Cardiff University

Computer Science and Informatics

CM3202 - Individual Project

AI to Detect Early Signs of Alzheimer's Disease

Final report

Author:

Jacob Winkworth

Supervisor:

Xianfang Sun

Abstract

As machine learning continues to perpetuate through all aspects of society, we must question its ability to improve all aspects of life, including the medical industry. With the average age of the global population on the rise, Alzheimer's disease is soon expected to reach epidemic proportions. Advancements in neuroimaging are currently leading the field in the diagnosis of the condition, but the earlier these signs are detected the more effective the treatment for the reduction of symptoms; a syndrome more commonly known as Dementia. This project aims to develop and train a deep learning model with the ability to distinguish Alzheimer's magnetic resonance imaging (MRI) from normal, healthy control data.

Acknowledgements

I would like to than my supervisor Xianfang Sun for his contribution to this project.

I would also like to thank all researchers at the ADNI for continuing to develop and advance the medical industry.

Table of Contents

Table of Figures

Introduction

Over the last two decades, neuroimaging has been at the forefront of both psychiatric and neurological analysis. Studies related to the comparison of healthy control subjects from those suffering with such disorders have depended heavily on mass-univariate analytical techniques. Such have yielded results reporting neuroanatomical differences in subjects at group level, leading to significant advances in our understanding of the neurobiology of a multitude of various neurological conditions. However, mass-univariate analytical techniques draw the assumption that regions of the brain act independently. This contradicts our current understanding of brain function, which suggests that the brain is intrinsically organized into dynamic, anticorrelated functional networks [1]. Furthermore, these techniques employ methods that detect disparities at group level, but do not allow for statistical inferences on a subject-wise basis. This negatively impacts the translational benefits from research to practice, as clinicians are required to make diagnostic conclusions based on the study of an individual.

As the field of neuroimaging advances, those researching approaches to statistical analysis of subjects have developed a growing interest in machine learning (ML). This is an area of artificial intelligence that focuses on the development of algorithms to discover trends in existing data that can be used to make predictions on unseen data. Firstly, unlike before, these are multivariate techniques that can recognise interdependencies between voxels from separate neural regions, thus improving upon the first univariate limitation. Secondly, these learned patterns can be used to make inferences on an individual level, as a clinician would. This resolves the second issue with univariate analysis, furthering the functionality of statistical models translated to applications in neuroimaging.

Since the comeuppance of ML in neuroimaging, various models have been tested across a diverse range of modalities, such as MRI, fMRI, and PET scans. One popular technique is Support Vector Machine (SVM), which finds a hyperplane in an N-dimensional space that distinctly classifies a set of data points. However, criticism has been directed towards SVM for its requirement of expert inspection to extract useful regions of interest from raw formats; a process referred to as 'feature selection' [2]. Thus, it is essential that those engaged in these tasks are competent in both the ML and neurology disciplines. This has led researchers to study methods within deep learning (DL), another branch on the ML tree. These models follow hierarchical structures with multiple levels of complexity that obtain higher levels of abstraction. This is achieved through the application of consecutive nonlinear transformations to the raw data. Motivation for the development of these models has arisen from the study of human

neuroanatomy, with the building blocks of deep-learning networks – artificial neurons – analogous to neurons within the brain.

This project focuses more deeply on one field from each of the three major disciplines mentioned thus far: Alzheimer's disease (AD) from neurology, magnetic resonance imaging (MRI) from neuroimaging and deep convolutional neural networks (CNNs) from machine learning. AD is an irreversible, progressive, neurodegenerative disorder, characterised among those within the elderly community. It is the leading cause of dementia, which, in 2015, was the most common cause of death for females, and the most common cause for death for males over the age of 80 [3]. As the global average life expectancy increases, AD is becoming more and more prevalent across society. It is projected that around 16 million Americans will be living with Alzheimer's in 2050, costing around $1.1 trillion (in 2020 USD) [4].

Medical imaging has been both theorised and practiced for over a century, but only of recent has it been commonly available to most major hospitals. MRI is a common imaging technique used predominantly in the medical industry for establishing a 3D portrait of anatomy in a digitized form. Both a strong magnetic field and radio waves are used to create a detailed image of the region of interest (roi), such as the brain or abdomen. This can aid in both diagnosis and prognosis of patients suffering from a multitude of various mental and physiological health defects. Such defects caused by Alzheimer's disease include abnormalities associated with mild cognitive impairment (MCI) and the decrease in size of multiple regions within the brain. These regions include the hippocampus (temporal lobe) (the forming of new memories), the frontal lobe (intelligence, judgement, behaviour) and the parietal lobe (language) [5]. Earlier detection of such abnormalities via thorough analysis of MRI scans may lead to the reduction of symptoms in a patient.

A CNN is an application of deep learning whereby a series of kernel convolutions breaks apart an image into its constituent features to map spatial relationships across a group of data. CNNs have been successfully implemented numerous times to develop classification algorithms for the field of neuroimaging [6] [7]. Particularly, those that can distinguish MRIs presenting a patient suffering with a neurological condition from healthy control data. MRIs successfully capture anatomical images in 3D space; therefore, this project will follow the process of constructing a 3D CNN that can differentiate MRI scans of patients with neurodegeneration caused by AD from normal subjects. More precisely, it will invoke a binary classifier that measures disparities between the two groups. The network will be trained and tested using the ADNI (Alzheimer's Disease Neuroimaging Initiative) dataset, which contains thousands of images in different modalities acquired over the last two decades.

One issue faced with such studies is the computational work required for rigid pre-processing stages that normalize the input data and rid redundant information from the scans. Thus, five datasets will be produced from a series of pre-processing measures that will register, skull-strip and smooth each MRI scan. Moreover, the formerly mentioned CNN will be tested across all five datasets to conclude which of the five approaches best suits the development of a basic neural network model for neurological analysis.

To reiterate, the overarching aim of this project is to develop and end-to-end pipeline that takes as input a series of MRI volumes, performs a series of complex pre-processing steps, and classifies individuals accordingly.

Background

1. Machine Learning

Overview

Machine learning is an application of artificial intelligence (AI) driven by observations pertaining to the natural world. A system that implements a machine learning algorithm will continually improve from experience without being explicitly programmed. A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [4].

These models fall under two categories: supervised and unsupervised. A supervised machine learning model must be given data that has been explicitly categorized by a label, and the model must be able to interpret both the datum and its label as paired entities. Whereas this is not a requirement for an unsupervised model. This project will adopt the former of the two. Next, the nature of the machine learning problem to be solved is recognized. Problems of relevance to supervised machine learning models can be further categorized as classification and regression problems. Classification problems attempt to map input data to a discrete set of variables, such as 'cat' or 'dog'. However, a regression problem handles real or continuous values, such as 'age' or 'salary'. Again, this project will be solving the former.

Supervised Learning

Supervised learning is the most common subbranch of machine learning today. In layman's terms, it can be thought of as a teacher overseeing the operation of the system. During the training phase, labelled data is used to infer a learning algorithm. These labels correctly classify datum into their corresponding category, which the model uses to search for patterns that correlate to the desired output. Following this, the model will be presented with a new set of inputs and will attempt to determine labels for each based on its previous learning experience. At its most basic form, a supervised learning algorithm can be represented as $Y = f(x)$, where Y is the predicted output, f is the mapping function and x is the input [9].

Classification problems in machine learning

Classification predictive modelling is the task of approximating a mapping function from input variables to discrete output variables. These problems belong to the category of supervised learning, where the target is also provided with the input data. There are many applications in classification in many domains, namely spam detection, credit approval and target marketing. In binary classification problems, the class space is comprised of only two categories. This project

poses the solution to a binary classification problem in medical diagnosis. In binary classification problems, the class space is comprised of only two categories, which are populated with our Alzheimer's and control MRI data.

Model Fitting

Overview

The crux of machine learning is to obtain a model with optimum values of parameters and hyperparameters such that they generalize to a set of similar data to that on which it was trained. This is model fitting, and practitioners often observe one of three phenomena during this stage: underfitting, good / robust fitting and overfitting.

Underfitting

A model is said to be underfitted if it does not observe the underlying patterns that correlate inputs to desired outputs on a set of training data. This often occurs when a there is a lack of data to build a complete and accurate model. As shown by the left-most graph, the model has not learned enough about the data to observe trends across a distribution of values. Thus, it has formed a linear fitting on a set of non-linear data.

Robust fit

The middle graph shows a suitable fitting of a model to the data. Here, the system has learned the general trend of patterns between training data without observing its noise and fluctuations. Thus, it can form close predictions given unseen inputs.

Overfitting

A model is said to be overfitted if it learns the relationships between training data too well such that it cannot generalize for unseen data. This can occur when the set of training data is too large for the problem. As shown by the right-most graph in figure 1, the model has accounted for noise and fluctuations as well as a general trend. Overfitted models can be identified by the large variance in accuracy between training and unseen data. Thus, it will form abnormally close predictions on training data, but will underperform when given new inputs. In classical machine learning, overfitting can be defined as:

Given a hypothesis space H, a hypothesis h E H is said to overfit the training data if there exists some alternative hypothesis h' E H, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances [4].

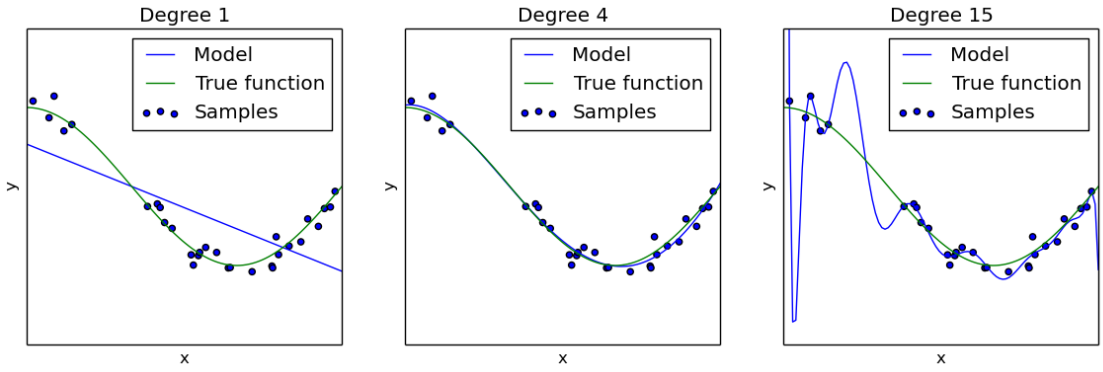

Figure 1 Underfitting vs Robust Fit vs Overfitting [10]

2. Deep learning

Overview

Deep learning refers to the training and testing of multi-layered artificial neural networks (ANN)s that can learn complex structures and achieve high levels of abstraction. ANNs have been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons [4]. It is estimated that the human brain consists of around 86 billion of these neurons, forming networks of such intricacy that human interpretation is infeasible [8]. This is a black-box approach to problem solving, whereby information is gathered from the environment, hidden calculations are performed, and a response is produced.

The implementation of DL in the context of supervised classification problems involves two main steps: training and testing. In the first step, a subset of the available data known as the training set is used to optimize the network's parameters to perform the desired task (classification). Learning is achieved through an iterative process of adjustment of the interconnections between the artificial neurons within the network, much like in the human brain. In the second step, the remainder subset – the test set – is used to assess whether the trained model can blind-predict the class of new observations. This section will cover the basic principles of Artificial Neural Networks, as well as introduce more complex schemas.

Artificial Neural Network basics

Structure and operation

The most elementary form of a deep neural network is the multilayer perceptron; a precursor to more complex models. It is formed of three layers: an input layer, a hidden layer, and an output layer. Each layer takes some vector, performs calculations, and outputs another vector to be fed to the next layer. At the input layer, the data enters the model. In neuroimaging, this can be represented as a one-dimensional vector with each value corresponding to the intensity of one voxel. The hidden layers learn and store increasingly more abstract features of the data. These features are then fed to the output layer that assigns the observations to classes.

Layers are comprised of nodes, or artificial neurons, such that each neuron is fully connected to all those in the previous layer. Each input neuron, x_i, has an associated weight value, w_i, which reflects the strength of that connection; much like a synapse between two biological neurons. Each layer also has an associated bias, b, which has its own weighted connection, b_i, to each neuron. The pre-activation is the sum of all weighted inputs, $\sum x_i w_i + b_i$, which is then passed through some non-linear activation function, f, that squashes the value between some range. The greater the yield

from this activation function, the greater the activation of the neuron. The output of this activation function, y_i, then serves as input to some neuron in the next layer.

Notable activation functions are the rectifier function (where neurons that use it are called rectified linear unit (ReLU)), the hyperbolic tangent function (Tanh), the sigmoid function and the softmax function. The latter is commonly used in the output layer as it can compute the probability of multiclass labels. However, for binary classification problems, the preference is generally to use sigmoid in the output with only one activation neuron and one class label (labels in the form of [1] or [0] for AD / control), versus softmax with one activation neuron and one-hot encoding (labels in the form of [1, 0] / [0, 1] for AD / control).

The input signal propagates through the network until the output layer is reached. For a classification algorithm, each neuron in this layer represents some class identifier. The most activated among these will be the final prediction for the model, with the value representing the certainty at which that prediction has been made. The loss is then backpropagated through the network, and neurons in the hidden layer(s) that directly contribute to the output will have their weights adjusted accordingly. This continues until each layer receives the loss signal.

Training and testing

Parameters and hyperparameters

When an ANN is being trained, it will attempt to determine the set of weight variables. These are known as network parameters, whose value can be either estimated or learned from the data. Network parameters are paramount to the operation of the system and are (often) not set by the practitioner, but rather learned iteratively by the network. However, model hyperparameters are a set of configuration variables that cannot be learned through training data. These are set and altered by the practitioner to obtain a model with optimum performance.

Loss functions

The loss function is a method of measuring how good the neural network is at a certain task [9]. To improve the network, the loss function must be minimized. It can be said that the problem of training the neural network is equivalent to the problem of minimizing the loss function. For binary classification problems, a common loss function is binary cross entropy. Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1 (does not have AD / has AD). Cross-entropy loss increases as the predicted probability diverges from the actual label. For binary classification, where the number of classes is two, cross-entropy can be calculated as:

$$-(y \log p + (1 - y) \log (1 - p)$$

Where log is the natural log, y is the binary indicator (0 or 1) if class label **c** is the correct classification for observation **o** and p is the predicted probability observation **o** is of class **c.**

Backpropagation

Backpropagation is a recursive process that feeds the loss signal for a single training example back through each layer of the network. The idea is to calculate how each layer should adjust its parameters to positively affect the layer ahead (when an input signal propagates forward) vias some optimization function. Each neuron in a given layer has its own desired adjustments for the connection weights between itself and neurons in the previous layer. The desires of all neurons are totalled to produce an overall list of adjustments for that layer. This process is recursively applied to each layer in the network until the input layer is reached and a final list is given.

Gradient descent

Artificial neural networks are trained through the adjustments of the weights *w* and bias *b* for each neuron. The model is first initialised with random values for these parameters, and adjustments are made during training to reduce the loss function via some specified optimisation function. An optimisation function may be represented as *o(w)*, where *w* is the set of parameters for the network. Traditionally, neural networks can learn through a gradient descent-based algorithm. The gradient descent algorithm aims to find the values of the network weights that best minimise the error (difference) between the estimated and true outputs, calculated via some loss function. Consider a large valley in 3D space. A person, stood at the crest of the hill, wishes to descend to the lowest point in the most efficient way. Trivially, at each point, they step in the direction with the most negative gradient, until they reach a point where the gradient in every direction is positive. This, essentially, is an analogy for backpropagation in a neural network using gradient descent optimisation. Gradient descent attempts to find the global minimum of the loss function. Given the loss function, *L*, the vector for the gradient of steepest descent at some point is given by $-\Delta L(W)$, where *L* is the loss function, $-\Delta L$ is the negative of the gradient (calculated from the differential) at that point, and *W*, is the vector containing all network parameters.

Each component of the (negative) gradient vector yields two pieces of information. The sign represents whether the corresponding component in the weight vector, *W*, should be increased or decreased, and the magnitude represents the relative importance of that change. That is, an adjustment to one weight may have a larger impact on the loss function than the same adjustment on a different weight. The weights will also be adjusted according to some learning rate, which is a hyperparameter that controls how much to change the model in response to the estimated error

each time the model weights are updated [10]. I.e., these parameters will be updated according to the learning rate, which is set by the practitioner. This makes each step proportional to the gradient at that point, allowing for smaller steps as the cost converges to some minimum. Selecting a good learning rate for a model is crucial. Learning rates too large will miss global minima, whereas learning rates too small will take a long time to converge.

Stochastic Gradient descent

Calculating the negative gradient of the loss function is computationally intensive. Thus, a common method for closely estimating the process is to randomly shuffle training data into equal-sized batches. Each step of the gradient descent algorithm is then performed sequentially according to each batch of data forward-fed through the network. This is known as stochastic gradient descent

Adam optimizer

Adam invokes an adaptive learning rate method [14]. Moreover, it computes individual learning rates for different parameters. The authors of Adam collated ideas from two former adaptations of the stochastic gradient descent algorithm; Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients; Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight.

Adam uses estimations of first moment (mean) as per the RMSProp algorithm, but also makes use of the second moment (uncentered variance) of the gradient to adapt the learning rate for each weight of the neural network. The N-th moment of a random variable is defined as the expected value of that variable to the power of n. More formally:

$$m_n = E[X^n]$$

Where m is the moment and X is a random variable.

Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient and includes parameters (beta1 and beta2) to control the decay rates of these moving averages.

Testing

The performance of a deep neural network can be scrutinised under several performance measures, namely sensitivity, specificity, accuracy, and F-score. Sensitivity refers to the proportion of true positives correctly identified. For this project, this measure would be the proportion of subjects that were correctly as patient. Specificity refers to true negatives correctly identified, which

would be the inverse of the former, i.e. the proportion of subjects that were predicted as healthy controls and that are true healthy controls. The accuracy of a classifier represents the overall proportion of correct classifications. The statistical significance of this overall accuracy can be tested using parametric tests such as permutation testing, which measures how likely the observed accuracy would be obtained by chance. Metrics such as F-score and balanced accuracy, which consider each group's sample size, are particularly useful in cases where classes are unbalanced. The F-score is a measure that combines precision or positive predictive value and sensitivity. Balanced accuracy, on the other hand, correlates to the average accuracy obtained on either class.

Overfitting and remedial strategies

The use of multiple non-linear functions across deep neural networks leads to highly complex models that attempt to estimate a very large volume of parameters. This can lead to model the learning noise and fluctuations in the training set that are immaterial to the problem of classification. This phenomenon is known as overfitting, and is analogous to that of ML. In neuroimaging, the alleviation of this occurrence is particularly challenging, as there exists a much greater number of data points (voxels) than there are subjects.

Regularization

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. The goal is to reduce variance and increase bias. L1 and L2 are the most common types of regularization, which update the general cost function by adding another term known as the regularization term:

$$Cost\ function\ =\ Loss\ +\ Regularization\ term$$

This leads to a much simpler model, meaning that the likelihood of overfitting significantly lowers. This regularization term differs slightly across L1 and L2.

For L1:

$$cost\ function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

For L2:

$$cost\ function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. In L1, we penalize the absolute value of the weights, which may end up at zero. Hence, it is very useful when trying to compress our model. In L2 regularization, also known as weight decay, the weights decay towards, but never reach, zero.

Another strategy is 'dropout', where the network is defined by the presence of neurons at some probability, *p,* during the training phase, i.e., neurons are left out of the network with a likelihood of 1-*p*. This means that the contribution of dropped-out neurons to the activation of downstream neurons is temporally removed during the forward pass and that any weight updates are not applied to these neurons during backpropagation. The aim of dropout is to extract different sets of features that can independently produce a useful output, thereby allowing higher levels of generalizability.

Data augmentation is the act of generating new training data from existing training data. This is particularly useful in the field of neuroimaging as data is often sparse and must be sourced under strict protocol. Common augmentation techniques are rotations, mirrors, random crops and gaussian noise and blur effects.

Normalization

Normalization is applied during the preparation of data to change the values to use a common scale when the features in the data have different ranges. Namely, voxel intensity for MRI scans. Z-score normalization results in the features being rescaled such that they have the properties of a standard normal distribution, with zero mean and a standard deviation (from the mean) of one. Z-scores are calculated as follows:

$$z = \frac{x - \mu}{\sigma}$$

Where μ is the mean value of the feature and σ is the standard deviation of the feature.

As this is an import pre-processing stage for the input layer, it would be intelligible to assume the effects translate to the hidden units in the network. This is known as batch normalization and reduces the amount that the hidden unit values shift around (covariance shift), increasing the stability of the network. This is achieved by, for the output of a given activation layer, subtracting the batch mean and dividing by the batch standard deviation. Batch normalization also adds two trainable parameters to each layer: gamma (standard deviation) and beta (mean), which allows the optimization function to 'denormalize' the output by computing the inverse calculation. if it aids in minimizing the cost function.

Other issues in neural networks

The dying ReLU problem

The dying ReLU refers to the problem when neurons activated with a ReLU function become inactive and only output 0 for any input [16]. This is most probably caused by the network learning a large negative bias term for its weights. Once a ReLU ends up in this state, it is unlikely to recover, as the function gradient at 0 is also 0, thus the weights will no longer be altered the gradient descent algorithm. One simple solution to exchange the ReLU function for the appropriately name Leaky ReLU function. This is a modification of the function that allows small negative values when the input is less than zero.

The vanishing gradient problem

The vanishing gradient problem outlines the phenomenon of extremely small loss function gradients caused by certain activation functions, like the sigmoid function, squishing a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small. This issue resolved by several network architectures, particularly ResNet.

Summary of artificial neural networks

To summarise, the multilayer perceptron is the elementary predecessor of more complex neural network. It consists three layers of artificial neurons: the input layer, the hidden layer and the output layer. Each neuron receives N+1 values, where N is the number of connections between that neuron and the neurons in the previous layer, and the additional value is the bias b. Each neuron will then calculate the weighted sum of these inputs given the weight vector w and the bias. The output is then passed through an activation function that squashes the value between some range before being transferred to the next layer. Training this network is a recursive process that uses backpropagation to distribute weight adjustments across the network given the network loss. Conclusively, a neural network can be considered mathematically as a large function, such that: the input to the function is the encoded data, e.g. each pixel in a Bitmap image; the output is the prediction of the network, i.e. some level of certainty across each neuron in the output layer; the parameters are the weights and biases of the network.

Feedforward and recurrent neural networks

Neural networks can be partitioned into two architectures: feedforward and feedback (recurrent). In a feedforward neural network, the input signal propagates in a single direction only. However, a recurrent neural network has a looping mechanism that allows results to be carried forward from previous stages. Each neuron has a hidden state, which is a representation of previous

inputs. This allows the network to hold information that may provide context to later iterations. For example, the sentence 'what time is it?' may be fed into a network one word at a time. A recurrent neural network will store each previous word in a hidden state, thus encoding the sentence. This may then be fed into a feedforward layer to classify an intent. This project will invoke the former of the two, as sequential inputs are not contextually dependent.

Convolutional Neural Networks

The focus of this project is to develop a 3D convolutional neural network (CNN) to differentiate patients suffering with Alzheimer's from healthy control data. CNNs are a special type of feedforward neural networks that were initially designed to process images, and as such are biologically inspired by the visual cortex. In addition to the input and output layers, CNNs are comprised of three types of layers: a convolution layer, a pooling layer, and a fully connected layer. In the convolution layer, a high-volume of training data undergoes a series of kernel convolutions to isolate scale- and shift-invariant features to produce a feature map. Each feature map is connected to a fixed set of neurons in a local region of the previous layer known as the receptive field in such a way that the whole image is covered (local connectivity). This allows the network to capture spatial and temporal dependencies in an image. As the layers progress, the features become more complex and distinguished. The pooling stage reduces the number of neurons in the previous layer by performing some function. After a series of convolution and pooling stages, the final output is flattened into a column vector via a fully connected layer and fed through a multi-level perceptron for classification. This sequence provides a better fitting to the dataset, as the number of network parameters is dramatically reduced, without the loss of critical attributes [13].

Convolution layer

In 2D CNNs, small portions of the image (called local receptive fields) are treated as inputs to the lowest layer of the hierarchical structure. One of the most important features of CNNs is that their complex architecture provides a level of invariance to shift, scale and rotation, as the local receptive field allows the neurons or to elementary features, such as oriented edges or corners. This network is primarily comprised of neurons with learnable weights and biases, forming the convolution layer. A kernel of size n is systematically applied across height and width of the input image. At each image section, the dot product between the kernel and that section is calculated and the result is mapped to a 2D matrix. The kernel is a learned image feature, and the output matrix provides a summary of the presence of that feature across the image. This is the process of convolution, which is repeated with many different kernels to form a convolution layer with a stack of filtered images. Essentially, a convolution layer will transform a single image to a stack of filtered images. Figure 2 displays this process.

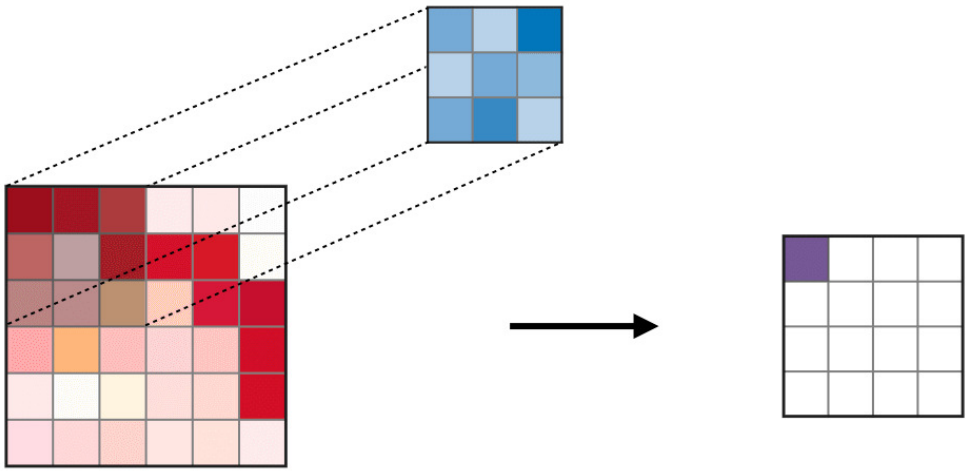

Figure 2: Convolution layer [18]

A convolution layer in a 3D CNN is not dissimilar from that of a 2D model. 3D convolutions yield a volumetric space by applying a 3D filter to the dataset that manoeuvres tridirectionally (x, y, z) to account for the extra spatial dimension. Employing 3D convolutions into our network resolves issues surrounding certain pre-processing steps, such as image slicing. However, these are much more computationally intensive processes.

Pooling layer

The pooling layer is used to down-sample the feature map procured from convolution layer, while further desensitizing the network to its spatial positioning and orientation. There are many pooling techniques, but a popular choice is max pooling. Both a window size and a stride length are selected as hyperparameters for the network, and the window is walked across a filtered image one stride at a time. At each image section, the maximum value within the window is selected and mapped to another 2D matrix. This process is repeated for all filtered images. Again, the methods applied to 3D pooling layer does not hugely differ from its lower dimensional counterpart. The pooling function is exerted over chunks of each 3D feature map to supply a down-sampled volume before being passed to the next layer. Figure 3 displays this process.

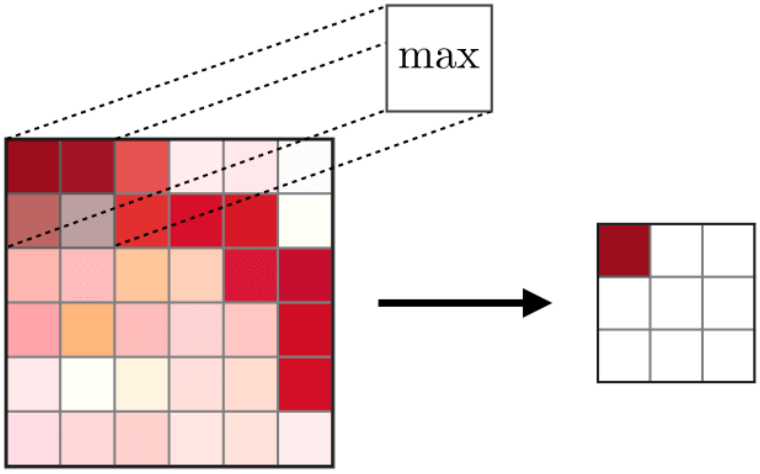

Figure 3: Max pooling layer [18]

Fully connected layer

Fully connected layers are similar to the hidden layers from the conventional MLP, where the neurons are connected to all neurons from the previous layer. Here, the filtered and pooled image is flattened to a single vector. When training the network, every element in this vector receives a connection weight to the output layer that determines how much it correlates to each class. For example, the connection between a feature representing wings and the class 'bird' in the output layer is much stronger than that of a feature representing four legs. When an unseen image is passed through the network, the weighted average of the vector is calculated for each class, which is taken as the model's prediction. As the output of this layer is analogous to its input, fully connected layers may be aligned sequentially for intermediate categorisation. These intermediate classes are often called 'hidden units' and can improve the accuracy of the model by further breaking down the structure of the image. Figure 4 displays this process.

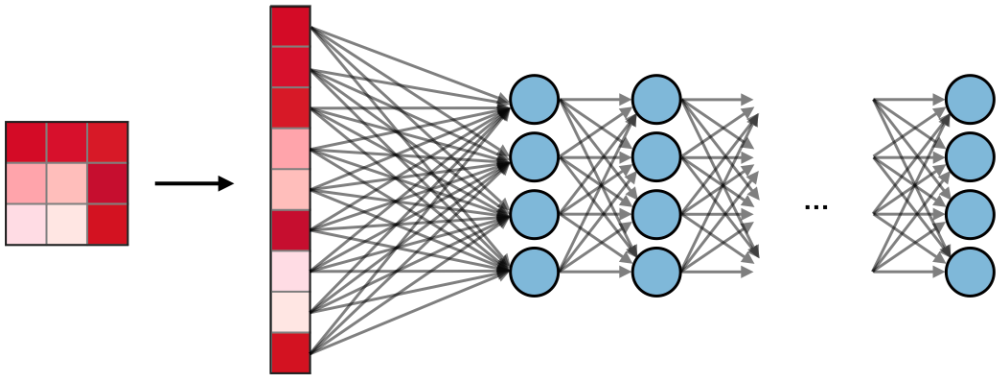

Figure 4: Fully connected layer [18]

Backpropagation in CNNs

The topology of CNNs utilizes spatial relationships to reduce the number of parameters that must be learned, improving upon general feed-forward backpropagation training. Equation 1 demonstrates how the gradient component for a given weight is calculated in the backpropagation step, where E is error function, y is the neuron $N_{i,j}$, x is the input, l represents layer numbers, w is filter weight with a and b indices, N is the number of neurons in a given layer, and m is the filter size:

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m}\frac{\partial E}{\partial x_{ij}^{l}}\frac{\partial x_{ij}^{l}}{\partial w_{ab}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m}\frac{\partial E}{\partial x_{ij}^{l}}y_{(i+a)(j+b)}^{l-1}$$

[19]

Equation 2 describes the backpropagation error for the previous layer using the chain rule. This equation is similar to the convolution definition, where $x_{(i+a)(j+b)}$ is replaced by $x_{(i-a)(j-b)}$. It demonstrates the backpropagation results in convolution while the weights are rotated. The rotation of the weights derives from a delta error in the convolutional neural network:

$$\frac{\partial E}{\partial y_{ij}^{l-1}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1}\frac{\partial E}{\partial x_{(i-a)(j-b)}^{l}}\frac{\partial x_{(i-a)(j-b)}^{l}}{\partial y_{ij}^{l-1}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1}\frac{\partial E}{\partial x_{(i-a)(j-b)}^{l}}w_{ab}$$

[19]

CNNs in neuroimaging

2D convolutional neural networks

One of the first documents detailing the use of DL in an AD study was carried out by A. Gupta, M. Ayhan, and A. Maida [7]. Here, the authors identified that natural images and brain imaging methods share similar, and therefore interchangeable, low-level features (e.g. lines and corners). As well as this, they outlined that natural images are more abundant, thus derived that a method of transfer learning could be employed. Transfer learning is the act of pretraining a network on one set of data to extrapolate for a similar set of data. Therefore, the authors pretrained a sparse autoencoder to learn features from natural images, which were then applied to structural MRI data via a CNN, achieving a classification accuracy of 94.7% for AD versus controls, 86.4% for MCI versus controls and 88.1% for AD versus MCI. Consistent with the authors' hypothesis, this method outperformed the one where the learned features were extracted from the neuroimaging data (93.8%, 83.3% and 86.3% for the same comparisons, respectively).

3D convolutional neural networks

Under the same presuppositions as Gupta, Ayhan, and Maida, A. Payan and G. Montana found comparable classification accuracies using features that were learned from the structural MRI data itself using a sparse autoencoder with a 3D CNN . Furthering the previous work, Payan and Montana pretrained the sparse autoencoder with general MRI data from various neurological patients, and fine-tuned the network with those suffering from AD. The advantage of taking this approach is that not only was the transfer learning more task-specific, 3D data contains more useful patterns for classification. Payan and Montana compared the results of their 2D CNN implementation against their 3D implementation, and found that their 3D model generally outperformed the 2D alternative (AD vs. HC (2D/3D) = 95.4%/95.4%; AD vs. MCI (2D/3D) = 82.2%/86.8%; MCI vs. HC (2D/3D) = 90.1%/92.1%).

3. Project Specifics

Pre-processing in Magnetic Resonance Imaging

Magnetic Resonance Imaging (MRI) is an imaging modality used in medicine to produce detailed images of human anatomy. First introduced to the medical industry in 1977, the MRI scanner has revolutionised clinical diagnosis and prognosis of patients suffering from neurological, abdominal, and many other variations of illness. MRI scans are riddled with many issues that force robust pre-processing mechanisms to be implemented in any analysis pipeline. For most neuroimaging pipelines proposing solutions to challenging ML problems, this is a vital part of the system. There exists many normalization and redundant information removal technique, but this small segment highlights those of interest to the project.

A frequent issue is with MRI images is bias field corruption. The bias field is a low-frequency and very smooth signal that corrupts the images, especially those produced by old MRI machines [20] . Further pre-processing algorithms, such as registration, brain extraction and segmentation, require a bias-corrected image to perform correctly.

Another issue is the variety of imaging modalities and protocols followed during MRI acquisition. Thus, it is nearly always desirable for the images to be normalized to some standard space. Image registration is the process of transforming different sets of data into one coordinate system. For MRIs, a common coordinate system is that defined by the Montreal Neurological Institute and Hospital. The MNI space defines the boundaries around the brain, expressed in millimetres, from a set origin [21]. For comparative analysis, it is important that all images undergo this procedure.

Many neuroimaging pipelines are also inclined to extract a more particular region of interest from the volume as to remove redundant information. A popular technique to do so is brain extraction. Intuitively, brain extraction is the process of removing skull and neck voxels from a 3D MRI volume. This is a costly process that requires both time and computationally intensive calculations. Furthering this, clinicians often spatially smooth the brains using a range of Gaussian kernels. Spatial smoothing is the act of averaging data points with neighbours. This has a low pass filtering effect, meaning that high frequencies of the signal are removed from the data while enhancing low frequencies [22]. The result is that sharp edges of the images are blurred and spatial correlation within the data is more pronounced. Moreover, this reduces the signal to noise ratio within the image. With relevance to the project's aim, this could reduce the risk of the convolutional neural network overfitting to the training data due to this noise reduction effect.

The ADNI dataset

The Alzheimer's Disease Neuroimaging Initiative (ADNI) is a multicentre study designed to develop clinical, imaging, genetic, and biochemical biomarkers for the early detection and tracking of Alzheimer's disease [23]. Research began in 2004 under the leadership of Dr. Michael Weiner, funded as a private-public partnership, and was split into four phases: ADNI-1, ADNI-GO, ADNI-2 and ADNI-3. The initial five-year study was extended by two years through a Grand Opportunities grant and furthered again in both 2011 and 2016. This dataset that will be used to train and test the deep convolutional neural network.

Software and libraries

Google Colab

Google Colab is a free cloud-based service that implements a high-performance interactive Python development environment [24]. Following dependencies, such as Tensorflow, come pre-installed on each notebook, which may be run for twelve hours at a time before being 'recycled' and local variables are deleted. A Google Drive can be mounted to each notebook for both permanent package installation and access to other files.

FSL

FSL is a comprehensive, standalone library of analysis tools for FMRI, MRI and DTI brain imaging data [25]. Tools can be run either directly from a command line interface, or through the FSL GUI. FSL provides functions for bias correction, image registration, brain extraction, segmentation, and other necessary pre-processing stages.

Python

Python is a programming language natively installed on Windows, Linux, and MAC OS [26]. It is one of the most popular language in the AI community, with an incredible ease of use and indicative documentation to follow. The following libraries will be used alongside Python to develop the system.

Deep Learning Tool Kit (DLTK)

The deep learning tool kit is a compilation of various popular neuroimaging packages that provides a more abstracted frontend solution to the individual package installations [27]. These packages include SimpleITK and Nibabel, both reputable tools for neuroimaging analysis.

Tensorflow

Tensorflow is an end-to-end opensource machine learning framework that hosts an eco-system of tools used to develop and train models. Tensorflow's strong community makes this a an extremely viable framework to base the solution upon [28].

Keras

Keras is an open-source API capable of running on-top of Tensorflow that provides a high-level framework for NN model development [29]. Keras can run seamlessly on available GPUs. The Sequential class was of particular interest for this project.

Nipype

Nipype is a python library that interfaces several popular neuroimaging analysis tools, including FSL [30]. Nodes are stringed together in flows to form an end-to-end pipeline. Each node is representative of an interfaced stdout command, which is called when the flow is run. This will form the basis of the pre-processing module implemented in this project.

Approach

This section details the approach taken for suitably constructing a viable solution to the problem.

1. Data Handling

Acquisition

The dataset for this application consisted of subjects from the ADNI study. It was pre-reviewed that the set acquired should be reasonably balanced. That is, no single class should dominate the overall population of the dataset. However, a slightly larger control set was inevitable. It was also noted that the information should be collected in line with the ADNI data usage agreement. After reviewing the available data for each phase of the study closely, subjects were to be taken from the baseline phases of the ADNI-1 and ADNI-2 phases. These both produced the most suitable sets of subjects, with minor disparities in MRI scanning modalities.

The ADNI advanced search interface was used to filter and select subjects accordingly. Firstly, all control subjects were filtered, selected and added to a collection named 'ADNI-1-2_CN', and a similar search was carried out for the AD subjects, whom were saved to a collection named 'ADNI-1-2_AD'. This method allowed all subjects to be readily compiled according to their class, precluding the chance of cross-contamination between datasets. The folders were then downloaded, and all files were extracted to obtain the relevant files and rename them to their complementary subject numbers. From this, two folders – 'cn' and 'ad' – containing [n] and [n] NifTi files were produced which would be used as inputs for subsequent stages.

Pre-processing

Multiple protocols were reviewed from several software libraries that would achieve the desired results for this module, but it was settled that brain extraction should be analogous to FSL's 'fsl_anat' command. Replicating this command in Python would allow the solution to be implemented across a single interface. As per the chosen command, all subjects were to be reoriented, bias-corrected, registered to MNI152 space, brain extracted and smoothed. From here, the network could be trained on five distinct datasets to review which gives better results during the classification stage. Those datasets are as follows: registered whole-head MRI scans, registered skull-stripped MRI scans, and MRI scans that were registered, skull-stripped and smoothed with three separate Gaussian kernels.

It is important that module yields suitable results and does not introduce further issues. That is, when brain extraction takes place, residual neck or skull voxels may remain. This would leave outliers in the dataset, which would be an issue when training the network. Hence, all pre-processed

images should be scrutinised by eye before further classification stages take place. This could be achieved via fsl's GUI 'fsleyes'.

Train test split

It was also decided that the dataset should have an appropriate split for training, validation and testing data. It was important for the split to leave enough training data, as it was preconceived that there would already be a shortage of scans available, especially those from the AD class. To fall in line with these preconditions, it was settled that the data should split should be 70%, 15%, 15% train, validation, test material, respectively. That is, 70% of the data should be labelled and used to train the model, 15% could be used to assess the progression of the model during its training phas, then the model would make predictions on the remaining unseen 15%.

Compatibility

To make the network run as seamlessly as possible, it was important to ensure that the flow of data was as fast as possible. However, neuroimage files, such as NifTi images, can be very large, thus, loading all data to memory was not seen as an appropriate method for handling the data. Therefore, it was decided that both the train and test datasets should be serialized to disk for fast read / write access. Hence, it was found to be appropriate that two TFRecord datasets should first be created from the raw data before being passed to the network.

2. Network Construction

Environment

The development environment for the network should meet satisfactory conditions such that the solution is able to run without hindrance due to performance issues. Thus, it was decided that the network should be implemented within a Google Colab notebook. This would enable GPU access for the Keras and in turn considerably improve performance.

Architecture

Basic structure

The structural design of the network was an important, multifaceted component of the pipeline that was crucial for correct classification of subjects. It was determined that a basic structure should first be introduced, before affirming a more particular solution to the issue. This consisted of five 3D convolution and max pooling pairs, followed by two dense layers. The input to the network was set to 80x100x80, as this is the size of the MRIs. The number of filters for each convolution was set to 32, 64, 128, 128, and 256, respectively, to ensure that the network could build a high enough abstraction of the scans. Each filter in each convolution layer was set to size 3x3x3, with a

stride length of 1 in each direction. The size of each pool in each pooling layer was set to 2x2x2, with a stride length 2 2 in each direction, and the padding was set to 'SAME'. This padding was chosen as it was important to retain as much information from each scan as possible. The neurons for the dense layers were set to 256 and 1, respectively. The process of choosing the number of neurons in the first dense layer was trivial; it could be altered for optimality. However, as the input data was labelled binarily ([0] or [1]) over one-hot encoding ([0, 1] or [1, 0], then it was necessary for the second layer to have only a single neuron with sigmoidal activation.

The activation for each convolution layer was set to ReLu, which was chosen over other activation functions as it does not saturate; that is, the gradient it always high if the neuron activates. Max pooling was selected over average pooling or other down sampling techniques as it extracts more verbose features. This was desirable as MRI images are grey-scale and all pixels surrounding the head are zeroed. Therefore, max pooling can define outstanding features more rigidly.

Loss and optimization functions were chosen that best fit a solution to a binary classification problem. Binary cross-entropy appeared most suitable as it would minimize the distance between the two probability distributions – predicted and actual. Adam was chosen as the optimization function, which was favoured over classic stochastic gradient descent as it has been shown to outperform in the field of binary classification on many occasions. Adam also yields extra hyperparameters that could be tuned to improve performance.

Training the network

The training phase of the network construction should be carried out such to minimise the number of train-test iterations. This would be a very time-consuming approach to building the model. The following procedure was administered for all training sets.

Capacity evaluation (strategic overfitting)

Firstly, the network was scrutinised with a capacity evaluation. All hyperparameters for the optimization function were kept default, and the all regularization and normalization methods were omitted. The model was trained for ten epochs and as expected, the model overfitted to all training sets. This showed that the model was deep enough to learn the features of the training data, and it did so without issue. It was easily established that the model was overfitting as the training loss and accuracy were respectably extremely low and high in comparison to the validation loss and accuracy. Hence, the model had learned all fluctuations of the training data such that it struggled to generalize for the unseen data.

Generalization

Next, the first strategy to rectify the generalization error was introduced – batch normalization. A batch normalization layer was implemented in between each convolution and max pooling layer and the network was retrained for a further ten epochs so the effect could be monitored. It was instantly observed that this had aided in solving the issue. The network took longer to successfully map training inputs to target values but achieved more similar results between both datasets before overfitting once again.

Following this, a dropout layer was administered after the first dense layer with default parameters. Again, the network was retrained for ten epochs so the effect could be recorded. Similar results were observed as per the introduction of the batch normalization layer, yet overfitting persisted. Thus, the final remedy was installed in the form of kernel regularization. Both l1 and l2 were trialled for this stage, but l2 achieved much better effects without causing the model to underfit to the training data due to network sparsity introduced by l1.

After introducing these additional features, the network was now significantly better at generalizing, but the model struggled to fit to the training data. Thus, it was hypothesised that the network may struggle with the 'dying ReLU' issue. Therefore, the regular ReLU activation was subsided and replaced with the Leaky ReLU activation function. The network was again retrained for a further ten epochs, and it was observed that this remedied the hindrance. The operation of the network was now at a point where further hyperparameters could be altered to improve performance.

Selecting hyperparameters

With a more complex model in place, hyperparameters for the network could be adapted to the new structure. It was established that there should be some organization to the alteration of these. Thus, various ranges of values were self-proposed; namely the number of epochs, batch size, learning rate, and dropout rate for the dropout layers. The number of epochs would be trialled within the range of (formatted as [start : increment : end]) [10 : 10 : 100], the batch size within the range of [4 : 4 : 32], the learning within the range [0.1 : 0.05 : 0.001] and the dropout rate within the range [0.2 : 0.1 : 0.5].

While trialling these values, it became immediately apparent that the onboard memory for the Colab GPU could not sustain a batch size greater than four for this model. Thus, this left no choice but to keep this as the default. Leading on from this, it was recognised that the learning rate should also be small. This is a general rule for neural networks, as the smaller the batch size the closer to a stochastic gradient descent. That is, there is much less 'confidence' in our descent towards the

minimum of the cost function as the overall cost is more approximated. This led to a much lengthier training procedure. The dropout rate was scaled between the prementioned range, but a value greater than 0.2 caused the model to severely underfit to the training data.

Solution and Implementation

This section details the replicable solution to the problem highlighted throughout the previous sections of this paper. The first segment outlines the data handling methods and supplementary code for review or replication of each stage. The second segment highlights the model itself as well as the network's operation and supplementary code for review or replication of each stage.

1. Data Handling

Directory structure, file handling and demography

Directory tree

As proclaimed, it was vital that both groups of subjects were separated immediately from the acquisition phase. To do this effectively, the access of directories must be consistent across the application. Thus, a simple structure was devised that would be suitable for all modules to access. Under each class directory, there contains four distinct folders. The first of which contains the raw files extracted from the zip file downloaded from the Loni repository. After doing so it was immediately apparent that this file structure would grow to several gigabytes in size. Therefore, the GNU gzip command was utilised for simple lossless file compression. The following three directories are data sinks for the Nipype pipeline implemented in the pre-processing stage of operation. The meta subdirectory is a folder introduced for the complex file tree made by Nipype to store intermediate files used during execution. For simplicity, this file structure will be ignored.

```
data/
├── cn/
│   ├── raw
│   ├── registered
│   ├── skull_stripped
│   ├── segmented
│   └── meta
├── ad/
│   ├── raw
│   ├── registered
│   ├── skull_stripped
│   ├── segmented
│   └── meta
```

Demography

For a more concise and robust implementation of latter processes, as well as a more general representation of the dataset population, a simple CSV file was created to hold the subject number and corresponding class for each MRI file. The code in figure 5 uses the python os class to list all files previously extracted to the raw folder of each group. The csv class is then used to open a file named 'demographics.csv' and write the subject with the corresponding class label (0 for control, 1 for AD) to a new line.

```
DIR = './data'

def write_csv():

    cn_path = DIR + '/cn/raw'
    ad_path = DIR + '/ad/raw'

    cn = os.listdir(cn_path)
    ad = os.listdir(ad_path)

    with open('{}/demographics.csv'.format(DIR), 'w', newline='') as f:
        for s in cn:
            subj_id = str(s.split('.')[0])
            writer = csv.writer(f, delimiter=',')
            writer.writerow([subj_id, '0'])

      for s in ad:
            subj_id = str(s.split('.')[0])
            writer = csv.writer(f, delimiter=',')
            writer.writerow([subj_id, '1'])

    f.close()
```

Figure 5: CSV Write function

Pre-processing pipeline

General structure

The pipeline implemented for the pre-processing section of this solution iss split across two modules. The flow.py module contains a series of functions that are imported to the pipeline.py module, which connected the flows accordingly. The method was implemented to abstract away from the complex operations performed during each pre-processing stage. All following code was taken from the pipeline.py module to reduce complexity of this section.

Data grabbing

First, the pre-processing module collects the raw data from the data directory. The datasource node handles the iterative requirement for the pipeline. That is, it will iterate through the subject list and pass each subject id to the datagrabber node. This node uses the subject id to pull the corresponding files from the raw data directory based on the template given. Figure 6 shows the creation of this node.

```
DIR = "./data"

subjects = pd.read_csv(
    '{}/demographics.csv'.format(DIR),
    dtype=object,
    keep_default_na=False,
    na_values=[]).iloc[:, 0].values

# Get data
datasource = pe.Node(interface=IdentityInterface(fields=['subject_id']),
                     name="datasource")
datasource.iterables = [('subject_id', subjects)]

datagrabber = pe.Node(interface=nio.DataGrabber(
                     infields=['subject_id'], outfields=['T1']), name="datagrabber")
datagrabber.inputs.template = '{}/raw/*/%s.nii'.format(DIR)
datagrabber.inputs.base_directory = os.getcwd()
datagrabber.inputs.sort_filelist = True
```

Figure 6: Data grabbing

Figure 7 shows the connection made between the datasource iterable and the datagrabber node.

```
# Connect io
preprocessing.connect(datasource, "subject_id", datagrabber, "subject_id")
```

Figure 7: Datagrasource / Datagrabbing connection

Data sinking

To handle the output of each stage, a data sink node was created. This allowed for important files to be 'sunk' to the file structure created previously. This ridded the issue of siphoning through Nipype's complex internal files structure to find the results necessary for later stages. Figure 8 shows the creation of this node.

```
# Create output folder for important outputs
datasink = pe.Node(interface=nio.DataSink(),
name="datasink")
datasink.inputs.container = "data"
datasink.inputs.base_directory = os.getcwd()
datasink.inputs.regexp_substitutions = [('_subject_id_.*/', '')]
```

Figure 8: Datasink

Figure 9 shows the connection between each relevant node and the data sink.

```
# Sink nodes
preprocessing.connect(brain_extraction, "t1_to_mni_lin.out_file", datasink, 'registered.@')
preprocessing.connect(brain_extraction, "brain_to_mni_lin.out_file", datasink, 'registered_skull_stripped.@')
preprocessing.connect(smoothing, "smoothing_s2.out_file", datasink, "registered_skull_stripped_smoothed.@)
```

Figure 9: Datasink connections

Bias correction and registration

Firstly, the pre-processing module takes the raw MRI images and reorients them, such that they appear 'the same way round'. Next, each image is corrected for the bias field signal. All images are then linearly registered to MNI standard space using the MNI152 template supplied by FSL as the reference image. Each image is then non-linearly registered to standard space with the same template, along with the linear registration matrix output from the previous registration as the affine transformation matrix. Figure 10 shows the process as seen by the pipeline.py module.

```
# Bias correction and registration
preprocessing.connect(datagrabber, "T1", bias_correction, "T1std.in_file")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "t1_to_mni_lin.in_file")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "t1_to_mni_nonlin.in_file")
```

Figure 10: Bias correction and registration

Brain extraction

Following this, each image is brain-extracted using the field coefficient file procured from the non-linear registration. The field is inversed and applied to the MNI152 brain mask, which is then applied to the original bias-corrected image to achieve a bias-corrected, brain extracted image. The FSL BET tool was also trialled for this operation, but the latter method substantially outperformed this. Figure 11 shows the process as seen by the pipeline.py module.

```
# Brain extraction
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "mni_to_t1_nonlin_field.reference")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "brain_mask.ref_file")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "brain.in_file")
```

Figure 11: Brain extraction

Full pipeline

Figure 12 shows the full pipeline implemented in the pipeline.py module.

```
# Initiate workflow
preprocessing = pe.Workflow(name="preprocessing", base_dir="meta")

# Import workflows
bias_correction = flows.bias_correction()
brain_extraction = flows.brain_extraction()

# Connect io
preprocessing.connect(datasource, "subject_id", datagrabber, "subject_id")

# Bias correction and registration
preprocessing.connect(datagrabber, "T1", bias_correction, "T1std.in_file")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "t1_to_mni_lin.in_file")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "t1_to_mni_nonlin.in_file")

# Brain extraction
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "mni_to_t1_nonlin_field.reference")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "brain_mask.ref_file")
preprocessing.connect(bias_correction, "biascorr.restored_image", brain_extraction, "brain.in_file")

# Sink nodes
preprocessing.connect(brain_extraction, "t1_to_mni_lin.out_file", datasink, 'registered.@')
preprocessing.connect(brain_extraction, "brain_to_mni_lin.out_file", datasink, 'skull_stripped.@')
```

Figure 12: Full pre-processing pipeline

Dataset construction

Train test split

The train-test split was generated using a combination of numpy and pandas operations function. The train, validation, test split was 70%, 15%, 15%, respectively. The datasets were then marshalled and serialized to their corresponding files. Figure 13 shows this.

```
[9]  subjects = pd.read_csv(
         '{}/demographics.csv'.format(img_dir),
         dtype=object,
         keep_default_na=False,
         na_values=[])

     # Path to save the TFRecords file
     train_filename = img_dir + '/train.tfrecords'
     validation_filename = img_dir + '/validation.tfrecords'
     test_filename =  img_dir + '/test.tfrecords'

     # Generate train test split
     train_split, validation_split, test_split = np.split(subjects.sample(frac=1), [int(.7*len(subjects)), int(.85*len(subjects))])

     train_split = train_split.values
     validation_split = validation_split.values
     test_split = test_split.values
```

Figure 13: Train test split

Loading data

Image loading was handled by the load_img function. This was called upon during later the marshalling stage of the dataset creation. The subject id and label were extracted from the metadata and the corresponding file name was generated. The image was loaded as a simpleitk object and converted to a numpy array. The image was then cropped, normalised using the DLTK normalise_zero_one function, reshaped to fit a dummy dimension for the channels and returned alongside the image label. Figure 14 shows this.

```
[74] def load_img(meta_data):
         x = []

         subject_id = meta_data[0]
         label = meta_data[1]

         if label == '0':
           t1_fn = img_dir + '/cn/{}.nii.gz'.format(subject_id)
         else:
           t1_fn = img_dir + '/ad/{}.nii.gz'.format(subject_id)

         # Read the .nii image containing a brain volume with SimpleITK and get
         # the numpy array:
         sitk_t1 = sitk.ReadImage(t1_fn)
         t1 = sitk.GetArrayFromImage(sitk_t1)

         t1 = crop_image(t1, (DEPTH, HEIGHT, WIDTH))

         # Normalise the image to zero - one range
         t1 = normalise_zero_one(t1)

         # Create a 4D Tensor with a dummy dimension for channels
         t1 = t1[..., np.newaxis]

         x = t1

         y = np.int32(label)

         return np.array(x), y
```

Figure 14: Load image function

Image cropping

Image volumes were cropped using a bounding box method sourced from stack overflow. **Figure** shows the implementation of this function. Figure 15 shows this.

```
[ ]  def crop_image(img, bounding):
         start = tuple(map(lambda a, da: a//2-da//2, img.shape, bounding))
         end = tuple(map(operator.add, start, bounding))
         slices = tuple(map(slice, start, end))
         return img[slices]
```

Figure 15: Image cropping[1]

Marshalling

Data marshalling was handled by the create_data_record function. Here, an output file name was received as argument, along with the set of addresses corresponding to the relevant dataset and the explicit string that detailed that mode of operation ('train' or 'test'). A TFRecordWriter object is instantiated and the filenames are cycled through. The load_image function was is called and the img, label pair is unpacked upon return. A feature is created and used to create an example protocol buffer, and the writer object is used to write the serialized img-label pair to the appropriate file. This function was later called after all filenames had been gathered and the train-test split was generated. Figure 16 shows the marshalling of the serialized files.

```
[76] def create_data_record(filename, split, mode):

       # open the TFRecords file
       writer = tf.io.TFRecordWriter(filename)

       i, j = 0, 0

       for meta_data in split:

           # Load the image
           img, label = load_img(meta_data)

           if label == 0:
             i += 1
           else:
             j += 1

           # Create a feature
           feature = {'label': _int64_feature(label),
                      'image': _float_feature(img.ravel())}

           # Create an example protocol buffer
           example = tf.train.Example(features=tf.train.Features(feature=feature))

           # Serialize to string and write on the file
           writer.write(example.SerializeToString())

       writer.close()
       sys.stdout.flush()

       print('{} items:\n\
      control: {}\n\
      AD: {}\n\
      total:{}\n'.format(mode, i, j, i+j))
```

Figure 16: Data marshalling

[1] https://stackoverflow.com/questions/39382412/crop-center-portion-of-a-numpy-image

Unmarshalling

After the TFRecord files have been written, the data could be deserialized and accessed at much higher speeds, without the need to load all data to memory. Two distinct TFRecord dataset objects are instantiated and the respective file names are passed as arguments. The function returns a numpy array and integer for the MRI data and label, respectively. Data is augmented, shuffled, and batched for the training data, whereas the augmentation and shuffle stages are excluded from the test data. The datasets were now in a compatible format for input to the network. Figure 17 shows the unmarshalling of the serialized files.

```
[11] def decode(serialized_example):
        # Decode examples stored in TFRecord
        features = tf.io.parse_single_example(
            serialized_example,
            features={'image': tf.io.FixedLenFeature([DEPTH, HEIGHT, WIDTH, 1], tf.float32),
                      'label': tf.io.FixedLenFeature([], tf.int64)})

        return features['image'], features['label']

    train_dataset = tf.data.TFRecordDataset(train_filename).map(decode).map(randomflip).shuffle(buffer_size=512).repeat(None).batch(BATCH_SIZE).prefetch(1)
    validation_dataset = tf.data.TFRecordDataset(validation_filename).map(decode).repeat(1).batch(BATCH_SIZE).prefetch(1)
    test_dataset = tf.data.TFRecordDataset(test_filename).map(decode).repeat(1).batch(BATCH_SIZE).prefetch(1)
```

Figure 17: Data unmarshalling

2. The Network

Input layer

The input layer takes as input a Tensorflow dataset object that yields two numpy arrays; namely, a data batch and corresponding labels. The data batch array is of shape (batch size, depth, height, width, channels), and the labels array is of shape (number of subjects in batch). Figure 18 shows this.

```
model.add(Conv3D(32, kernel_size=(3, 3, 3), strides=(1), activation='linear', kernel_regularizer=None, kernel_initializer=init, input_shape=(DEPTH, HEIGHT, WIDTH, NUM_CHANNELS)))
model.add(LeakyReLU())
model.add(MaxPooling3D(pool_size=(2, 2, 2), strides=(2), padding='SAME'))
```

Figure 18: Input layer

Feature selection

During the feature selection phase, the input data faces a series of five 3D convolution and pooling stages to attain some interpretation of the image. As these stages progress, the model captures higher-level features. After the input layer, these features may be simple lines, then at the next stage they may progress to edges and so-forth. For patients suffering with AD, higher-level features may have been in the form of anatomical lesions caused by the disease. The information distilled from this stage was aggregated and flattened into a single 1D vector before entering the dense (fully connected) layers for classification.

Figure 19: Feature selection stage

Classification

The classification segment of the neural network comprises two dense layers, with the second being the output layer. During the training stage the batch loss is calculated and backpropagated through the network, which is displayed alongside any chosen metrics for the model. After each epoch, the network is validated with the test data which gives insight to the progression of the model's fit. Figure shows the code for this.

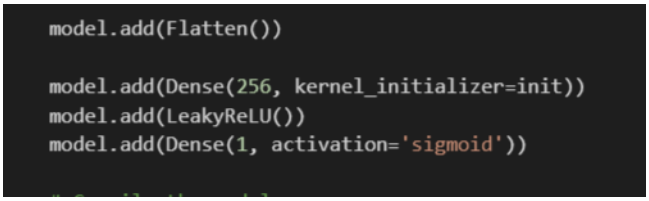

Figure 20: Classification stage

Compiling the model

The model is then compiled with the previously agreed optimization and loss functions. The learning rate is set with the global learning rate variable. The accuracy metric is also requested to be displayed when running the model. shows the code for this Figure 20 shows this.

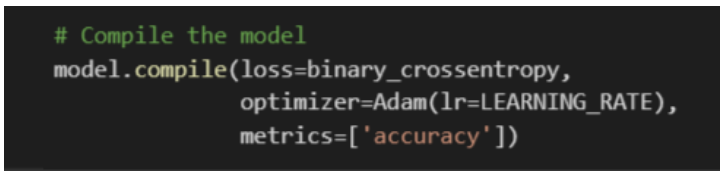

Figure 21: Model compling

Results and Evaluation

The aim of the project has been achieved. A convolutional neural network that can distinguish patients suffering with AD from healthy control subjects has been developed and the effect of various approaches to pre-processing has been discussed. Regions of interest were extracted from the input MRIs via the multiple convolution layers and mapped to class labels. These learned patterns were used to make inferences on an individual level, as a clinician would. Many models have been developed for the ADNI dataset from those much more knowledgeable on the topic of neuroimaging analysis, thus direct network performance comparison is rather trivial. However, the comparison of network performance from different pre-processing techniques is scarcely documented for basic models (particularly those that do not employ pre-training or transfer learning). It is therefore worth nothing that this project did contribute this information towards the ML and neuroimaging community, which could be furthered for more complex analysis from piers more well-versed on the subject.

1. Evaluation Metrics and Methods

For each dataset, a single different performance metrics were used to assess the strength of the network. This selection was sufficient to suitably assess the performance of the model given each dataset. Accuracy is a measure of the number of correctly predicted labels of an MRI out of all samples. The validity of accuracy as a performance metric is dubious when the dataset is imbalanced. That is, the dataset has a disproportionate population of a single class. Given a dataset with 90% of one class to 10% of another class, the model simply learns that if it guesses the label representative of the larger class all the time, it will achieve at least 90% accuracy.

As our dataset was reasonably balanced, it was decided that a confusion matrix would not be necessary. A confusion matrix is an n x n matrix, where n is the number of classes, used to summarise the performance of a classification model. A confusion matrix shows the prediction for a sample against its true label and specifies where erroneous classifications are made. It was expected that the model would not be perfect due to three reasons: lack of training data, insufficient knowledge of robust pre-processing methods and possible ignorance towards this, and insufficient resources to build a more complex model.

2. Comparison of Results

Testing

As previously explained, this project aimed to discuss the effect of different approaches to neuroimage pre-processing when developing a classification pipeline. A total of five datasets were used to train and test the network, namely: whole-head and affine registered MRIs, skull-stripped and affine registered MRIs, and MRIs that were skull-stripped, affine registered and smoothed using a Gaussian smoothing kernel with sigma = 2. It was expected that the removal of redundant information – namely skull and neck voxels – would aid in both increasing the network's throughput as well as improve accuracy in the classification of the subject. It was uncertain whether the smoothing would increase or decrease the accuracy of the model, as spatially smoothing each of the images would improve the signal-to-noise ratio (SNR), but at the expense of image resolution. However, it was hypothesised that due to the improvement of the SNR ratio, spatial smoothing could help to prevent the model from overfitting

Comparison of pre-processing techniques				
	Accuracy (%)		Loss	
	Train	Test	Train	Test
Registered	96.6	84.4	0.035	1.242
Registered and skull-stripped	96.8	85.0	0.034	1.171
Registered, skull-stripped and smoothed	97.6	87.1	0.032	1.303

Figure 22: Table of results

The above table shows that, as hypothesised, the more robust pre-processing pipeline yielded better results than other approaches.

Conclusion

As this project concludes, it is worth revisiting the aims and critically evaluating whether they were met. To reiterate, the aim of this project was to develop a convolutional neural network that could distinguish patients suffering with AD from normal healthy control subjects. The intention was not to present a clinically valid and readily available model, but rather to explore various approaches to doing so and present a proof of concept that could be reviewed and built upon. It was immediately recognised that the validity of the pre-processing module would be the crux of the project, thus this was a major focus in the development of the pipeline. The various outputs from the module allowed us to explore to some extent the importance for rigid pre-processing procedures for high accuracy classification. Future work conducted by those more knowledgeable on neuroimaging analysis could take note of the results presented by this project before implementing their own strategies for developing a classification pipeline. Moreover, these results are not limited to classification of only AD patients but may also be extrapolated and applied to the detection of other anatomical disorders detectable by MRI scans. However, some of the pre-processing measures discussed are specific only to neuroanatomical MRI scans, more specifically the skull-stripping section. On the other hand, bias field correction, registration and smoothing all translate to the other anatomical scans.

The results suggested that the throughput of a CNN developed for neurological assessment is conclusively linked to the amount of pre-processing performed to rid redundant information from the volumes prior to entering the network. Not only this, but classification accuracy is severely improved by doing so. However, it was also shown that a relatively successful pipeline could be built with minimal processing performed on each volume. Furthermore, this paper indicates that neural networks are mostly limited by the practitioner's competency when handling the specified data modality. The processes undergone throughout this project were conducted under no prior knowledge of neuroimaging data. A clinician well-versed in this subject may have performed better even with lesser knowledge of machine learning.

The black-box approach to training a deep-learning model learning has proven to be both advantageous as well as a draw back. Programmatically, the construction of the model is concise and easily replicable due to the highly abstracted APIs used (Keras and Tensorflow). However, there is an inherent inability to assess where the model struggles to learn. Thus, when a model does not perform well, there are few approaches that yield better results than trial and error when attempting to rectify the issue.

As with many deep learning tasks, the results are extremely limited by available computational resources. Despite Google Colab enabling GPU usage, the shortage of on-board memory was an issue when training the network. Due to the nature of a neural network' s learning algorithm, the restriction on the hyperparameters imposed by this issue undoubtedly influenced the outcome of the solution. Thus, it is possible that stronger results may be obtained for those with access to a desktop computer with higher specification, or even access to a supercomputer cluster.

Reflection

The past few months taken to complete this project have been riddled with issue. The current global pandemic has caused a great deal of anxiety, animosity, and struggle, and it would be a ridicule if I were not to say that it has had a major effect on the quality of this project. The lack of order had taken its toll, and I fell behind in the early stages. Communication between myself and my supervisor was minimal, mostly due to personal issues on my behalf regarding the pandemic. Nonetheless, I pushed through and pieced together a final piece that I could be somewhat proud of considering the situation.

References

1] | S. Herculano-Houzel, "The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost," 20 June 2012. [Online]. Available: https://www.pnas.org/content/109/Supplement_1/10661.

2] | Y. B. Y. H. G. Lecun, "Deep learning," 27 May 2015. [Online]. Available: https://www.scopus.com/record/display.uri?eid=2-s2.0-84930630277&origin=inward&txGid=c045c6f0d17173503f5b4cf9a6df3570.

3] | Office for National Statistics, "Deaths registered in England and Wales (series DR): 2015," 14 November 2016. [Online]. Available: https://www.ons.gov.uk/peoplepopulationandcommunity/birthsdeathsandmarriages/deaths/bulletins/deathsregisteredinenglandandwalesseriesdr/2015.

4] | Alzheimers.net, "2019 Alzheimer's Statistics," Alzheimers.net, 2019. [Online]. Available: https://www.alzheimers.net/resources/alzheimers-statistics/#:~:text=By%202050%2C%20it's%20estimated%20there,with%20some%20form%20of%20dementia..

5] | National Institute on Aging, "What Happens to the Brain in Alzheimer's Disease?," 16 May 2017. [Online]. Available: https://www.nia.nih.gov/health/what-happens-brain-alzheimers-disease.

6] | G. M. Adrien Payan, "Predicting Alzheimer's disease: a neuroimaging study with 3D convolutional neural networks," 09 Februray 2015. [Online]. Available: arXiv:1502.02506.

7] | M. S. A. A. S. M. Asish Gupta, "Natural Image Bases to Represent Neuroimaging Data," in *ICML'13: Proceedings of the 30th International Conference on International Conference on Machine Learning* , http://proceedings.mlr.press/v28/gupta13b.pdf, 2013, pp. 987 - 994.

8] | T. Mitchell, "Machine Learning," 01 March 1997. [Online]. Available: http://profsite.um.ac.ir/~monsefi/machine-learning/pdf/Machine-Learning-Tom-Mitchell.pdf.

9] | J. Brownlee, "Supervised and Unsupervised Machine Learnign Algorithms," Machine Learning Mastery, 12 August 2019. [Online]. Available: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/.

10] | "Underfitting vs Overfitting," Scikit-learn, [Online]. Available: https://scikit-learn.org/0.15/auto_examples/plot_underfitting_overfitting.html.

11] | V. Bushaev, "How do we 'train' neural networks ?," 27 November 2017. [Online]. Available: https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73.

| J. Brownlee, "A Gentle Introduction to Cross-Entropy for Machine Learning," Machine Learning Mastery, 20 December 2019. [Online]. Available: https://machinelearningmastery.com/cross-entropy-for-machine-learning/. 12]

| J. Brownlee, "Understand the Impact of Learning Rate on Neural Network Performance," 06 February 2020. [Online]. Available: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/. 13]

| J. B. Diederik P. Kingma, "Adam: A Method for Stochastic Optimization," 22 December 2014. [Online]. Available: arXiv:1412.6980. 14]

| R. Khandelwal, "L1 and L2 Regularization," Medium, 04 November 2018. [Online]. Available: https://medium.com/datadriveninvestor/l1-l2-regularization-7f1b4fe948f2. 15]

| Y. S. Y. S. G. E. K. Lu Lu, "Dying ReLU and Initialization: Theory and Numerical Examples," 15 March 2019. [Online]. Available: https://arxiv.org/abs/1903.06733. 16]

| S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way," 15 December 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. 17]

| S. A. Afshine Amidi, "Convolutional Neural Networks cheatsheet," [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks. 18]

| Jefkine, "Backpropagation In Convolutional Neural Networks," DeepGrid, 05 Septmber 2016. [Online]. Available: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/. 19]

| J. S. D. V. D. a. J. G. Jaber Juntu, "Bias Field Correction for MRI Images," in *) Computer Recognition Systems. Advances in Soft Computing, vol 30.*, Berlin, Springer, 2005. 20]

| M. J. Brett, "The problem of functional localization in the human brain," in *Nature Reviews Neuroscience*, http://doi.org/10.1038/nrn756, 2002, pp. 243-249. 21]

| "Spatial Smoothing," Brain Innovation, 09 April 2018. [Online]. Available: https://support.brainvoyager.com/brainvoyager/functional-analysis-preparation/29-pre-processing/86-spatial-smoothing#:~:text=Spatial%20smoothing%20means%20that%20data%20points%20are%20averaged%20with%20their%20neighbours.&text=The%20standard%20procedure%2. 22]

| ADNI Organisation, "About ADNI," [Online]. Available: http://adni.loni.usc.edu/about/. 23]

| "Google Colab," Google, [Online]. Available: https://colab.research.google.com.
24]

| "FSL Wiki," FSL, [Online]. Available: https://fsl.fmrib.ox.ac.uk/fsl/fslwiki.
25]

| "Python," Python, [Online]. Available: https://www.python.org/.
26]

| "DLTK," Deep Learning Toolkit, [Online]. Available: https://dltk.github.io/.
27]

| "Tensorflow," Tensorflow, [Online]. Available: https://www.tensorflow.org/.
28]

| "Keras," Keras, [Online]. Available: https://keras.io/.
29]

| "Nipype: Neuroimaging in Python," Nipype, [Online]. Available:
30] https://nipype.readthedocs.io.

| T. Lewis, "What is an MRI (Magnetic Resonance Imaging)?," 12 August 2017. [Online].
31] Available: https://www.livescience.com/39074-what-is-an-mri.html.

| A. Wilson, "A Brief Introduction to Supervised Learning," 29 September 2019.
32] [Online]. Available: https://towardsdatascience.com/a-brief-introduction-to-supervised-learning-54a3e3932590.

| M. R. M. Talabis, I. Miyamoto, D. Kaye, R. McPherson and J. L. Martin, Information
33] Security Analytics, Elsevier Inc, 2015.

| A. Bhande, "What is underfitting and overfitting in machine learning and how to deal
34] with it.," 11 March 2018. [Online]. Available: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76.

| Missing Link, "Neural Network Concepts: 7 Types of Neural Network Activation
35] Functions: How to Choose?," [Online]. Available: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/.

| R. Protocol, "Everything you need to know about Neural Networks," 4 December
36] 2017. [Online]. Available: https://medium.com/ravenprotocol/everything-you-need-to-know-about-neural-networks-6fcc7a15cb4.

| J. Dacom, "An introduction to Artificial Neural Networks (with example)," 03 October 2017. [Online]. Available: https://medium.com/@jamesdacombe/an-introduction-to-artificial-neural-networks-with-example-ad459bb6941b. 37]

| S. Sharma, "Activation Functions in Neural Networks," 06 September 2017. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. 38]

| Missing Link, "A Recurrent Neural Network Glossary: Uses, Types, and Basic Structure," [Online]. Available: https://missinglink.ai/guides/neural-network-concepts/recurrent-neural-network-glossary-uses-types-basic-structure/. 39]

| M. Jadhav, "Gradient descent: Why and How ?," 30 September 2019. [Online]. Available: https://medium.com/analytics-vidhya/gradient-descent-why-and-how-e369950ae7d3. 40]

| S. Bhattarai, "What is Gradient Descent in Machine Learning," 22 June 2018. [Online]. Available: https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/. 41]

| M. D. Fox, A. Z. Snyder, J. L. Vincent, M. Corbetta, D. C. V. Essen and M. E. Raichle, "The human brain is intrinsically organized into dynamic, anticorrelated functional networks," 5 July 2005. [Online]. Available: https://doi.org/10.1073/pnas.0504136102. 42]

| "Alzheimer's Disease Fact Sheet," National Institute on Aging, [Online]. Available: https://www.nia.nih.gov/health/alzheimers-disease-fact-sheet. 43]

| Alzheimer's Disease Iniative, "About ADNI," ADNI, [Online]. Available: http://adni.loni.usc.edu/about/. 44]