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 6, 2014

Abstract

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.

In terms of progress, a deep convolutional neural network, pretrained on 10 million images from ImageNet, has been trained on 150,000 pipe weld images for the simplest possible task of discriminating 'good' pipe weld from 'bad' pipe welds. A classification error rate of ??% has been attained. Although this figure is encouraging, it corresponds to a much simpler formulation of the task: the objective of this project is to achieve multi-tagging for 23 characteristics, some of which require considerable human training. 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 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.

1.1 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.

1.1.1 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 [4], 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 [4]. Since a contamination can cost up to 100,000 [4], 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 [4]. 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

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

operational time constraints in order to keep the shutting off of gas or water access to a minimum, so the protocol can be a significant impediment. Automated, immediate classification would therefore bring strong benefits.

1.1.2 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 [2]:

> 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 [2]. 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.

1.1.3 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 .

1.1.4 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 [5]. 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.

1.2 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).

1.2.1 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.

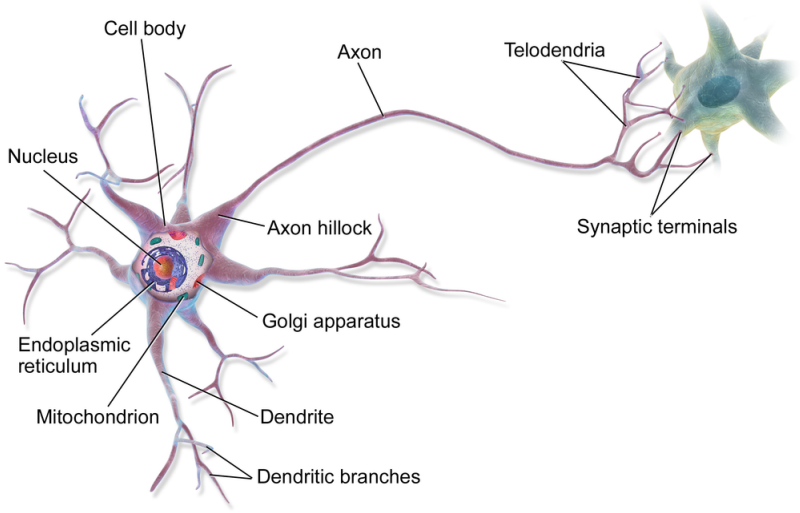

Figure 2: a multipolar biological neuron

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

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

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 [6]. 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).

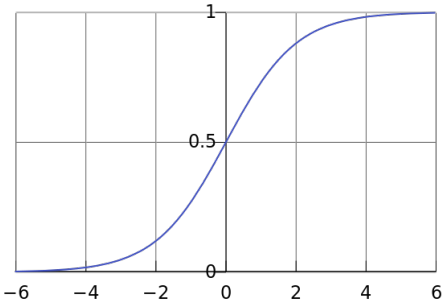

Figure 3: single-input logistic sigmoid neuron

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 [8]. Until now, this has more than offset the per-neuron information loss - and saturation risks - of the rectifier versus the sigmoid unit [9].

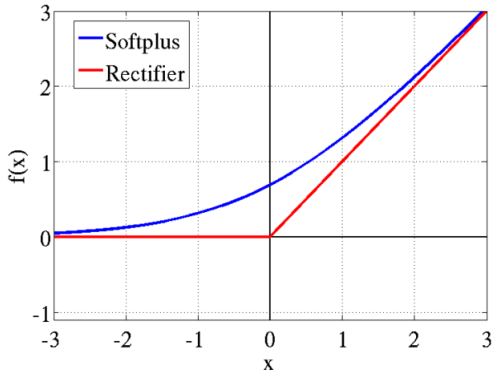

Figure 4: single-input rectified linear neuron

Softmax Neuron

$$y_j = \frac{\exp(z_j)}{\sum_{i=1}^{k} \exp(z_i)}, \text{ where } z_j = \sum_{i=1}^{k} x_i \cdot w_{i,j} \tag{5}$$

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 $\frac{dy}{dz} = \frac{y}{1-y}$.

1.2.2 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 [6]) illustrates this.

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 [6], 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 [6]. This was generalised and proved in *Perceptrons: an Introduction to Computation Geometry* by Minsky and Papert (1969) and

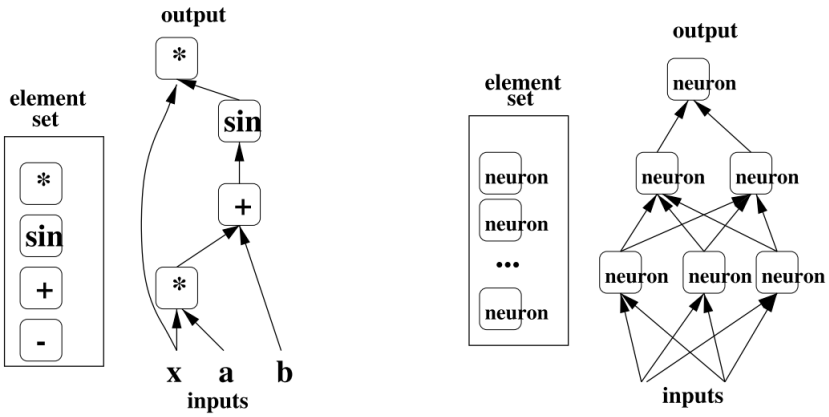

Figure 5: graphical representation of $y = x * sin(a * x + b)$ and of a feed-forward neural network

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 [6]. 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 [10]; 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.

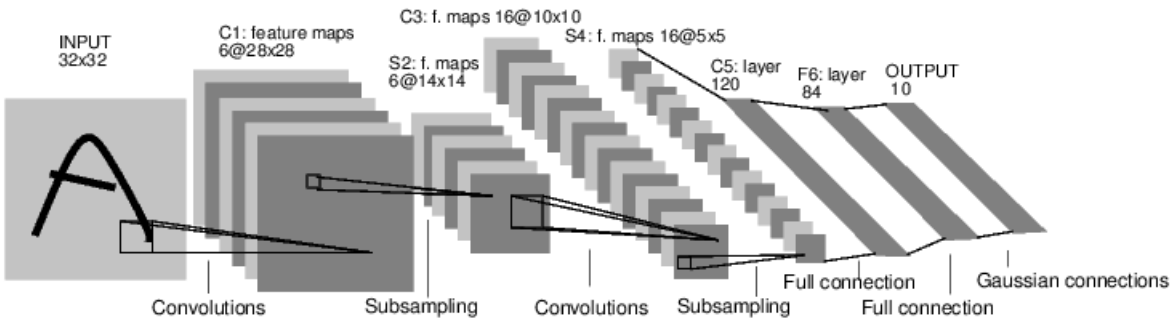

Figure 6: LeNet7 architecture: each square is a kernel

1.3 Training

Gradient Descent the supervised training of deep neural networks relies on the derivative of the error function with respect to the network's parameters, the parameter space needs to be continues, and is therefore $\mathbb{R}^{\mathbb{P}}$ (where P is the number of parameters in the network).

As mentioned above, 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.

Backpropagation This supervised learning algorithm is used to train deep feed-forward neural networks. Its aim is to find values for all the weights in the network that minimise the network's prediction error (you need labeled data to compute prediction error, hence why it only works with supervised learning). As a 1st step, it efficiently estimates the partial derivative of the error with respect to every weight (ie "how much would the error increase by if I were to slightly increase the value of this weight?"). As a 2nd step, it uses these partial derivative estimates to update ie learn the weight values.

Compute Error-Weight Partial Derivatives This might seem straightforward: tweak the value of one weight, see how the error changes, and you've got your partial derivative with respect to that weight. That would be called randomly perturbing weights. But computing the error means feeding a training case through the entire network. You also need a statistically significant estimate, so you'd have to measure the error with many (typically 100) cases. You'd have to do all of that, one weight at a time (otherwise you don't know which of the weights you've simultaneously tweaked are responsible for what proportion of the error). With cuda-convnet, 60 million weights. Geoffrey Hinton and others realised in the 80s that instead of randomly perturbing weights, you can randomly perturb the output of neurons (one by one), and use the error-neuronOutput partial derivatives to compute all of the error-weight partial derivatives (using the chain rule in calculus). In a 2-layer network of a+b neurons for example, you have a*b weights, so the efficiency gain is significant. This is the algorithm's achievement. That's backpropagation stage 1.

Update Weight Values (With Gradient Descent) At every learning iteration, backpropagation computes these slopes, and uses the values to update the weights. It stops updating weights

once all slopes are zero (ie no small change in the value of any weight will reduce the error). That's gradient descent. To get the intuition for learning in general, and gradient descent specifically: given K weights and an error function, picture the K+1 dimensional space (or rather picture it for K=2!) where the vertical dimension is where the error is plotted, and all of the other dimensions are where weight values are plotted. Now picture the surface that is obtained as the weight vector spans all of its possible values. (for K=2, picture the surface as plain ground with bumps and craters here and there). Learning is simply about finding the lowest point on that surface, ie finding the weights that minimise the error. Graphically, backpropagation stops once the bottom of a crater has been met. But we have no idea whether this crater is the deepest one. Worse even, by default (we'll have to check whether this is also the case with cuda-convnet) the first crater to be met is the final one. Backprop doesn't explore elsewhere to see if a deeper crater can be found. If we're lucky, the error function has only one crater. Either that, or the initial weights - the ones the network began training with - are somewhere on the slope of the deepest crater. So the initial weights are crucial. (And gradient descent seems pretty naive).

Dropout

1.4 Challenges specific to the Pipe Weld Classification Task

1.4.1 Multi-Instance Multi-Label Learning

1.4.2 Transfer Learning: learning with a restricted training set

1.4.3 Class Imbalance

2 Progress

2.1 Data: pipe weld images

There are 227,730 640x480 'RedBox' images. There are 1280x960 'BlueBox' images. They all have xx tags.

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 flag)	49,039

Table 2: Count of Redbox images with given label

2.1.1 Visual Inspection

2.1.2 Analysis

ANOVA

t-SNE

2.2 Implementation: Cuda-Convnet

Cuda-Convnet is a GPU implementation of a deep convolutional neural network implemented in CUDA/C++. It was written by Alex Krizhevsky to train the net that set ground-breaking records at the ILSVRC 2012 competition, and subsequently open-sourced. However, parts of his work are missing and his open source repository is not sufficient to reproduce his network (see test run for how this was dealt with).

2.2.1 Test Run

Use Alex Krizhevsky's GPU implementation. Use Daniel Nouri's noccn scripts. Download the file containing serialized code for Yangqing's decaf [1] network parameters, reverse engineer it using Daniel Nouri's skynet configuration files. Write batching script for decaf-net's image dimensions. Write data augmentation scripts to reduce overfit. Write c++ code to import the decaf parameters to initialise the weights of a new network, to re-initialise the weights of the fully connected layers, to train only the fully connected layers, keeping the lower layers 'frozen'.

888 batches of 128 images each. Used 10% of the data for testing - so trained on 102,400 images.

DeCAF By Yangqinq Jia and University of California, Berkeley. *"As the underlying architecture for our feature we adopt the deep convolutional neural network architecture proposed by Krizhevsky et al. (2012), which won the ImageNet Large Scale Visual Recognition Challenge 2012 (Berg et al., 2012) with a top-1 validation error rate of 40.7%. 3 We chose this model due to its performance on a difficult 1000-way classification task, hypothesizing that the activa- tions of the neurons in its late hidden layers might serve as very strong features for a variety of object recognition tasks. Its inputs are the mean-centered raw RGB pixel intensity values of a 224224 image. These values are forward propagated through 5 convolutional layers (with pooling and ReLU non-linearities applied along the way) and 3 fully-connected layers to determine its final neuron activ- ities: a distribution over the tasks 1000 object categories. Our instance of the model attains an error rate of 42.9% on the ILSVRC-2012 validation set 2.2% shy of the 40.7% achieved by (Krizhevsky et al., 2012). We refer to Krizhevsky et al. (2012) for a detailed discus- sion of the architecture and training protocol, which we closely followed with the exception of two small differ- ences in the input data. First, we ignore the images orig- inal aspect ratio and warp it to 256 256, rather than re- sizing and cropping to preserve the proportions. Secondly, we did not perform the data augmentation trick of adding random multiples of the principle components of the RGB pixel values throughout the dataset, proposed as a way of capturing invariance to changes in illumination and color.*

Network Architecture *Don't want to saturate neurons. Need a standard deviation of weight initialisations proportional to the square root of the fan-in. Convolutional layers have a much smaller fan-in than, say, fully connected layers, so the standard deviation needs to be much lower.*

Use dropout of 0.5 in each fully connected layer, so add twice as many outputs. Go from 4096 to 8192. Actually no, keep it at 4096. Furthermore, reduce the number of neurons in each fully connected layer, because don't have as many classes to classify. Just need a few abstract features. The last layer has number of neurons fixed to number of classes. So actually, maybe reduce the number of neurons only in the 2nd fully connected one, but not the first: this is like having a few very abstract features, which are combinations of a very high number of less abstract features.

2.3 Further Work

Directions for further work can be divided into three sections: improving the current model i.e. with small changes, and modifying the model i.e. with significant changes, and finally improving the implementation with more flexible, higher performing APIs.

2.3.1 Improve Current Model

Improving the current model consists in optimising the hyperparameters of the model.

Tweak Architecture *Change proportion of data used as training data. Change fan-in for the fully connected layers. Change dropout rate (probably don't need to). Does number of classes to classify influences network hyper-parameters, other than number of neurons in top layer? Maybe reduce the number of neurons in each fully connected layer even more so, because don't have as many classes to classify. Just need a few abstract features. The last layer has number of neurons fixed to number of classes. So actually, maybe reduce the number of neurons only in the 2nd fully connected one, but not the first: this is like having a few very abstract features, which are combinations of a very high number of less abstract features.*

Tweak Training *I'm training a CNN to recognise 3 classes, but 91% of my data is of the same class. Is there anything I should alter in the way I train?*

For example, should I make the proportions of each class in my validation batches even (1/3, 1/3, 1/3)? Because the thing is, weights are only saved when new validation error is lower than lowest validation error until now. But if this validation error is computed over hardly any examples of the other 2 classes, how can it be a good indicator that the network is on its way to learning good features?

See the plot for training: over 39 epochs, there is no progress, weight updates look random. This is a bad sign.

If one views the problem as picking a direction to walk along in the error surface, it looks like the algorithm is stuck at the stage of randomly trying directions, heading down one a little bit, but deciding that it is not a good direction and starting from scratch. This is either due to the correct direction not being computed, or the correct direction not being correctly recognised.

Maybe dropout is too high: dropout makes the logprob over time more spiky because it's like training lots of models concurrently, so there is never much focus on a particular model. Maybe the dropout rate should be reduced.

Maybe the 2 other classes don't appear often enough, so it's impossible to learn anything: this can apparently be solved by choosing a different error function to minimise: the F-score [3]. However, The F-Measure is proposed as an alternative to the MSE - but can it be used as an alternative to cross entropy? I have a softmax output layer, so all ouput neuron values are between 0 and 1, which drastically reduces the range of the MSE. Would that be the same for F-Measure? Is there a way of making the range of the F-Measure wide for a softmax output layer?

Starting from 39.415, the test frequency was raised from 20 batches to 200 batches. The idea here is to let the weights run along a longer path before evaluating whether this is a good direction (more depth in the search).

2.3.2 Modify Model

Transfer Learning

Hidden Markov Model *It might be interesting to explore hidden markov models because the data has obvious hidden states which humans benefit from learning: pipe welds can be T welds and standard welds, and this alters where scratch marks need to be seen.*

Hidden Markov for pipe type. Whether the joint is a T-joint or not alters the way the image has to be assessed for each of the flags. In other words, whether or not the joint is a T-joint modifies the visual features that have to be picked up by the neural network for classification. Unless I'm mistaken, we do not have data for each image about whether or not it is a T-joint. At first, I had found that a bit disappointing, But I've just been thinking, this might be a very exciting research opportunity: there's a (powerful and super cool) type of machine learning model called a Hidden Markov model, where unknown states can be learned. In this case, the unknown state would "whether or not joint is a T-joint". Knowing this state should in theory help the network to improve classification, because it could indicate to it that a slightly different set of visual features need to be used for classification. I don't think much work has been done on combining neural networks with hidden markov models - and I don't know whether it's possible. But I'd like to mention it in my first report as a potential research path (unless Jack thinks it's a bad idea!). Hence why I wanted to be sure about the terminology for these different joints!

2.3.3 Improve Implementation

Caffe

Theano

Torch

References

[1] *Donahue, Jeff; Jia, Yangqing; Vinyals, Oriol; Hoffman, Judy; Zhang, Ning; Tzeng, Eric; Darrell, Trevor;* DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition
arXiv preprint arXiv:1310.1531, 2013

[2] *Zhou, Zhi-Hua; Zhang, Min-Ling;* Multi-Instance Multi-Label Learning with Application to Scene Classification
Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006

[3] *Pastor-Pellicer, Joan; Zamora-Martinez, Francisco; Espana-Boquera, Salvador; Castro-Bleda, Maria Jose;* F-Measure as the Error Function to Train Neural Networks

[4] *Fusion Group - ControlPoint LLP,* Company Description
URL: `http://www.fusionprovida.com/companies/control-point`, *last accessed 5th June 2014.*

[5] *Barron, Andrew R.,* Universal Approximation Bounds for Superpositions of a Sigmoidal Function
IEEE Transactions on Information Theory, Vol. 39, No. 3 May 1993

[6] *Bengio, Yoshua;* Learning Deep Architectures for AI
Foundations and Trends in Machine Learning, Vol. 2, No. 1 (2009) 1-127 2009

[7] *Russell, Stuart J; Norvig, Peter;* Artificial Intelligence: A Modern Approach
2003

[8] *Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E.;* ImageNet Classification with Deep Convolutional Neural Networks
2012

[9] *Glorot, Xavier; Bordes, Antoine; Bengio, Yoshua;* Deep Sparse Rectifier Neural Networks
2013

[10] *Hornik, Kur; Stinchcombe, Maxwell; White, Halber;* Multilayer Feed-Forward Networks are Universal Approximators
1989