Classification of Pipe Weld Images with Deep Neural Networks
— Literature Survey —

Dalyac Alexandre
ad6813@ic.ac.uk

Supervisors: Prof Murray Shanahan and Mr Jack Kelly
Course: CO541, Imperial College London

June 12, 2014

Abstract

Automatic image classification experienced a breakthrough in 2012 with the advent of GPU implementations of deep neural networks. Since then, state-of-the-art has centred around improving these deep neural networks. The following is a literature survey of papers relevant to the task of learning to automatically multi-tag images of pipe welds, from a restrictive number of training cases, and with high-level knowledge of some abstract features. It is therefore divided into 5 sections: foundations of machine learning with neural networks, deep convolutional neural networks (including deep belief networks), multi-tag learning, learning with few training examples, and incorporating knowledge of the structure of the data into the network architecture to optimise learning.

Include a separate section on progress that describes: the activities and accomplishments of the project to date; any problems or obstacles that might have cropped up and how those problems or obstacles are being dealt with; and plans for the next phases of the project.

Contents

1 Introduction

Automatic classification of pipe weld images has strong academic and economic motivations: the classification task is highly stochastic, to the extent that until 2012 technology was incapable of providing the means to automate it. That is to say, the distribution of each class to detect is of such high variance that it has it is rarely possible for computer vision programmers to hand-specify sufficiently abstract feature detectors that attain high classification accuracy. Deep learning distinguishes itself from other forms of machine learning by learning features; given sufficiently many training examples and computational power, it can learn a high number of features at many different levels of abstractness. In this particular case, the number of training cases may be insufficient to learn good features with deep learning algorithms. One way to remedy this would be make use of knowledge of the high-level features used by humans for this particular task. However, little to no research exists in this field.

The economic motivation is industrial: ControlPoint currently resorts to humans to classify images, which results in high costs and a much slow turnover than automatic classification could offer.

Include a separate section on progress that describes: the activities and accomplishments of the project to date; any problems or obstacles that might have cropped up and how those problems or obstacles are being dealt with; and plans for the next phases of the project.

1.1 Motivation and Objectives

1.2 Contributions

2 Background

This project aims to automate the classification of pipe weld images with deep neural networks. After explaining and formalising the problem, we will explain fundamental concepts in machine learning, then go on to explain the architecture of a deep convolutional neural network with restricted linear units, and finally explain how the network is trained with stochastic gradient descent, backpropagation and dropout. The last sections focus on three challenges specific to the pipe weld image classification task: multi-tagging, learning features from a restricted training set, and class imbalance.

2.1 Defining the Problem

The problem consists in building a classifier of pipe weld images capable of detecting the presence of multiple characteristics in each image.

2.1.1 Explaining the Problem

Practically speaking, the reason for why this task involves multiple tags per image is because the quality of a pipe weld is assessed not on one, but 17 characteristics, as shown below.

Characteristic	Penalty Value
No Ground Sheet	5
No Insertion Depth Markings	5
No Visible Hatch Markings	5
Other	5
Photo Does Not Show Enough Of Clamps	5
Photo Does Not Show Enough Of Scrape Zones	5
Fitting Proximity	15
Soil Contamination Low Risk	15
Unsuitable Scraping Or Peeling	15
Water Contamination Low Risk	15
Joint Misaligned	35
Inadequate Or Incorrect Clamping	50
No Clamp Used	50
No Visible Evidence Of Scraping Or Peeling	50
Soil Contamination High Risk	50
Water Contamination High Risk	50
Unsuitable Photo	100

Table 1: Code Coverage for Request Server

At this point, it may help to explain the procedure through which these welds are made, and how pictures of them are taken. The situation is that of fitting two disjoint polyethylene pipes with electrofusion joints [**?**], in the context of gas or water infrastructure. Since the jointing is done by hand, in an industry affected with alleged "poor quality workmanship", and is most often followed by burial of the pipe under the ground, poor joints occur with relative frequency [**?**]. Since a contamination can cost up to 100,000 [**?**], there exists a strong case for putting in place protocols to reduce the likelihood of such an event. ControlPoint currently has one in place in which, following the welding of a joint, the on-site worker sends one or more photos, at arm's length, of the completed joint.

These images are then manually inspected at the ControlPoint headquarters and checked for the presence of the adverse characteristics listed above. The joint is accepted and counted as finished if the number of penalty points is sufficiently low (the threshold varies from an installation contractor to the next, but 50 and above is generally considered as unacceptable). Although these characteristics are all outer observations of the pipe fitting, they have shown to be very good indicators of the quality of the weld [**?**]. Manual inspection of the pipes is not only expensive, but also delaying: as images are queued for inspection, so is the completion of a pipe fitting. Contractors are often under tight operational time constraints in order to keep the shutting off of

Figure 1: soil contamination risk (left), water contamination risk (centre), no risk (right)

gas or water access to a minimum, so the protocol can be a significant impediment. Automated, immediate classification would therefore bring strong benefits.

2.1.2 Formalising the problem: Multi-Instance Multi-Label Supervised Learning

The problem of learning to classify pipe weld images from a labelled dataset is a Multi-Instance Multi-Label (MIML) supervised learning classification problem [**?**]:

Given an instance space $\mathcal{X}$, a set of class labels $\mathcal{Y}$, a dataset $\{(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)\}$, learn a function $f : 2^{\mathcal{X}} \to 2^{\mathcal{Y}}$ where
$X_i \subseteq \mathcal{X}$ is a set of instances $\{x_1^{(i)}, x_2^{(i)}, ..., x_{p_i}^{(i)}\}$
$Y_i \subseteq \mathcal{Y}$ is the set of classes $\{y_1^{(i)}, y_2^{(i)}, ..., y_{p_i}^{(i)}\}$ such that $x_j^{(i)}$ is an instance of class $y_j^{(i)}$
p_i is the number of class instances (i.e. labels) present in X_i.

This differs from the traditional supervised learning classification task, formally given by:

Given an instance space $\mathcal{X}$, a set of class labels $\mathcal{Y}$, a dataset $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, learn a function $f : \mathcal{X} \to \mathcal{Y}$ where
$x_i \in \mathcal{X}$ is an instance
$y_i \in \mathcal{Y}$ is the class of which x_i is an instance.

In the case of MIML, not only are there multiple instances present in each case, but the number of instances is unknown. MIML has been used in the image classification literature when one wishes to identify all objects which are present in the image [**?**]. Although in this case, the motivation is to look out for a specific set of pipe weld visual characteristics, the problem is actually conceptually the same; the number of identifiable classes is simply lower.

2.1.3 Supervised Learning

Learning in the case of classification consists in using the dataset $\mathcal{D}$ to find the hypothesis function f^h that best approximates the unknown function $f^* : 2^{\mathcal{X}} \to 2^{\mathcal{Y}}$ which would perfectly classify any subset of the instance space $\mathcal{X}$. Supervised learning arises when $f^*(x)$ is known for every instance in the dataset, i.e. when the dataset is labelled and of the form $\{(x_1, f^*(x_1)), (x_2, f^*(x_2)), ..., (x_n, f^*(x_n))\}$. This means that $|\mathcal{D}|$ points of f^* are known, and can be used to fit f^h to them, using an appropriate cost function $\mathcal{C}$. $\mathcal{D}$ is therefore referred to as the *training set*.

Formally, supervised learning therefore consists in finding

$$f^h = \underset{\mathcal{F}}{\operatorname{argmin}}\, \mathcal{C}(\mathcal{D}) \tag{1}$$

where $\mathcal{F}$ is the chosen target function space in which to search for f^h .

2.1.4 Approximation vs Generalisation

It is important to note that supervised learning does not consist in merely finding the function which best fits the training set - the availability of numerous universal approximating function

classes (such as the set of all finite order polynomials) would make this a relatively simple task [**?**]. The crux of supervised learning is to find a hypothesis function which fits the training set well *and* would fit well to any subset of the instance space. In other words, approximation and generalisation are the two optimisation criteria for supervised learning, and both need to be incorporated into the cost function.

2.2 Architecture of a Deep Convolutional Neural Network with Rectified Linear Neurons

Learning a hypothesis function f^h comes down to searching a target function space for the function which minimises the cost function. A function space is defined by a parametrised function equation, and a parameter space. Choosing a deep convolutional neural network with rectified linear neurons sets the parametrised function equation. By explaining the architecture of such a neural network, this subsection justifies the chosen function equation. As for the parameter space, it is $\mathbb{R}^{\mathbb{P}}$ (where P is the number of parameters in the network); its continuity must be noted as this enables the use of gradient descent as the optimisation algorithm (as is discussed later).

2.2.1 Models of Neurons

Before we consider the neural network architecture as a whole, let us start with the building block of a neural network: the neuron (mathematically referred to as the *activation function*). Two types of neuron models are used in current state-of-the-art implementations of deep convolutional neural networks: the rectified linear unit and the softmax unit (note that the terms "neuron" and "unit" are used interchangeably). In order to bring out their specific characteristics, we shall first consider two other compatible neuron models: the binary threshold neuron, which is the most intuitive, and the hyperbolic tangent neuron, which is the most analytically appealing. It may also help to know what is being modelled, so a very brief look at a biological neuron shall first be given.

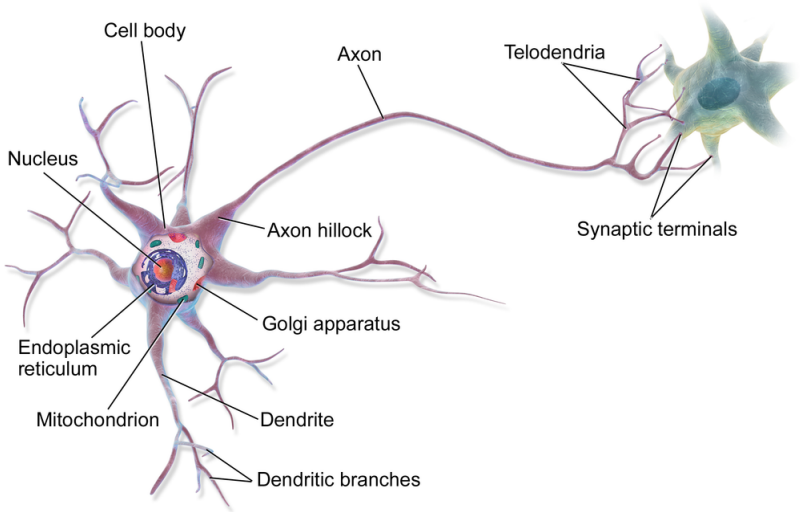

Figure 2: a multipolar biological neuron

Multipolar Biological Neuron A multipolar neuron receives electric charges from neighbouring incoming neurons through its dendritic branches, and sends electric charges to its neighbouring outgoing neurons through its axon. Neurons connect at synapses, which is where the tip of the telodendria of one neuron is in close vicinity of the dendritic branch of another neuron. Because a single axon feeds into all of the telodendria but mutiple dendritic branches feed into the axon hillock, a neuron receives multiple inputs and sends out a single output. Similarly, all of the neuron models below are functions from a multidimensional space to a unidimensional one.

Binary Threshold Neuron

$$y = \begin{cases} 1 & \text{if } M <= b + \sum_{i=1}^{k} x_i \cdot w_i \text{ , where } M \text{ is a threshold parameter} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Intuitively, y takes a hard decision, just like biological neurons: either a charge is sent, or it isn't. y can be seen as producing spikes, x_i as the indicator value of some feature, and $w_[i]$ as a parameter of the function that indicates how important x_i is in determining y. Although this model is closer than most most to reality, the function is not differentiable, which makes it impossible to use greedy local optimisation learning algorithms - such as gradient descent - which need to compute derivatives involving the activation functions.

Logistic Sigmoid Neuron

$$y = \frac{1}{1 + \exp(-z)}, \text{ where } z = b + \sum_{i=1}^{k} x_i \cdot w_i \tag{3}$$

Like the binary threshold neuron, the output domain of this neuron is bounded by 0 and 1. But this time, the function is fully differentiable. Moreover, it is nonlinear, which helps to increase performance [**?**]. To see why, the graph plot below lends itself to the following intuition: if the input x is the amount of evidence for the components of the feature that the neuron detects, and y is the evidence for the feature itself, then the marginal evidence for the feature is decreasing with the amount of evidence for its components (in absolute value terms).

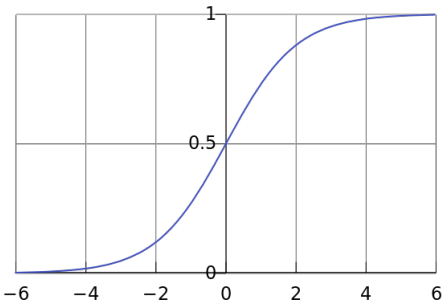

Figure 3: single-input logistic sigmoid neuron

This is like saying that to completely convince y of the total presence or absence of the feature, a lot of evidence is required. However, if there is not much evidence for either case, then y is more lenient. A disadvantage of this neuron model is that it is computationally expensive to compute.

Rectified Linear Neuron

$$y = \max\{0, b + \sum_{i=1}^{k} x_i \cdot w_i\} \tag{4}$$

As can be seen in the graph plot below, the rectified linear neuron is neither fully differentiable (not at 0), nor bounded above. Moreover, it only has two slopes, so its derivative with respect to x_i can only be one of two values: 0 or w_i. Although this may come as a strong downgrade in sophistication compared to the logistic sigmoid neuron, it is so much more efficient to compute (both its value and its partial derivatives) that it enables much larger network implementations [**?**]. Until now, this has more than offset the per-neuron information loss - and saturation risks - of the rectifier versus the sigmoid unit [**?**].

Softmax Neuron

$$y_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}}, \text{ where } z_j = b_j + \sum_{i=1}^{k} x_i \cdot w_{i,j} \tag{5}$$

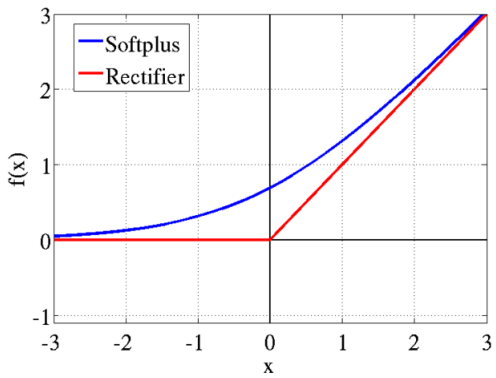

Figure 4: single-input rectified linear neuron

The equation of a softmax neuron needs to be understood in the context of a layer of k such neurons within a neural network: therefore, the notation y_j corresponds to the output of the j^{th} softmax neuron, and $w_{i,j}$ corresponds to the weight of x_i as in input for the j^{th} softmax neuron. A layer of softmax neurons distinguishes itself from others in that neighbouring neurons interact with each other: as can be seen from the equation, the input vectors of all the softmax neurons $z_1, z_2, ..., z_k$ serve to enforce $\sum_{i=1}^{k} y_i = 1$. In other words, the vector $(y_1, y_2, ..., y_k)$ defines a probability mass function. This makes the softmax layer ideal for classification: neuron j can be made to represent the probability that the input is an instance of class j. Another attractive aspect of the softmax neuron is that its derivative is quick to compute: it is given by $\frac{dy}{dz} = \frac{y}{1-y}$.

2.2.2 Feed-Forward Architecture

A feed-forward neural network is a representation of a function in the form of a directed acyclic graph, so this graph can be interpreted both biologically and mathematically. A node represents a neuron as well as an activation function f, an edge represents a synapse as well as the composition of two activation functions $f \circ g$, and an edge weight represents the strength of the connection between two neurons as well as a parameter of f. The figure below (taken from [**?**]) illustrates this.

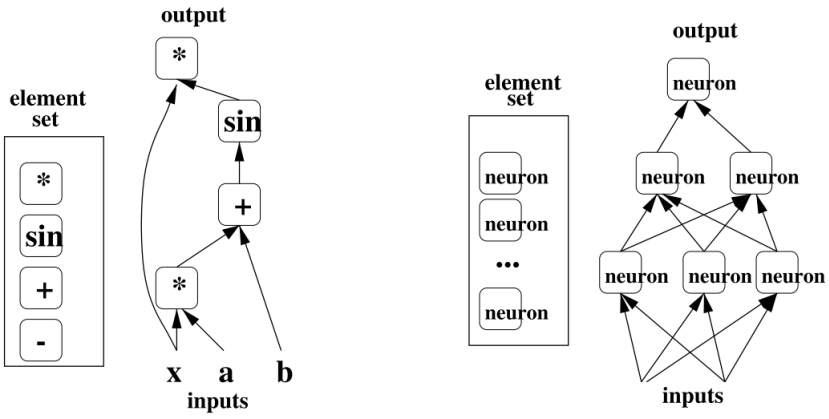

Figure 5: graphical representation of $y = x * sin(a * x + b)$ and of a feed-forward neural network

The architecture is feed-forward in the sense that data travels in one direction, from one layer to the other. This defines an input layer (at the bottom) and an output layer (at the top) and enables the representation of a mathematical function.

Shallow Feed-Forward Neural Networks: the Perceptron A feed-forward neural net is called a perceptron if there exist no layers between the input and output layers. The first neural networks, introduced in the 1960s [**?**], were of this kind. This architecture severely reduces the function space: for example, with $g_1 : x \to sin(s)$, $g_2 : x, y \to x * y$, $g_3 : x, y \to x + y$ as activation functions (i.e. neurons), it cannot represent $f(x) \to x * sin(a * x + b)$ mentioned above [**?**]. This

was generalised and proved in *Perceptrons: an Introduction to Computation Geometry* by Minsky and Papert (1969) and lead to a move away from artificial neural networks for machine learning by the academic community throughout the 1970s: the so-called "AI Winter" [**?**].

Deep Feed-Forward Neural Networks: the Multilayer Perceptron The official name for a deep neural network is Multilayer Perceptron (MLP), and can be represented by a directed acyclic graph made up of more than two layers (i.e. not just an input and an output layer). These other layers are called hidden layers, because the "roles" of the neurons within them are not set from the start, but learned throughout training. When training is successful, each neuron becomes a feature detector. At this point, it is important to note that feature learning is what sets machine learning with MLPs apart from most other machine learning techniques, in which features are specified by the programmer [**?**]. It is therefore a strong candidate for classification tasks where features are too numerous, complex or abstract to be hand-coded - which is arguably the case with pipe weld images.

Intuitively, having a hidden layer feed into another hidden layer above enables the learning of complex, abstract features, as a higher hidden layer can learn features which combine, build upon and complexify the features detected in the layer below. The neurons of the output layer can be viewed as using information about features in the input to determine the output value. In the case of classification, where each output neuron corresponds to the probability of membership of a specific class, the neuron can be seen as using information about the most abstract features (i.e. those closest to defining the entire object) to determine the probability of a certain class membership.

Mathematically, it was proved in 1989 that MLPs are universal approximators [**?**]; hidden layers therefore increase the size of the function space, and solve the initial limitation faced by perceptrons.

Deep Convolutional Neural Networks: for translation invariance A convolutional neural network uses a specific network topology that is inspired by the biological visual cortex and tailored for computer vision tasks, because it achieves translation invariance of the features. Consider the following image of a geranium: a good feature to classify this image would be the blue flower. This feature appears all across the image; therefore, if the network can learn it, it should then sweep the entire image to look for it. A convolutional layer implements this: it is divided into groups of neurons (called *kernels*), where all of the neurons in a kernel are set to the same parameter values, but are 'wired' to different pixel windows across the image. As a result, one feature can be detected anywhere on the image, and the information of where on the image this feature was detected is contained in the output of the kernel. Below is a representation of LeNet5, a deep convolutional neural network used to classify handwritten characters.

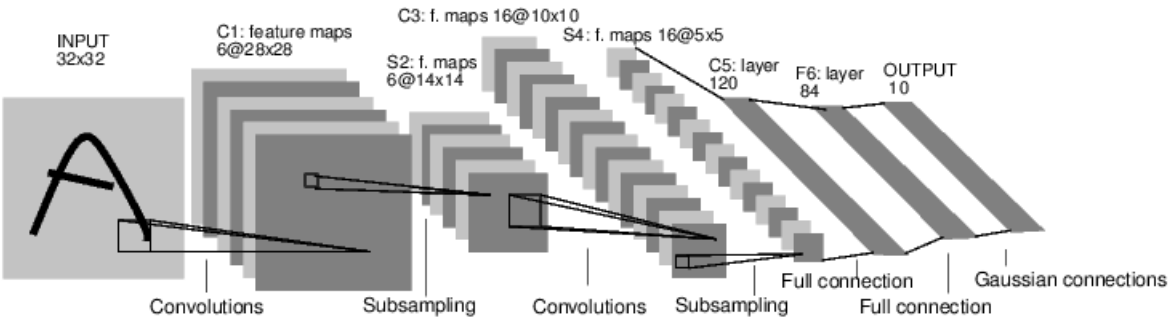

Figure 6: LeNet7 architecture: each square is a kernel

2.3 Training: Backpropagation

Now that the architecture of a deep CNN has been explained, the question remains of how to train it. Mathematically: now that the function space has been explained, the question remains of how this space is searched. In the case of feed-forward neural networks and supervised learning, this is done with gradient descent, a local (therefore greedy) optimisation algorithm. Gradient descent relies on the partial derivatives of the error (a.k.a cost) function with respect to each parameter

of the network; the backpropagation algorithm is an implementation of gradient descent which efficiently computes these values.

2.3.1 Compute Error-Weight Partial Derivatives

Let t be the target output (with classification, this is the label) and let $y = (y_1, y_2, ..., y_P)$ be actual value of the output layer on a training case. (Note that classification is assumed here: there are multiple output neurons, one for each class).

The error is given by

$$E = \mathcal{C}(\mathbf{t} - \mathbf{y}) \tag{6}$$

where $\mathcal{C}$ is the chosen cost function. The error-weight partial derivatives are given by

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i} \cdot \frac{\partial y_i}{\partial net} \cdot \frac{\partial net}{\partial w_{ij}} \tag{7}$$

Since in general, a derivative $\frac{\partial f}{\partial x}$ is numerically obtained by perturbing x and taking the change in $f(x)$, the advantage with this formula is that instead of individually perturbing each weight w_{ij}, only the unit outputs y_i are perturbed. In a neural network with k fully connected layers and n units per layer, this amounts to $\Theta(k \cdot n)$ unit perturbations instead of $\Theta(k \cdot n^2)$ weight perturbations (note that the bound on weight perturbations is no longer tight if we drop the assumption of fully connected layers).

How Gradient Propagates It is important for intuition to realise how the gradient propagates through a network - especially since a single piece of information - the error - is supposed to adjust up to millions of parameters, which could be considered as a bit of a stretch. Let us note that in the case of classification, the error is not a scalar, but a vector of size equal to the number of classes, so it carries more than "one piece" of information.

With the following notation:

- y_j, the output of unit (a.k.a neuron) j, but also used to refer to the unit j itself
- w_{ij}, the weight of the edge connecting lower-layer neuron y_i to upper-layer neuron y_j
- $z_j := b + \langle x, w \rangle = b + \sum_{i=1}^{k} x_i \cdot w_{ij}$, the input vector for y_j – therefore $y_j = \psi(z_j)$

The rules for propagating the gradient backwards through a network are:

- to **initialise**: $grad \leftarrow \mathcal{C}'(y_L)$, where y_L is the output unit
- to **propagate through a unit** y_j: $grad \leftarrow grad \cdot \psi'(z_j)$
- to **propagate through an edge** w_{ij}: $grad \leftarrow grad \cdot w_{ij}$
- to **stop at an edge** w_{ij}: $grad \leftarrow grad \cdot y_i$

So for example, given the figure below:

- for $\frac{\partial E}{\partial w_{57}}$: initialise, propagate through y_7, then stop at w_{57}: $\mathcal{C}'(y_7) \cdot \psi'(z_7) \cdot y_5$
- for $\frac{\partial E}{\partial w_{25}}$: initialise, propagate through y_7, w_{57}, y_5 then stop at w_{25}: $\mathcal{C}'(y_7) \cdot \psi'(z_7) \cdot w_{57} \cdot \psi'(z_5) \cdot y_2$

2.3.2 Update Weight Values (with Gradient Descent)

The learning rule is given by $w_{i,t+1} = w_{i,t+1} + \tau \cdot \frac{\partial E}{\partial w_{i,t}}$

Visually, this means that weight values move in the direction the will reduce the error quickest, i.e. the direction of steepest descent on the error surface is taken. Notice that given the learning rule, gradient descent converges (i.e. $w_{i,t+1}$ equals $w_{i,t+1}$) when the partial derivative reaches zero. This corresponds to a local minimum on the error surface. In the figure below, two potential training sessions are illustrated. The minima attained in each cases are not the same. This illustrates a strong shortcoming with backpropagation: parameter values can get stuck in poor local minima.

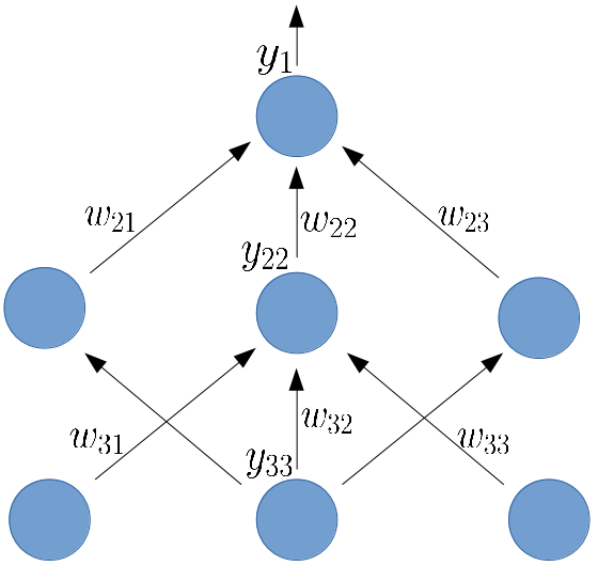

Figure 7: (CHANGE NOTATION to reflect one above with $\mathbf{y}_L$ for top layer) Feed-Forward Neural Network with one Hidden Layer

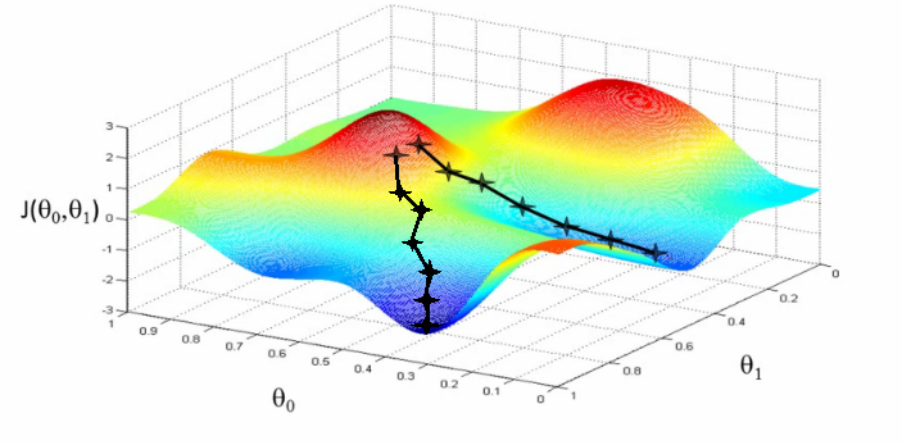

Figure 8: an error surface with poor local minima

Training, Validation and Test Sets As mentioned previously, learning is not a mere approximation problem because the hypothesis function must generalise well to any subset of the instance space. Approximation and generalisation are incorporated into the training of deep neural networks (as well as other models [**?**]) by separating the labelled dataset into a training set, a validation set and a test set. The partial derivatives are computed from the error over the training set, but the function that is learned is the one that minimises the error over the validation set, and its performance is measured on the test set. The distinction between training and validation sets is what prevents the function from overfitting the training data: if the function begins to overfit, the error on the validation set will increase and training will be stopped. The distinction between the test set and the validation set is to obtain a stochastically impartial measure of performance: since the function is chosen to minimise the error over the validation set, there could be some non-negligible overfit to the validation set which can only be reflected by assessing the function's performance on yet another set.

2.4 Challenges specific to the Pipe Weld Classification Task

A number of significant challenges have arisen from this task: multi-tagging, domain change, small dataset size (by deep learning standards) and class imbalance. Before going into them, an overview of the data is given below.

2.4.1 Data Overview

ControlPoint recently upgraded the photographical equipment with which photos are taken (from 'Redbox' equipment to 'Bluebox' equipment), which means that the resolution and finishing of the photos has been altered. There are 113,865 640x480 'RedBox' images. There are 13,790 1280x960 'BlueBox' images. Label frequencies for the Redbox images are given below.

Characteristic	Count
No Ground Sheet	30,015
No Insertion Depth Markings	17,667
No Visible Hatch Markings	28,155
Other	251
Photo Does Not Show Enough Of Clamps	5,059
Photo Does Not Show Enough Of Scrape Zones	21,272
Fitting Proximity	1,233
Soil Contamination Low Risk	10
Unsuitable Scraping Or Peeling	2,125
Water Contamination Low Risk	3
Joint Misaligned	391
Inadequate Or Incorrect Clamping	1,401
No Clamp Used	8,041
No Visible Evidence Of Scraping Or Peeling	25,499
Soil Contamination High Risk	6,541
Water Contamination High Risk	1,927
Unsuitable Photo	2
Perfect (no labels)	49,039

Table 2: Count of Redbox images with given label

2.4.2 Multi-Tagging

:

As mentioned earlier, training images contain varying numbers of class instances; this uncertainty complexifies the training task.

2.4.3 Domain Change

:

Domain change can be lethal to machine vision algorithms: for example, a feature learned (at the pixel level) from the 640x480 Redbox images could end up being out of scale for the 1280x960 Bluebox images. However, this simple example is not relevant to a CNN implementation, since the largest networks can only manage 256x256 images, so Bluebox and Redbox images will both be downsized to identical resolutions. However, more worrying is the difference in image sharpness between Redbox and Bluebox images, as can be seen below. It remains to be seen how a CNN could be made to deal with this type of domain change.

Nevertheless, evidence has been found to suggest that deep neural networks are robust to it: an experiment run by Donahue et al on the *Office* dataset [**?**], consisting of images of the same products taken with three different types of photographical equipment (professional studio equipment, digital SLR, webcam) found that their implementation of a deep convolutional neural network produced similar feature representations of two images of the same object even when the two images were taken with different equipment, but that this was not the case when using SURF, the currently best performing set of hand-coded features on the *Office* dataset [**?**].

2.4.4 Small Dataset Size

Alex Krizhevsky's record-breaking CNN was trained on 1 million images [**?**]. Such a large dataset enabled the training of a 60-million parameter neural network, without leading to overfit.

Figure 9: left: a Redbox photo - right: a Bluebox photo

In this case, there are 'only' 127,000, and 43% of them are images of "perfect" welds, meaning that these are label-less. Training a similarly sized network leads to overfit, but training a smaller network could prevent the network from learning sufficiently abstract and complex features for the task at hand. A solution to consider is that of transfer learning [**?**], which consists in importing a net which has been pretrained in a similar task with vast amounts of data, and to use it as a feature extractor. This would bring the major advantage that a large network architecture can be used, but the number of free parameters can be reduced to fit the size of the training set by "freezing" backpropagation on the lower layers of the network. Intuitively, it would make sense to freeze the lower (convolutional) layers and to re-train the higher ones, since low-level features (such as edges and corners) are likely to be similar across any object recognition task, but the way in which these features are combined are specific to the objects to detect.

2.4.5 Class Imbalance

The dataset suffers from a similar adverse characteristic to that of medical datasets: pathology observations are significantly less frequent that healthy observations. This can make mini-batch training of the network especially difficult. Consider the simple case of training a neural network to learn the following labels: No Clamp Used, Photo Does Not Show Enough Of Clamps, Clamp Detected (this label is not in the list, but can be constructed as the default label). Only 8% of the Redbox images contain the first label, and only 5% contain the second label, so if the partial derivatives of the error are computed over a batch of 128 images (as is the case with the best implementations [**?**], [**?**], [**?**]), one can only expect a handful of them to contain either of the first two labels. Intuitively, one may ask: how could I learn to recognise something if I'm hardly ever shown it?

One possible solution would be to use a different cost function: the F-measure [**?**], which is known to deal with these circumstances. Although the F-measure has been adapted to into a fully differentiable cost function (which is mandatory for gradient descent), there currently exists no generalisation of it for the case of $n \geq 2$ classes. Moreover, one would need to ensure that the range of the error is as large as possible for a softmax activation unit, whose own output range is merely $[0; 1]$.

3 Design

4 Implementation

5 Experimentation

6 Conclusions and Future Work

this is just [1] a sentence [2] showing [3] how citations/references work.

References

[1] Krishnan Anantheswaran, *istanbul: A Javascript code coverage tool written in JS* URL: `http://gotwarlost.github.io/istanbul/`, last accessed 9th March 2014.

[2] H. Goeau, A. Joly, P. Bonnet, *LifeCLEF 2014, Plant Task*. URL: `http://www.imageclef.org/node/179`, last accessed: 11th March 2014.

[3] TJ Holowaychuk, *Mocha - the fun, simple, flexible JavaScript test framework* URL: `http://visionmedia.github.io/mocha/#getting-started`, last accessed 9th March 2014.

7 From Plant Report - useful looking stuff

- **Specification** : The original outline of the project, our interpretation of it, and the specific tasks we set for ourselves.
- **System Architecture** : An overview describing how the three components of our product fit together and interact.
- **Product Design** : The look and feel of our product to the user.
- **Methodology** : Our production development strategy, including unit testing.
- **Implementation** : The biggest challenges we encountered and how we dealt with them.
- **Final Product** : Evaluation of the performance of our product, and the commercial opportunities it presents.

Cat	Description	Priority	ECD	Status
E	**iOS 7 application that can take and store photos.**	H	Sprint1	Complete
E	**Application can upload photos to server.**	H	Sprint1	Complete
E	**Application can receive and handle classification result from server.**	H	Sprint1	Complete
E	**Classification result can be displayed to user.**	H	Sprint1	Complete
S	*Photo geographical location stored.*	L	Sprint1	Complete
S	*HTML5 web interface available for desktop and non-iOS mobile.*	M	Sprint3	Removed
S	*Link to web (e.g. Wikipedia) entry for species.*	M	Sprint3	Complete
S	*Application released on Apple's App Store.*	M	Sprint3	Complete
S	*Comparison image displayed.*	M	Sprint3	Removed
S	*User feedback of result returned to server.*	L	Sprint3	Removed
+	**User can select one of four plant components (e.g. Leaf, Fruit, Flower) to improve result accuracy.**	-	Sprint3	Complete

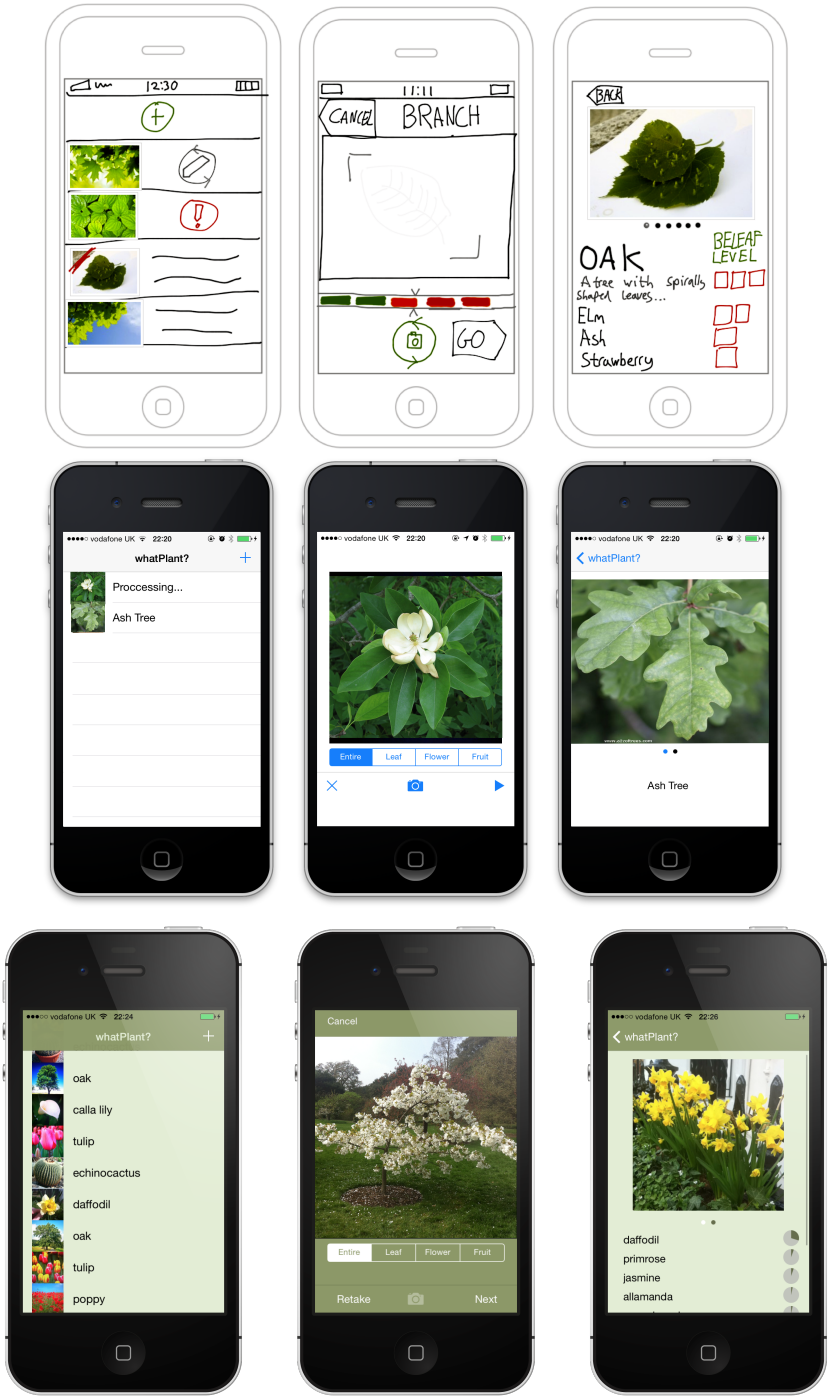

Figure 10: Evolution of the App

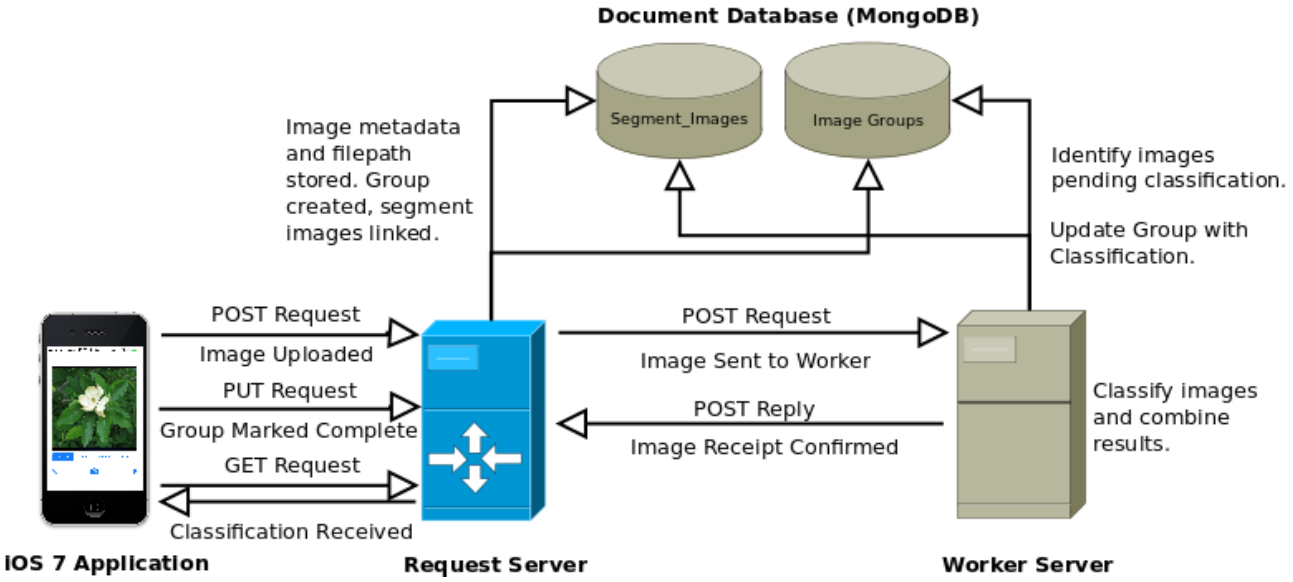

Figure 11: System Context Diagram

Module	Statement Coverage	Branch Coverage
Configuration Parsing	100% - 52/52	100% - 22/22
POST and GET Routes	81% - 102/126	77% - 26/34
Server Instantiation	88% - 46/52	75% - 9/12
Total Coverage	86% - 200/230	87% - 57/68

Table 3: Code Coverage for Request Server

Challenge: *Data Processing*

Deep learning itself is distinguished from other machine learning techniques by not simply learning the relative importance of features, but by learning the features themselves. The additional information needed for this raises the desired volume of training data. The PlantCLEF dataset which was originally intended to underlie training, provided a meagre average of 12 images per species; our data sourcing pursuits managed to bring this number up to 2,000 per species. The only significantly large labelled dataset was ImageNet.

To ensure our neural network wasnt burdened having to learn to recognise species for which we had a limited training subset. A Bucketing algorithim was created in order to bucket those images into a higher level in our taxonomy tree which we constructed using WordNet [**?**]. This meant we could decrease our neural network error rate while maximising our use of available training data. Outlined below is the pseudocode for the bucketing algorithm.

```
1: procedure UPDATE-DESCENDANT-COUNT(path)
2:     count ← 0
3:     for all nodes ∈ path do
4:         count ← count + NumPlants[node]
5:         UPDATE Bucket = node, BucketSpecies = Species, Count += count
6:         FROM plants
7:         WHERE SynsetID = node
8:     end for
9: end procedure
```

```
1: procedure ASSIGN-BUCKETS(path)
2:     for i ← 0...path.length do
3:         bucket ← path[i]
4:         count ← NumPlants[bucket]
5:         if count >= threshold then
6:             UPDATE Bucket = bucket, BucketSpecies = species
7:             FROM plants
8:             WHERE SynsetID IN path[0...i]
9:             break
10:        end if
11:    end for
12: end procedure
```

A appendix part 1

A.1 appendix part 1.1

A.2 appendix part 1.2

B appendix part 2

B.1 appendix part 2.1

B.2 appendix part 2.2