('relu'))
model.add(UpSampling2D(size=(2,2)))  # Output shape = (30*60*100)
model.add(Convolution2D(30, 5, 5, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(30, 5, 5, border_mode='same'))
model.add(Activation('relu'))
model.add(UpSampling2D(size=(2,2)))  # Output shape = (30*120*200)
model.add(Convolution2D(30, 5, 5, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(30, 5, 5, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(1, 5, 5, border_mode='same'))  # Final convolution combines all feature maps. Output shape = (1*120*200)
```

Listing 4.1: Python instructions (with the Keras framework) to generate our up-convolutional network. The code allows the easy visualization of the shape of the data as it moves to the upper layers. Notice how the input shape is 50, not 52 (because its dimensionality is reduced by PCA).

Crucially, what distinguishes the up-convolutional layers from what would otherwise be a simple zoom-in on the output of the FC layers is that, with the exception of the final convolution, they always work on a total of 30 feature maps. Since filters will, during the training process, adjust themselves to look for particular features in their input, it stands to reason that each up-convolution along the depth dimension will be highly specialized in
zooming in on some feature – while a simple zoom-in would focus on the entire FC layer output at once.

As is the case with typical CNN’s, we do not actually know, \textit{a priori}, what features the network will be looking for to zoom in on [52]. What we do know, from our experiments, is that having several feature maps, each specialized in focusing on a particular feature (or set of features) is the key to obtaining a good final image of the plasma profile – though our experiments also showed that increasing the number of feature maps only helps so far. At a certain point, increasing their number will actually make network accuracy on the final image output begin to decrease.

![Figure 4.7: Schematic of our up-convolutional network.](image)

4.4 Training and results

We had, before training the network, decided to use 4 data sets corresponding to different JET campaign time frames, and carry out separate training processes for each of them. In all those cases, we used 90\% of the available data for training/validation, and 10\% for testing. In the training/validation phase, we further subdivided the data into 90\% for the training set, and 10\% for the validation set, which was used to monitor the training progress. Thus, we had a total of 4 training, validation and test sets, corresponding to different JET campaign time frames.

We decided to use simple stochastic gradient descent as our network loss optimizer. This decision was due to the fact that by using SGD, we could directly tune the network learning rate – as opposed to other more advanced optimizers, such as Adagrad or Adadelta, that automatically change it. Tuning the learning rate turned out to be an important factor. Specifically, with the scaling of the data we had carried out before training, we obtained good results with a small learning rate – 0.001. Larger learning rates meant that overfitting quickly became a problem, with large validation loss fluctuations from epoch to epoch. Large fluctuations made it difficult to judge whether the training process was progressing well. They also meant that we could achieve a final result that was very good, but only because of a coincidence – that is, in the final epoch, some fluctuation might have occurred that rendered
a very good result, but had no effective meaning in the long-term trend of the training process. Also for these reasons, learning rate momentum and decay were not used.

Weight initialization used the Glorot uniform distribution implemented by Keras. We used a batch size of 10, i.e., network parameters were updated based on the error value for batches of 10 training samples. 4 training processes were carried out, each one with one of the 4 data sets we had previously chosen.

The loss function we chose was the mean absolute error (MAE), also implemented by Keras. The choice of this loss function meant that the network would attempt to carry out a pixel-by-pixel optimization on the absolute error of its outputs. We chose it because of its common usage as an optimizer.

We ran the network on an NVIDIA GeForce GTX 480 GPU with 480 CUDA cores at 1401 MHz. Training took approximately 48 hours for each of the 4 sets of campaigns.

In each of the 4 training cases the overall trend of the validation loss was to decrease (along with the training loss) until around epoch 1500, when it started to behave asymptotically. Because of this, we have no reason to believe that overfitting occurred until that time, and therefore, the 4 training processes were carried out for 1500 epochs.

![Loss and validation loss progression in all training processes.](image)

Figure 4.8: Loss (blue) and validation loss (green) progression in all training processes. Y-axis: MAE values. X-axis: training epoch.

An alternative to MAE would have been to train the network using Keras’s mean squared error (MSE) loss function. However, we did not use it, because MSE greatly compounds large errors relative to small ones and consequently may make the training process more difficult, with greater loss fluctuations from epoch to epoch. Similarly, it generates loss and validation loss values that, for experimental purposes, are harder to interpret than MAE values.

In any case, both MAE and MSE are potentially poor predictors of signal fidelity [49]. Specifically, the MAE...
is computed as the average over the absolute errors of all pixels of a network output. This means that it is possible for an output as a whole to have a relatively good average loss value, when in practice, what is happening is that some of its pixels are perfectly adjusted to their true values, while others have potentially large deviations. This is also true for other loss functions such as mean square error (MSE).

Thus, we restricted the use of MAE to the training process – that is, we only calculated it on the training and validation sets, to monitor the training progress. Measurement of the actual, final image quality (post-training, on the test set of each data grouping) was done by using three other metrics. Those were the Normalized Root Mean Squared Error (NRMSE), Peak Signal-to-Noise Ratio (PSNR) [23] and Structural Similarity (SSIM) [50]. Each metric compared the already existing reconstructions (in the JET data) with the ones generated by the network. In all cases, we performed a scaling of the images, squeezing all pixel values to the range \([1; 1]\).

The main advantage of using NRMSE over MAE or even MSE is that NRMSE is normalized over each input; therefore, we can compare the error of two output samples even if they have significantly different values (and indeed, the data preprocessing stage of our work had shown us that output image pixel values could have significant variation). We used PSNR because it is commonly used in image quality assessment. PSNR is logarithmic and is measured in decibels, which means that even a small increase in its value can indicate a significant increase in signal quality. PSNR has no maximum value \(\text{per se}\); the intuition behind it is merely that the larger it is, the better.

Another reason for using those two metrics is that we did not want to restrict ourselves to the use of SSIM, whose inner working is significantly different. Unlike other error metrics, SSIM attempts to measure structural changes over the entire image, rather than performing a point-by-point error measurement. SSIM’s output values are in the interval \([-1; 1]\), where an SSIM value of \(-1\) means two images are completely different, and \(1\) means they are identical.

To calculate all these metrics on our output data, we used used python’s scikit-image library \(^3\).  

<table>
<thead>
<tr>
<th></th>
<th>SSIM</th>
<th>NRMSE</th>
<th>PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>worst</td>
<td>mean</td>
<td>best</td>
</tr>
<tr>
<td>C29</td>
<td>0.899</td>
<td>0.980</td>
<td>0.999</td>
</tr>
<tr>
<td>C30b + C30c</td>
<td>0.922</td>
<td>0.964</td>
<td>0.985</td>
</tr>
<tr>
<td>C31 + C32</td>
<td>0.953</td>
<td>0.990</td>
<td>0.998</td>
</tr>
<tr>
<td>C33</td>
<td>0.895</td>
<td>0.986</td>
<td>0.998</td>
</tr>
</tbody>
</table>

Table 4.2: Final values of SSIM, NRMSE and PSNR.

Table 4.2 shows, for each metric, the minimum, mean and maximum value over each test set. All image data was scaled to values in the interval \([-1; 1]\) before the metrics were calculated. The overall worst values for SSIM (0.895), NRMSE (0.096) and PSNR (27.208) all occurred in images generated from data in campaign C33. The best values for SSIM are all very close to 1, which indicates that most of the reconstructions carried out by the network were, according to that metric, very good reproductions of the originals. The PSNR values were similar to those expectable in image compression tasks, which usually generate a PSNR between 30 and 50. The NRMSE gives, overall, similarly good values.

Though this was not the main focus of our work, we also took a rough estimate of the time it took the trained

\(^3\)http://scikit-image.org/docs/dev/api/skimage.measure.html
network to reconstruct individual images. In all the 4 trained networks, this was around 10ms.

Finally, we took some samples of individual reconstructions to have a sense of the images the network was producing. These can be seen in figure 4.9.
Figure 4.9: Sample reconstructions from experiments with JET plasma profiles. In each pair, the left image is the profile obtained by traditional tomography, while the one on the right is the one reproduced by our neural network. In each line, two reconstructions are presented from each set of campaigns. From top to bottom: C29; C30b+C30c; C31+C32; C33.
Chapter 5

Conclusion

We used an up-convolutional neural network to generate tomographic reconstructions of plasma emissivity profiles based on plasma projection data. We divided our data into several sets corresponding to different timeframes of JET campaigns. In all of those cases, our overall results were good, with the overall worst values being 0.895 for SSIM, 0.096 for NRMSE and 27.208 for PSNR. Since all of these occurred in the same campaign (C33), one can conclude that there probably was some inherent aspect of the data from that time that rendered training more difficult. The results also showed that while the network training process itself may take some time, post-training image reconstruction is quick. This is in line with the results from previous experiments with neural networks for tomographic reconstruction.

We trained the network on projection/emissivity data samples, where the emissivity profiles were obtained through traditional tomographic methods. This means that the accuracy of our network is ultimately limited by the quality of the existing reconstructions. For this reason, in future work, one way to increase the accuracy of our results is to make sure the reconstructions obtained through traditional tomography are more accurate than the ones currently existing. Similarly, we believe that the existing reconstructions are generally very similar to each other. Better results (or at least, better capacity for network generalization) might be achieved by training with images with greater variability.

Our network was optimized using Mean Absolute Error because of its common usage in deep learning experiments. While this optimizer allowed us to achieve good results, it is not necessarily the best one, because it performs a pixel-by-pixel optimization. This means that outputs may simultaneously have pixels with good and poor values. The values of the three additional metrics we used lead us to believe that this phenomenon, if it exists, is somewhat restricted. Nevertheless, one possible improvement to our results would be to develop some network training optimizer based on, for instance, SSIM. Another (probably more complex) possibility would be an optimizer akin to the work done by Dosovitskiy and Brox in [13], which uses a loss function that computes image distances (i.e., error) using image features. One additional way to check the accuracy of our results would be to perform a cross-validation with some other diagnostic existing at JET.

We did not further increase network size due to lack of computing power. In any case, our experiments indicate that there are no significant benefits in increasing the network size.

All in all, our conclusion is that, using this hardware and network topology, the best improvements to be
made would be to use better training sets or, perhaps more importantly, using alternative (more appropriate) loss functions than MAE. Nevertheless, we believe our results were satisfactory.

5.1 Contributions

In summary, in this work we have achieved the following main results:

- We have confirmed that neural networks are a viable approach to produce the reconstructions that are typically obtained through plasma tomography algorithms.

- We have devised an up-convolutional network to perform the opposite function of a CNN. While a CNN turns a 2D input image into a 1D vector of class probabilities, with the up-convolutional network we turn a 1D vector of sensor data into a 2D reconstruction image.

- We have applied the up-convolutional network to data collected from a tomography diagnostic installed at JET, and we have shown that the network, once trained, can reproduce the reconstructions with high accuracy, as indicated by a structural similarity (SSIM) of about 0.98.

- We have realized that some of the main obstacles in training a deep neural network are constraints on memory and time. On one hand, the available GPU memory limits the network size – particularly where the FC layers are concerned. On the other hand, it is impractical to keep training the network for more than a few days, so eventually the training must be stopped at a point where the validation loss was still decreasing.

- We have not observed overfitting (i.e. the point at which the validation loss starts increasing). This is an indication that there might be some room for improvement, particularly by training for more epochs.

5.2 Future Work

With the constant advances in computer hardware, one might expect, in the future, better GPUs, with more memory and processing power. Taking this into account, one way to improve on our approach is to increase the size of our network, especially in the FC layers, and also to train it for more epochs – at least until clear signs of overfitting are detected. We expected that the same kind of deep learning approach can be applied to other diagnostics or other fusion devices with tomography diagnostics.
Bibliography


