Classification of Pipe Weld Images with Deep Neural Networks
— Final Report —

Dalyac Alexandre
ad6813@ic.ac.uk

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

September 2, 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.

In terms of progress, several instances of large-scale neural networks have been trained on a simplifed task: that of detecting clamps in Redbox images. None of them have shown signs of parameter convergence, suggesting that the classification task is particularly challenging. Potential reasons are discussed; the next steps of the project are to test them.

Contents

1 Introduction **5**

2 Defining the Problem **5**
2.1 Explaining the Problem . . . 5
2.2 Formalising the problem: Multi-Instance Multi-Label Supervised Learning . . . 6
2.3 Challenges specific to the Pipe Weld Classification Task . . . 6
2.3.1 Data Overview . . . 7
2.3.2 Multi-Tagging . . . 7
2.3.3 Domain Change . . . 7
2.3.4 Small Dataset Size . . . 8
2.3.5 Class Imbalance . . . 8

3 Deep Supervised Learning **11**
3.1 Supervised Learning . . . 11
3.1.1 Approximation vs Generalisation . . . 11
3.2 Deep Learning: Neurons and Depth . . . 11
3.2.1 Models of Neurons . . . 11
3.2.2 Why Depth . . . 14
3.3 Backpropagation . . . 17
3.3.1 Compute Error-Weight Partial Derivatives . . . 17
3.3.2 Update Weight Values (with Gradient Descent) . . . 17
3.4 Cross Validation . . . 17

4 Deep Convolutional Neural Networks **19**
4.1 Operations in a convolutional layer . . . 19
4.1.1 Pixel Feature . . . 20
4.1.2 Non-linear Activation . . . 21
4.1.3 Pooling aka Spatial Feature . . . 21
4.1.4 Optional Normalisation . . . 22
4.2 Dropout . . . 23
4.3 Data augmentation . . . 23

5 Analysis 1: ReLU Activation **24**
5.1 Motivations . . . 24
5.2 Mathematical Analysis . . . 25
5.2.1 How the Gradient Propagates . . . 25
5.2.2 An Example . . . 25
5.2.3 Vanishing Gradient . . . 26
5.2.4 Impact of the ReLU . . . 27

6 Analysis 2: Early Stopping **29**

7 Task 1: Generic Clamp Detection **29**
7.1 Motivations . . . 29
7.2 Implementation: Cuda-Convnet . . . 29
7.2.1 Cuda-Convnet: An Out-of-the-box API . . . 30
7.2.2 Hardware: NVidia GeForce GTX 780 . . . 30
7.3 Discovery . . . 30
7.3.1 Non-Converging Error Rates . . . 31
7.3.2 Class Imbalance . . . 35
7.3.3 Mislabelling . . . 36
7.3.4 Data Complexity . . . 37

8 Task 2: Transfer Learning **38**
8.1 Motivations . . . 38
8.2 Implementation: Caffe . . . 38
8.3 Experimentation . . . 39
8.3.1 Test Run . . . 39
8.3.2 Initialising Free Layers . . . 40
8.3.3 Freezing Backprop on various layers . . . 40
8.3.4 Parametric vs Non-parametric . . . 42

9 Task 3: Class Imbalance **44**
9.1 Motivations . . . 44
9.2 Implementation . . . 44
9.3 Experimentation . . . 44
9.3.1 Test Run . . . 45
9.3.2 Transfer Learning . . . 45
9.3.3 Batch Size . . . 45
9.3.4 Learning Rate . . . 47
9.3.5 Bayesian Cross Entropy Cost Function . . . 48
9.3.6 Under-Sampling . . . 50
9.3.7 Over-Sampling . . . 50
9.3.8 Test-time threshold . . . 52

10 Task 4: Conserving Spatial Information **53**
10.1 Motivations . . . 53
10.2 Implementation . . . 54
10.3 Experimentation . . . 54

11 Final Results **54**
11.1 Merging Classes . . . 54
11.2 Learning Rate . . . 54
11.3 Soil Risk Contamination Task . . . 55
11.4 Hatch Markings . . . 56

12 Conclusions and Future Work **58**

Thanks to my supervisors Jack Kelly for ... and Prof Murray Shanahan for ... Thanks to Razvan Ranca for wonderful strategic advice on which experiments to choose for the exploration of hypotheses, help with C++, musical and caffeine tricks for staying up all night, and much else. I look excitingly forward to the times ahead cofounding a machine learning startup together. Thanks to Alexandre de Brebisson for his great insights into convnet architecture that helped me make sense of experimental results, and the great conversations we had sharing our passion for deep learning. The Google+ Deep Learning community was perhaps the 2nd most precious resource, after arXiv and before stackoverflow. It provided a large and steady stream of wonderful resources, comments and guidance. It seemed like a revolutionary way to be in touch with the world's leading researchers such as Prof Yann LeCun and Prof Yoshua Bengio. Within the community, particular thanks go to Dr Soumith Chintala for his comments on transfer learning and Prof Francisco Zamora-Martinez for his on class imbalance.

1 Introduction

The background goes through the essential material regarding machine learning with feed-forward neural networks. Since this project was experimental from an early phase, the rest of the report is divided into chapters each of which go over the conceptual motivations, the design and the implementation of a main experiment.

Deep learning has proved its prowess at extracting high-level representations from high-dimensional sensory data in fields such as visual object recognition, information retrieval, natural language processing, and speech perception.

Contributions (beyond training accurate pipe weld classifiers for ControlPoint): 1. theory - intuition for ReLU vs sigmoid activation function (mention that since then you have found Quoc Le people at Google Brain thinking the same bit.ly/1ohsQcb) 2. empirics - class imbalance rule of thumb? 3. software - data provider for cuda convnet - class imbalance controller for cuda convnet - training time series plotter for cuda convnet - symlink lookup for caffe - class imbalance controller for caffe - f measure for caffe - data augmentation for caffe

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 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 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.

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 [11], 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 [11]. Since a contamination can cost up to 100,000 [11], 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 [11]. 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 gas or water access to a minimum, so the protocol can be a significant impediment. Automated, immediate classification would therefore bring strong benefits.

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

2.2 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 [9]:

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 $f : 2^{\mathcal{X}} \rightarrow 2^{\mathcal{Y}}$ 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 $f : \mathcal{X} \rightarrow \mathcal{Y}$ 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 [9]. 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.

2.3 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.

2.3.1 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	Redbox Count	Bluebox Count
Fitting Proximity	1,233	32
Inadequate Or Incorrect Clamping	1,401	83
Joint Misaligned	391	35
No Clamp Used	8,041	1,571
No Ground Sheet	30,015	5,541
No Insertion Depth Markings	17,667	897
No Visible Evidence Of Scraping Or Peeling	25,499	1,410
No Visible Hatch Markings	28,155	3,793
Other	251	103
Photo Does Not Show Enough Of Clamps	5,059	363
Photo Does Not Show Enough Of Scrape Zones	21,272	2,545
Soil Contamination High Risk	6,541	3
Soil Contamination Low Risk	10	N/A
Soil Contamination Risk	?	529
Unsuitable Photo	2	N/A
Unsuitable Scraping Or Peeling	2,125	292
Water Contamination High Risk	1,927	9
Water Contamination Low Risk	3	7
Water Contamination Risk	?	296
Perfect (no labels)	49,039	4,182

Table 2: Count of Redbox images with given label

2.3.2 Multi-Tagging

:

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

2.3.3 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 [17], 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 [19].

2.3.4 Small Dataset Size

Alex Krizhevsky's record-breaking CNN was trained on 1 million images [1]. Such a large dataset enabled the training of a 60-million parameter neural network, without leading to overfit. 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 [20], 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.

2.3.5 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 [1], [20], [6]), one can only expect a handful of them to contain either of the first two labels. Intuively, 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 [10], 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 $[0; 1]$.


PURITON
2
PIPE


F360
C20M.

Figure 2: left: a Redbox photo - right: a Bluebox photo

3 Deep Supervised Learning

3.1 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 $f^* : 2^{\mathcal{X}} \rightarrow 2^{\mathcal{Y}}$ 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 .

3.1.1 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 [12]. 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.

3.2 Deep Learning: Neurons and Depth

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).

3.2.1 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.

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

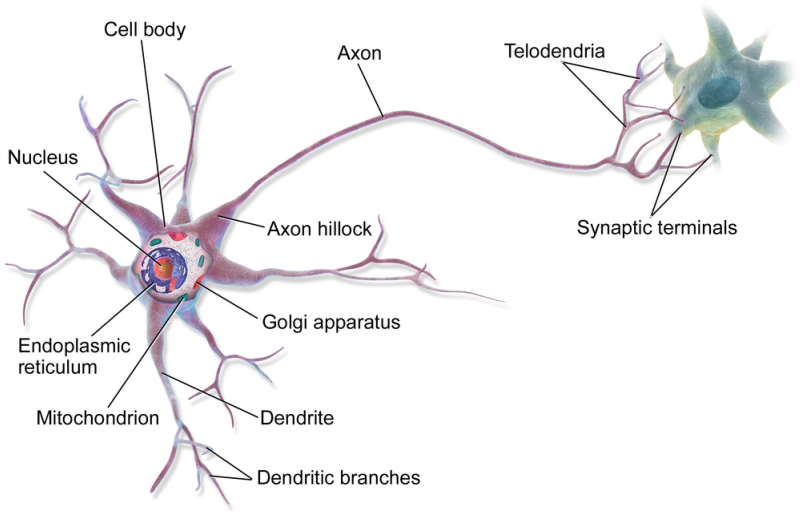


Figure 3: a multipolar biological neuron

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 [13]. 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).

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

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

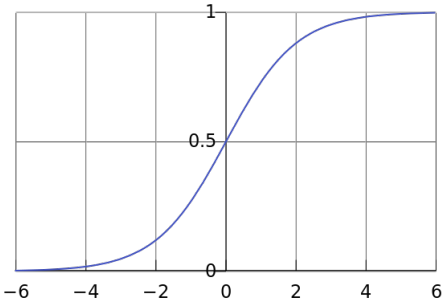


Figure 4: single-input logistic sigmoid neuron

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 [1]. Until now, this has more than offset the per-neuron information loss - and saturation risks - of the rectifier versus the sigmoid unit [2].

ReLU introduces a non-linearity with its angular point (a smooth approximation to it is the softplus $f(x) = \log(1 + e^x)$).

Explain also the no neighbouring cancellations in pooling.

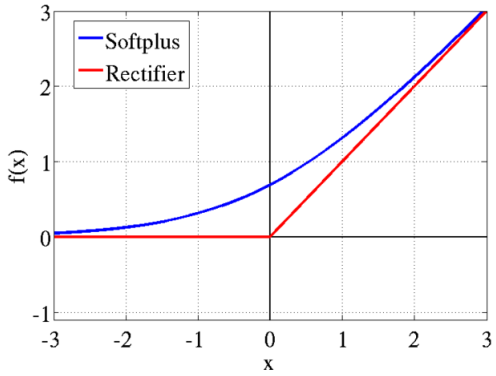


Figure 5: single-input rectified linear neuron

Softmax Neuron

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

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

intuition: Bishop textbook: "softmax function, as it represents a smoothed version of the max function because, if ak ? aj for all j ?= k, then p(Ck—x) ? 1, and p(Cj—x) ? 0."

3.2.2 Why Depth

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 [13]) illustrates this.

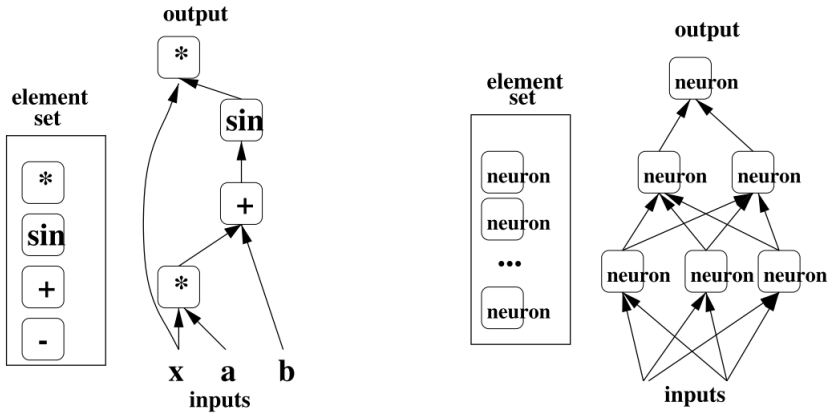


Figure 6: graphical representation of $y = x * sin(a * x + b)$ and of a feed-forward neural network

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 [13], 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 [13]. 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 [13]. 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.

Mathematically, it was proved in 1989 that MLPs are universal approximators [16]; hidden layers therefore increase the size of the function space, and solve the initial limitation faced by perceptrons.

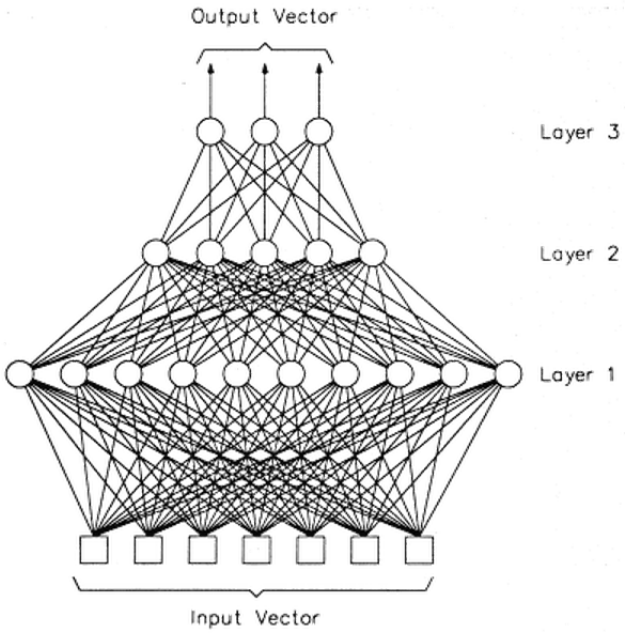


Figure 7: Multi Layer Perceptron with 2 hidden layers

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 class membership.

To make the case for deep architectures, consider a model with the same number of parameters but fewer layers (i.e. a greater number of neurons per layer). (Goodfellow et al 2013) [4] ran experiments to compare and found that depth is better: intuitively, if neurons are side by side, they cannot use the computation of their neighbour, whereas with depth, the neurons above can make use of the work done by the neurons below.

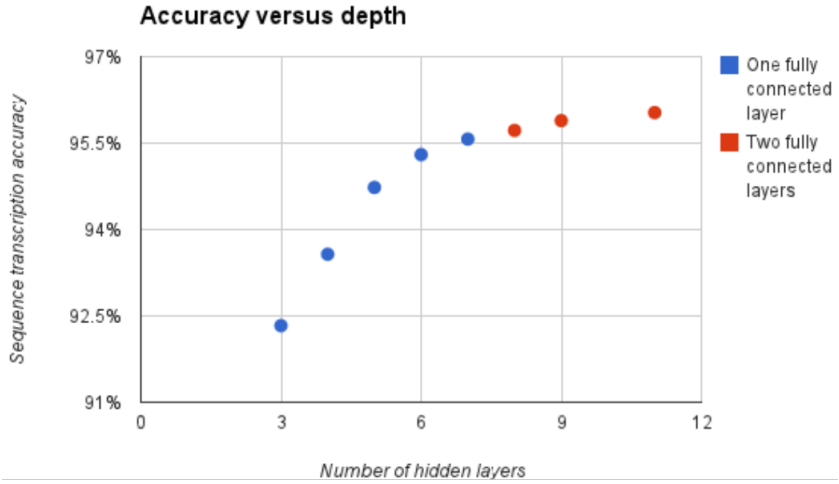


Figure 8: LeNet7 architecture: each square is a kernel

As a formalisation of the intuition given above, (Bengio 2009) [13] advances that having multiple layers creates a **distributed representation** of the data, rather than a **local representation**. Con-

sider for example the one-hot encoding of the integer set $\{0, .., n-1\}$, consisting of n-bit vectors where the i-th bit is 1 if the vector represents the integer i. This encoding is sparse, because every element of the set is represented by a vector of n-1 zeros and one 1. Moreover, it is local, because the k-th vector entry marks the presence of a feature that is present in only a single element, the integer k. On the other hand, the binary representation of the integers $0, .., n-1$ is a dense distributed representation, because it requires only $log_2(n)$ bits, and the k-th vector entry marks the presence of a feature that is shared by multiple elements of the set: whether or not not the integer is $\geq 2^k$. The binary representation is richer and more powerful than the one-hot representation because its features are not mutually exclusive: each class (i.e. each integer) is represented by a combination of features rather than a unique one. It is in this sense that the representation is distributed, and that the features are general.

(Bengio 2009) [13] details the belief that a deep architecture incites such representations, since, by having a neuron feed into multiple neurons of another hidden layer above, the feature it learns will be used by several feature detectors (or eventually, class detectors) above. The crucial benefit of this is generalisation: features learned from data in a specific region of the input space (a.k.a data space) are used in other regions of the input space, even if little to no data from this region is present to learn from. It is in this sense that the representation is general, not local.

However, it may seem "risky" to be imposing the use of features in regions of the input space that are unknown. "As usual in ML, there is no real free lunch" [18]. Indeed, one can think of this as imposing a prior on the function space to be searched, that is of the form: the observed data can be explained by a number of underlying factors which can be learned without seeing all of their configurations. If the prior is true for the classes we are learning, the gains are large. Successes in deep learning seem to indicate that this prior covers many tasks that humans deal with [18]. This prior is particularly powerful when learning from high dimensional data, where the curse of dimensionality means that a vast majority of the configurations are missing from the training set.

3.3 Backpropagation

Now that the architecture of a deep neural network has been motivated, 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.

3.3.1 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

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

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 [1]. Therefore, backpropagation scales linearly with the number of neurons.

3.3.2 Update Weight Values (with Gradient Descent)

The learning rule is given by:

$$w_{i,t+1} = w_{i,t+1} + \tau \cdot \frac{\partial E}{\partial w_{i,t}} \tag{8}$$

Visually, this means that weight values move in the direction that will (locally) reduce the error quickest, i.e. the direction of steepest (local) 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 training sessions are illustrated: the only difference is the initialisation of the (two) weights, and the minima attained in each case are not the same. This illustrates a strong shortcoming with backpropagation: parameter values can get stuck in poor local minima.

3.4 Cross Validation

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

[1] the bound on weight perturbations is no longer tight if we drop the assumption of fully connected layers

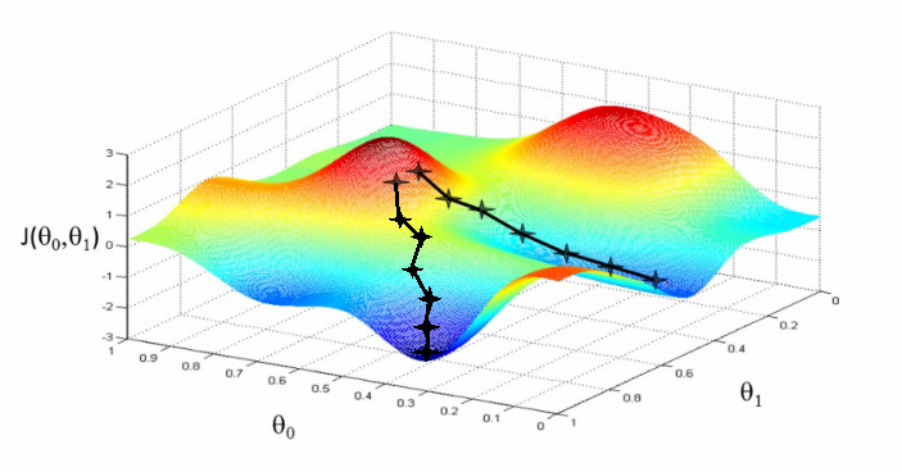


Figure 9: an error surface with poor local minima

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.

4 Deep Convolutional Neural Networks

A convolutional neural network is a deep feed-forward neural network with at least one convolutional layer. A convolutional layer differs from a traditional fully connected layer in that it imposes specific operations on the data before and after the data is fed through the activation functions. These specific operations are taken from the pre deep learning era of computer vision, and are detailed in this subsection.

Before diving into the details, a notable point to bear in mind is that these operations impose a prior on the underlying structure of the observed data: translation invariance of the features. Consider the following image of a geranium: a good (albeit complex) feature to classify this image would be the blue flower, regardless of the location of the blue flower. This feature could appear anywhere on the image; therefore, if the network can learn it, it should then sweep the entire image to look for it. It is in this sense that the pixel feature is "convolved" over the image.

Figure 10: LeNet7 architecture: each square is a kernel

The following subsection is an explanation of the computation that occurs in a convolutional layer. Users of Convolutional Neural Networks for classification tasks are sometimes accused of using them as a black box without understanding them; therefore, this section is crucial in establishing an in-depth understanding of how they function and why they might be so successful.

4.1 Operations in a convolutional layer

A[2] convolution layer has a pixel feature i.e. filter which is convolved over the entire image, followed by a non-linearity, followed by a spatial feature, optionally followed by a normalisation between feature responses. This structure is similar to hand-crafted features in computer vision such as SIFT and HOG [3]. The key difference is that each operation is learned, i.e. optimised to maximise performance on the training set.

[2]A large portion of the content from this subsection on explaining the operations in a convolutional layer is heavily inspired from Prof Robert Fergus's NIPS 2013 tutorial [7].

4.1.1 Pixel Feature

A[3] $k \times k$ pixel feature, also referred to as sliding kernel or sliding filter, is defined as a $k \times k$ matrix $\mathcal{W}$ that is applied to $k \times k$ windows $\mathcal{X}$ of the image by performing the matrix dot product $\sum_{i=0}^{k-1} \sum_{j=1}^{k} x_{ij} \cdot w_{ij}$. An example with $k = 3$ is shown below.

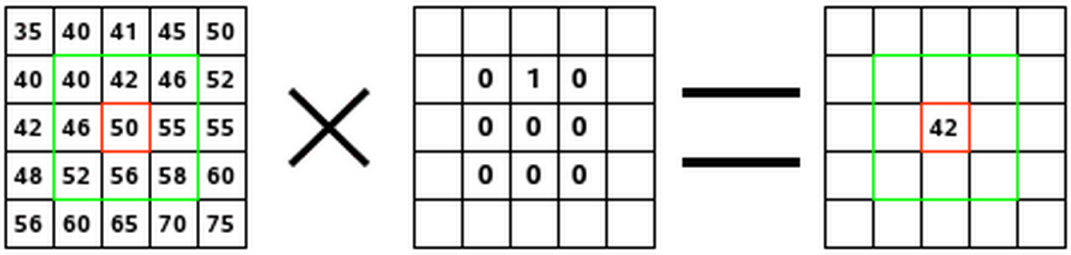


Figure 11: Selecting a pixel window and applying a kernel to it (source: convmatrix Gimp documentation)

Notice that this is equivalent to the vector product component of the generic neural network inputs combination $z = b + \sum_{i=0}^{n-1} x_i \cdot w_i$, where the pixel matrix representation of the entire image is flattened into the vector $\mathbf{x}$, the weights of the kernel are flattened into a portion of the weight vector $\mathbf{w}$, and all weights corresponding to a pixel that is not part of the window are set (and fixed) to zero. Fixing many weights to zero imposes a strong prior on the network and significantly reduces the function space, making it easier to search for good functions.

By convolving the kernel over the image, one obtains a feature response map (referred to as kernel map in computer vision jargon):

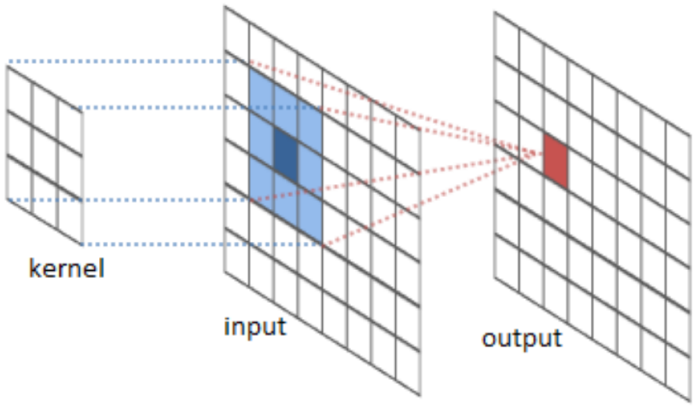


Figure 12: Producing a kernel map (source: River Trail documentation)

Another advantage of such pixel features is that one can make sense of what the weights represent i.e. what the network is learning. For example, consider the following image, sliding kernel and resulting kernel map:

A zero value is obtained when pixel $(1, 0)$ and pixel $(1, 1)$ have the same value, which is likely to be the case, hence why most of the kernel map is black. A high value is obtained when pixel $(1, 0)$ is of a much lower value than pixel $(1, 1)$, which occurs most often when the window is on a vertical edge separating a dark region to the left from a light region to the right. Therefore, an intuitive representation for the kernel matrix is the pixel window below, which has become a popular way of representing "deep learning [computer vision] features":

In order to impose convolution i.e. application of the same kernel over the entire image, the learning rule (8) given in section 3.3.2 is modified to:

[3]This subsection on explaining sliding kernels is heavily inspired from a blog post by Christopher Colah [8].

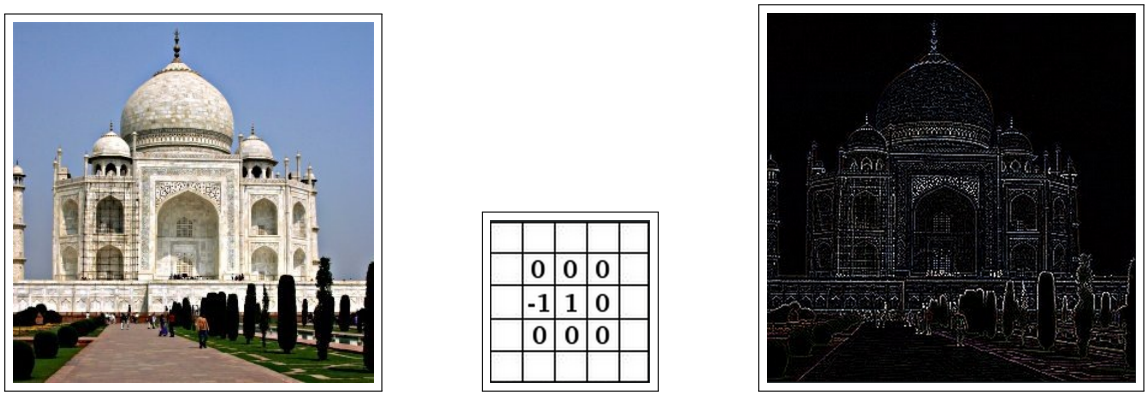
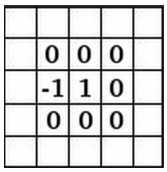
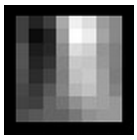


Figure 13: Image, kernel and resulting kernel map (source: convmatrix Gip documentation



(a) Kernel



(b) (Approximate) intuitive representation

$$w_{i,t+1} = w_{i,t+1} + \tau \cdot \sum_{j \in \mathcal{K}} \frac{\partial E}{\partial w_{j,t}} \tag{9}$$

Where $\mathcal{K}$ is the index set for all neurons intended to apply the kernel to a different location on the image. This learning rule ensures that the i-th weight of all such neurons are updated by the same value. At initialisation, each i-th weight is also set to the same value. The result is that the weight vectors for all these neurons are identical throughout training, i.e. the corresponding kernel is convolved over the entire image.

4.1.2 Non-linear Activation

Going to be applied to each output of the feature map independently, no interaction between the elements in the feature map. it's here because the kernel convolution is a linear operation, and we want non-linearity. the currently best performing known activation is the ReLU (extensively discussed in section 5 of the report). an important element is the bias, which in the case of the ReLU allows to choose the input threshold at which the neuron turns off. the bias is the same across every neuron of the feature map.

4.1.3 Pooling aka Spatial Feature

Take a $k \times k$ spatial neighbourhood of the feature map, and compute some summary statistic over it (average or max). The most common operations are average and max. Max is the currently best performing pooling operation, and this is theoretically backed by (Boureau et al 2010). The intuition behind the theoretical analysis can be illustrated by the following example: given a pixel feature, a 3×3 spatial neighbourhood to pool over and two different images, consider the case where image A has the feature present in a single location of the neighbourhood (activation output is 1 for this location, 0 elsewhere), whereas image B does not (activation outputs are 0 across all locations of the neighbourhood). Since the feature map seeks to discriminate inputs based on the presence of its feature, we would wish for the pooling operation to assign starkly different values to images A and B. With max pooling, A receives pooled value 1 and B receives pooled value 0; with average pooling, A

(a) 9 input images of the "human" class

(b) 9 corresponding kernel maps with similar pooling outputs

Figure 15: source: (Zeiler and Fergus 2013)

receives a mere $\frac{1}{9}$ and B receives 0. Therefore, max is better.

In general, a potential advantage of average over max is that it conserves more information; on the other hand, it will dilute an activation that is strong in a single location of the given neighbourhood and weak in the others. In the worst case, opposing activations will cancel each other out, and result in the pooled value as that for a neighbourhood with zero activations (however, note that this does not occur with ReLUs since they are not antisymmetric).

The key contribution of pooling is invariance to small transformations, since the same pooling output is obtained regardless of where in the neighbourhood the activations occur. At higher convolutional layers, the resulting invariance can be spectacular. For example, below are kernel maps for a certain feature of the 5-th convolutional layer of a net from (Zeiler and Fergus 2013) [14], computed for 9 different input images from the same class ("human"), which obtain similar pooled activation values. One may be struck by the fact that they receive similar pooled activation values, since one can tell that the similarities between the kernel maps are complex and would be difficult to describe in terms of "small transformations".

By stacking n convolutional layers with $k \times k$ pooling operations, the output of a pooling neuron on the n-th layer has a $k^n \times k^n$ receptive field over the image. Choosing the number of convolutional layers is therefore a matter of finding n such that k^{2n} equals the dimensionality of the input set, i.e. such that the highest convolutional layer can recognise complex features spanning the entire image, which is when total translation invariance is achieved.

4.1.4 Contrast Normalisation

Pick a single location of the output, have a little spatial neighbourhood around that, and perform the linear transformation on all the output values in this neighourhood that results in the best fit of a zero mean, unit standard deviation Gaussian. These transformations are used in image processing to handle images where contrast intensity varies across different areas of the image. The neighbourhood can be defined geographically over isolated kernel maps, or across kernel maps. This normalisation achieves a rescaling of the features at each layer. For example, if in a certain neighbourhood output values are all low and in another neighbourhood all output values are high, this may be due to photographical factors (e.g. different expositions to light) which have nothing to do with the underlying objects of interest; local contrast normalisation will pull up the values of the low value neighbourhood and reduce those of the high value neighbourhood to put both neighbourhoods on an equal footing for the next layer to receive unbiased data.

4.2 AlexNet Architecture

The architecture from (Krizhevsky et al 2012)'s record-breaking image classification model - often referred to in the deep learning literature as AlexNet - is summarised below. It was obtained by manual optimisation of hyperparameters.

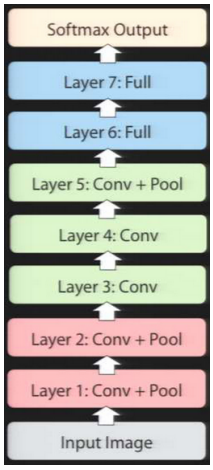
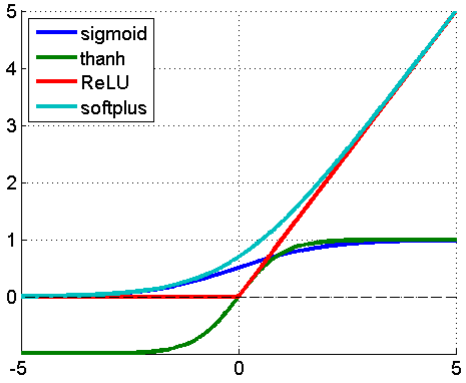


Figure 16: Architecture of CNN from (Krizhevsky et al 2012)

4.3 Dropout

4.4 Data augmentation



(a) Activation functions

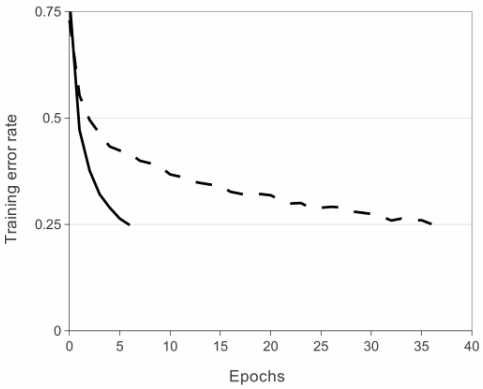


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

(b) source: Krizhevsky et al 2012

5 Analysis 1: ReLU Activation

5.1 Motivations

Differentiable, symmetric, bounded activation functions that introduce a non-linearity smoothly over the entire input domain were the activation units of choice in deep feed-forward neural networks, until Rectified Linear Units were used for CNNs in 2012 [1] and found to outperform their counterparts. They make the observations that $f(x) = \tanh x = \frac{1-e^{-2x}}{1+e^{-2x}}$ or $f(x) = (1 + exp^{-x})^{-1}$ "saturating non-linearities" are slower to train than the $f(x) = max(0, x)$ "non-saturating non-linearity", and report an "accelerated ability to fit the training set", but do not provide any explanations.

The importance attributed to non-linearity of activation functions is due to the function space that is spanned by composing such functions as deep neural networks do. If activation functions are linear, the overall network is linear too, and as seen in section 3.2.2, such networks are greatly limited in what they can learn. Therefore, one may wonder whether or how the function space is reduced by using an activation function such as the ReLU that is non-linear only in the neighbourhood of 0.

(Glorot, Bordes and Bengio 2013) [2] explain that "the only non-linearity in the network comes from the path selection associated with individual neurons being fired or not. [...] We can therefore see the model as an *exponential number of linear models that share parameters*". Since for a certain range of inputs, a ReLU will not fire, one can view the network as selecting different subsets of itself based on the input. Each subset is a linear function, but two inputs which are very close (in the sense of euclidean distance) in the input space might (de)activate different neurons, and therefore be processed by different (linear) functions that output completely different results. In this sense, non-linearity is achieved.

(Glorot, Bordes and Bengio 2013) also remark that "because of [the ReLU's] linearity, gradients flow well on the active paths of neurons (there is no gradient vanishing effect due to activation non-linearities of sigmoid or tanh units)", but no explanations of enhanced flow or vanishing gradient are given (instead, the paper focuses on the benefits of sparse representations and their greater resemblance to neuron activations in the brain). Another empirical finding of the paper is that "deep rectifier networks can reach their best performance without requiring any unsupervised pretraining". This section of the report is a simple mathematical analysis of backpropagation that conveniently provides an explanation for these points as well as (Krizhevsky et al 2012)'s observations. In the context of building pipe weld classifiers, the motivation for doing is to understand the mechanics of

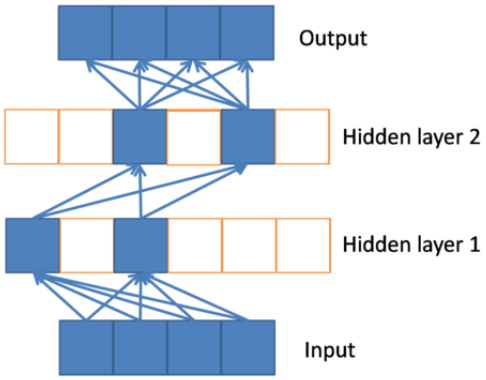


Figure 18: ReLU path selection (source: Glorot et al 2013)

training for the CNNs to be subsequently used.

5.2 Mathematical Analysis

Recall from section 3.3 that the model is trained with backpropagation: each of the weights w are adjusted by $\tau \cdot \frac{\partial E}{\partial w}$. The choice of activation function modifies $\frac{\partial E}{\partial w}$; this section looks at how ReLU does so compared to sigmoid or tanh.

5.2.1 How the Gradient Propagates

It may be useful for intuition to think of $\frac{\partial E}{\partial w}$ in the context of the gradient travelling through the network. 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
- ψ, the activation function used – therefore $y_j = \psi(z_j)$

The backpropagation algorithm can be formulated as a set of rules for propagating the gradient through the network:

– 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 along an edge** w_{ij}: $grad \leftarrow grad \cdot w_{ij}$

– to **stop at an edge** w_{ij}: $grad \leftarrow grad \cdot y_i$

5.2.2 An Example

Given the figure above:

– for $\frac{\partial E}{\partial w_{31}}$: initialise, propagate through y_1, then stop at w_{31}: $\mathcal{C}'(y_1) \cdot \psi'(z_1) \cdot y_3$

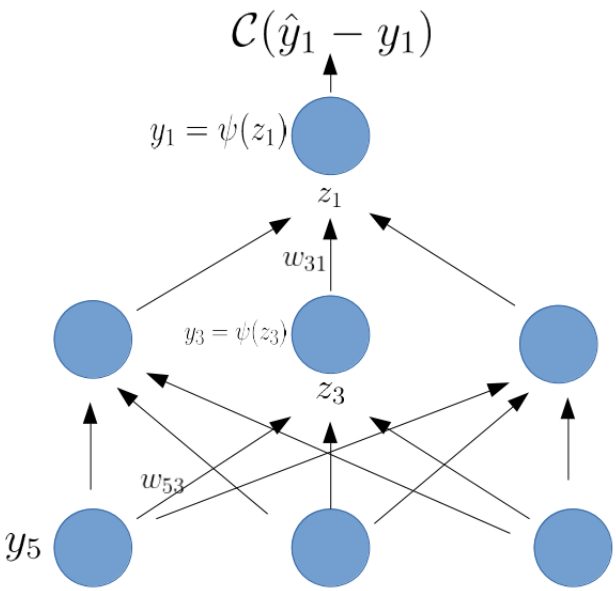


Figure 19: $\mathbb{R}^3 \to \mathbb{R}$ MLP with 1 hidden layer

– for $\frac{\partial E}{\partial w_{53}}$: initialise, propagate through y_1, then along w_{53}, then stop at w_{53}:
$\mathcal{C}'(y_1) \cdot \psi'(z_1) \cdot w_{31} \cdot \psi'(z_3) \cdot y_5$

Intuitively, the partial derivative with respect to a weight can be roughly seen as the product of the partial derivative of every component along the path from the weight to the output unit [4].

5.2.3 Vanishing Gradient

Notice that $\psi'(z_1)$ is a factor in both partial derivatives. Now, consider the derivatives of the tanh and sigmoid functions:

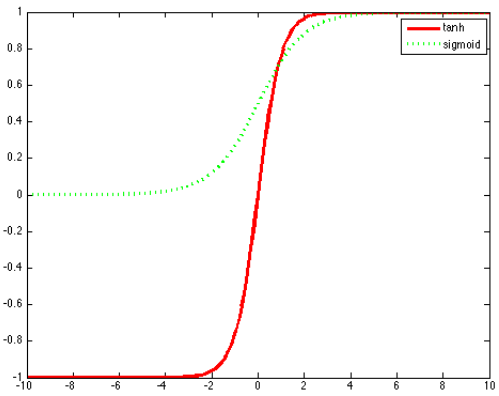
$$tan'(x) = 1 - tan^2(x) \tag{10}$$

$$sigmoid'(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

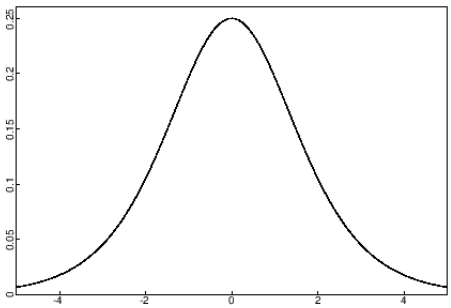
The formulae do not lend themselves to intuition, but their graphical representation given below does [5]. Such functions are called "saturating" by (Krizhevsky et al 2012) because, when the input is high in absolute value, the derivative approaches zero. In the context of backpropagation, this means that a tanh or sigmoid unit that is "heavily activated" during the forward pass will strongly reduce the gradient as it propagates through it during the backward pass. Moreover, this will affect the partial derivative of every weight that lies behind the unit in some input-output path. As a result, during training it becomes slow to alter any such weight when the unit's weights are on average high in absolute value. This difficulty increases with the depth of the network, since in order to reach a weight that is low down in the network, the gradient must propagate through a higher number of units, so its probability of vanishing increases. This is consistent with empirical findings [13].

[4] this only goes for this simple case where we have one hidden layer and one output node. That is because, if we consider all paths from an input node to an output node, every edge exists in exactly one path. However, if we had more output nodes or more hidden layers, there would exist edges belonging to several input-output paths. In this case, the partial derivative would be the sum across all paths from the weight to some output unit.

[5] both are the same up to a scaling factor, therefore only one is given.



(a) tanh and sigmoid functions



(b) derivative fot he tanh and sigmoid

One may argue that a unit which has learned a feature that is useful for the task is one that is heavily activated for certain inputs. Therefore, it is a good thing that it becomes rigid to change. However, traditional training of deep neural networks occurs by first randomly initialising the weights. At first, it is therefore unlikely that a unit is heavily activated because it is a detector for a meaningful feature. If the network is built on saturating activations, such non-meaningful features will be harder to get rid of.

5.2.4 Impact of the ReLU

On the other hand, the derivative of the ReLU is given by:

$$ReLU'(x) = \begin{cases} 1 & \text{if } x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Which means that for any positive input, the gradient will *not be modified* when propagating through the unit. It is in this sense that "gradients flow well on the active paths of neurons". On the other hand, for any negative input, the gradient is brought to zero i.e. halted from propagating altogether. This could be cause for concern, because if the weights of a ReLU are initialised to values that, given the distribution of the input, always lead to a zero activation, then there is no way of modifying them, and the neuron is effectively a useless component of the network. When training, one should therefore take care to choose a sufficiently high positive value for the neuron's initial bias [6]. On the other hand, halting the gradient in this regard has the intuitive benefit of reducing the areas of the network to which it propagates, and focusing it on the areas responsible for the error. If one goes back to the vision of a deep sparse rectifier network as an exponential number of linear models, gradient halting enables the training of a smaller network at every iteration.

Since vanishing gradient makes it slow to get rid of non-meaningful features that receive heavy activations, this provides an explanation for why "deep sparse rectifier networks can reach their best performance without requiring any unsupervised pre-training", which initialises the weights of the network to features capable of reconstructing the data. It also explains (Krizhevsky et al 2012)'s observations that ReLU networks train faster. By providing a clear mathematical explanation to the main empirical findings regarding deep sparse rectifier networks, this report would argue that the ability of ReLU to enable gradients to "flow well" is the key reason for their success.

One final remark concerning ReLUs is that they reduce the computation in both forward and backward passes by turning off a number of neurons and by not interfering with the gradient as it

[6] This makes the bias b in $z = b + \langle x, w \rangle$ a crucial element of a neural network, despite them being conventionally excluded from graphical representations

propagates through a unit.

6 Analysis 2: Early Stopping

Early stopping is a form of regularisation to prevent overfitting. It makes use of cross validation: separating the dataset into a training set, a validation set and a test set. The model's parameters are optimised with respect to the training set (i.e. backpropagation is run on the training set only), but are considered optimal at the point which minimises the model's error on the validation set.

But maybe be taken for granted but is in fact shocking is that when learning a DNN with backpropagation and first order gradient descent, the time series of the validation error always assumes a strictly convex curve (bar some noise). In other words, the validation error has a unique minimum, is strictly decreasing before it, and strictly increasing afterwards. It presents the simplest most convenient shape possible. This is what enables the straightforward strategy of early stopping: (write this as a little algorithm to make it look more classy) while validation error is decreasing, train.

One may wonder why the validation error time series is shaped so: why is it not more random? A theoretical and intuitive answer may lie in the optimisation algorithm that is used, first order gradient descent.

Gradient Descent Gradient descent consists in finding the weights of the model that locally minimise the model's error. Therefore, if we consider $E : \mathbb{R}^n -> \mathbb{R}$ to be the error function (also referred to as surface) which to a set of n parameters associates the model's error, then the problem can be reformulated as finding the minimum of E.

(erase this?) In the optimisation literature, E is assumed to be strictly convex, because this is the necessary and sufficient condition for gradient descent to be a globally valid algorithm.

For the sake of providing mathematical intuition as to why early stopping and validation error behave the way they do, assume that E is of the form $E(x) = \frac{1}{2}x^T Ax - b^T x$, where A is an $n \times n$ positive definite matrix and b is a vector. (For $w_k = (\sum_{t=0}^{T}(I - A')^k)b'$ where

Combine your intuition with the stuff from that conference, about gradient descent and extra polynomial order.

I've been thinking about it in terms of early stopping. early stopping seems really neat in that it first learns patterns that generalise and then eventually learns patterns that don't generalise, which is when we stop training. I was wondering why it works so nicely, and thought this inverse approximation formulation of grad descent makes it look like you're adding ever higher order polynomials as you go along. so maybe what's happening is that you're not learning first the patterns that generalise per se, but rather the simplest patterns (that can be fitted with low order polynomial), and as you go along, increasingly complex ones. if you have a large enough dataset the patterns that don't generalise are going to be really complex and require really high order polynomials to fit them, hence why overfit only takes place later.

7 Task 1: Generic Clamp Detection

7.1 Motivations

Clamp detection was suggested by ControlPoint as a simple test run for training a CNN on their dataset. The detection task in itself was deemed simple since clamps are large objects which would be easy to see. However, four significant obstacles arose: non-converging error rates, class imbalance, mislabelling and data complexity.

7.2 Implementation: Cuda-Convnet

Used AlexNet. No need to describe its achitecture here, this is already done above.

7.2.1 Cuda-Convnet: An Out-of-the-box API

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. In order to do so, data batching scripts were written for turning raw .jpg images and separate .dat files into pickled python objects each containing 128 stacked jpg values in matrix format with RGB values split across, and a dictionary of labels and metadata for the network.

Wrote scripts to pre-process the data because Cuda-Convnet and noccn don't provide everything. Had to write scripts to:

- get all labels
- visually inspect random samples of the data
- leave out images tagged as poor
- write the data in the xml format that cuda-convnet's batching API can read
- potentially treat unlabelled images as an extra class
- potentially leave out classes
- potentially merge classes
- randomly remove images to reach given class balance ratio
- unit tests for all of the above

The scripts are written in python and the code is 800 lines.

7.2.2 Hardware: NVidia GeForce GTX 780

CUDA-enabled GPU. How many teraflops? Which operations are parallelised? First a GeForce GTX 780 with 4GB RAM.

7.3 Discovery

Also mention that unsupervised learning was considered to deal with the mis-labelled data, the reasons for why this was put aside (discussions with ControlPoint), but that it would make for interesting further research? Maybe it should be renamed to summary, and be included as final subsubsection of every implementation subsection?

Several weeks into training, it was discovered that in the case of tapping-T joints, for the Redbox images, the glint of a slim portion of a clamp is sufficient to judge it present.

Worse still, sometimes the presence of clamps "doesn't count": these are cases for which the purpose of the clamp is other than to secure the welding. Therefore, if such a clamp is present, but the clamp that serves to secure the weld is absent, then the image is assigned the "No Clamps" label. For example, bottom left, a clamp can clearly be seen, but it's not a weld clamp. So this image should have a "No Clamps" flag raised (sadly, it doesn't). Bottom right: the thin metallic clamp that is fastened on the vertical pipe is not the clamp we're interested in. The glint from the thin metallic rod going along the thick, horizontal pipe tells us that a tapping-T clamp is present, even though that clamp is hidden underneath the pipe.

This makes learning very difficult, as a class can be pictured as a disjoint set of subclasses, some of which fine-grained, and that one subclass could trigger a false positive of another class.

Figure 21: The clamp wraps around under the pipe - the glint of a metal rod gives it away

Figure 22: This clamp is not a weld clamp

Figure 23: The clamp on the vertical rod is not a weld clamp

The solution was to train a classifier capable of recognising the type of joint having been welded, using the small subset of data which contained joint-type labels, in order to try and tag the remaining dataset. The advantage of knowing the joint-type is that we can isolate clamp types, and train a classifier on s single type of clamp. That way, the learning task is simplified, and one is able to focus on the other challenges.

7.3.1 Non-Converging Error Rates

The training produced confusing results. LOOK THROUGH ENTIRE DOCUMENT, when talking about this training, the classification validation error achieved was 11%, not 40%! It may also be good to get a test error.

They were trained on the clamp detection task using the Redbox data only. The purpose of these models is to set a benchmark for the difficulty of the classification task, by taking several out-of-the-box network configurations: namely, those found on open source projects or detailed in papers [1], [20], [6].

The classification task is a challenging one: indeed, none of the training sessions have shown any signs of parameter convergence, as the below plot of the training and validation errors over batch

iterations can show. These results are very confusing: backpropagation is guaranteed to converge [cite! go into more detail about this! include the proof in your background!], and yet both training and validation errors seem stuck in volatile periodicity.

You should also justify why you are not bothering with the test error! Because test error optimum is obtained at validation error optimum; test error is merely a truly impartial measure of performance. A bunch of research contents itself with validation error when merely iterating through various models.

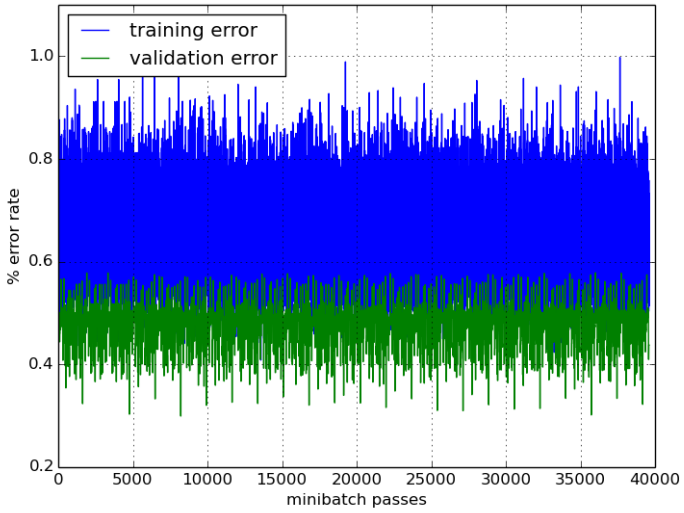


Figure 24: Test Run Training Results

The error being shown is the negative of the log probability of the likelihood function:

$$-\frac{1}{n}\sum_{i=1}^{n} ln(f(W|x_i)) \tag{13}$$

Where f is the learned function, a.k.a the model i.e. the neural network. So in this case it is the cross entropy I think. (Verify this, provide intuition).

A number of potential explanations for the volatile periodic error rates were considered:

- The learning rates are too high (the minimum gets 'overshot' even when the direction of partial derivatives is correct)
- The dropout rate is too high (dropout is a technique which consists in randomly dropping out neurons from the net at every iteration to prevent overfit - but it also means that a different model is tested every time)
- The number of parameters is too high (the out-of-the-box implementation contains millions of parameters, which is more than than the number of training cases, so collinearities between the parameters cannot even be broken, and most of them are rendered useless)
- Class imbalance (not enough information about the clampless classes to be able to learn feature for them)
- Mis-labeled data (at the time of human tagging of these images, tags were left out) ...
- The error rates are not computed correctly ...

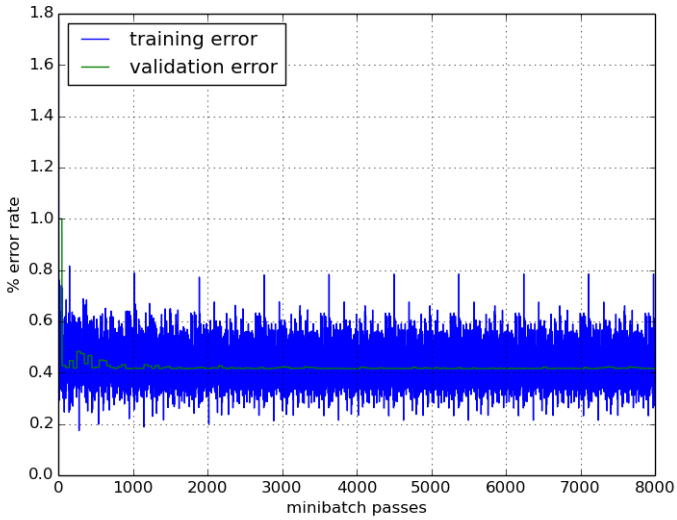


Figure 25: test and validation error rates for clamp detection after fixing a large test range

Increase Test Error Precision By computing the test error on a large and unchanging sample of images, one obtains more precise estimates, and convergence can clearly be seen.

I trained AlexNet on 100k images of 3 different classes with class imbalance (1 class takes up 90% of data), 40% mis-labelling, and a very small learning rate (0.0001). The emerging periodicity of the training error corresponds to the number of batches, so it looks like the parameters have stopped changing, ie the network has reached a local minimum (flesh out the intuition for why periodicity implies convergence, just going around in the same circle). Why does no overfitting take place? There is not much data compared to capacity of the network, there is a weight decay of 0.0005, there is no dropout. Since overfitting hasn't been reached, maybe this is a poor local minimum?

Francis Quintal Lauzon: I would tend to think learning was indeed stuck, though I would not go so far as to suggest this is caused by a local minimum. Indeed, I have sometimes seen long plateaus followed by sudden steep drop in error. One way I see this is learning with a given learning rate and reducing the learning rate when validation error (or rather, a smoothed version of the validation error) stops going down. This suggest than rather being stuck at a local minima, learning is simply blundering around some minimum, may be because of a combination of the local shape of the cost surface and learning steps too large.

Plateau hypothesis: on the image I show only 10 epochs, but I trained it for 44 epochs and the periodicity extends all the way. When you were experiencing plateaus, did they stretch over more than 1 epoch? because to me, this periodicity suggests roughly the same gradients are repetitively being computed at the same locations on the error surface, so no good keeping on going.

Learning rate hypothesis: so you are suggesting that the minimum keeps getting overshot. But 0.0001 learning rate is already very small no?

I guess another possibility is that I'm zigzagging in a very tight bowl, but seems unlikely it would go on for as long as 40 epochs?

Francis Quintal Lauzon: I trained on much larger (though highly correlated) dataset so I never trained for 44 epochs. I did use learning rate of the same magnitude you are using and still, reducing it after a plateau does help (in my experience). If you are using momentum, you also might want to reset any momentum to zero after facing a plateau (this might helps as well).

Another striking aspect is how volatile the training error is, even when the test error is stable. That the training error is computed correctly was checked by training an out-of-the-box net on a

well-documented task: MNIST. Therefore, it was established that in this setup, the training error is indeed volatile and periodic despite the validation error converging, and that an explanation must exist. This could be interpreted as the network's parameters having converged to values that make it good on certain batches, and bad on others. What is curious is that this would suggest that the contents of some batches are very different from one another (unless one considers the paper "Intriguing Properties of Neural Networks"). Therefore, it might be interesting to look at these batches in detail.

Alter Momentum What is momentum for?

the intuition (maybe wrong): if a smaller learning rate and zero momentum could help you deal with a plateau, it means that moving in very small steps is better. but if a plateau is a continuous flat surface, then surely a smaller step will take you longer to reach the other side? on the other hand, if you're in a very tight and stretched bowl, or if there's a very narrow ditch surrounded by a plateau, then the smaller step will help?

Increase Is that used to get past the local minimum? Or is it used to rush past a plateau?

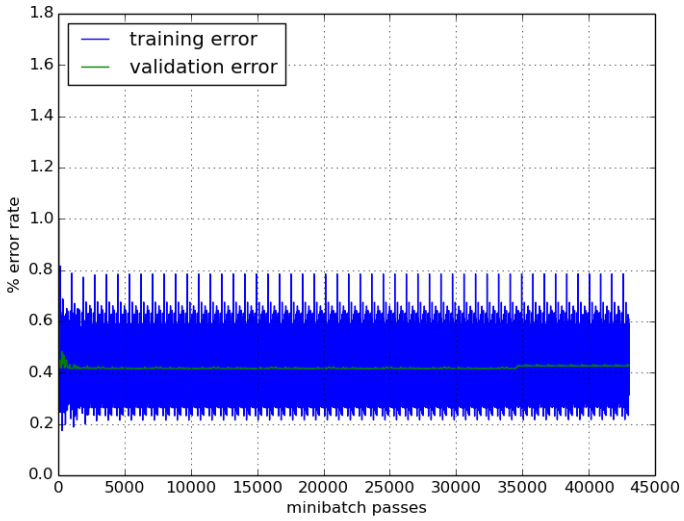


Figure 26: test and validation error rates for clamp detection after raising momentum

Barely noticeable change in train error: still periodic, slightly different shape of period. Slight increase in test error though. Can interpret that weight updates are slightly less optimal since they don't follow the direction given by the gradients, since there's this extra momentum factor, which is bigger.

Decrease Because Francis Quintal Lauzon says so. But does he mean to reduce it once you're done with the plateau?

The training error decreases a little on average (would be good to assert this statistically). Once again, because momentum is not distorting direction of descent.

The test error stops to jitter completely. So the tiny pulses observed during convergence (i.e. between 5000 batches and 35000) were caused by momentum. I guess it's the result of going upslope a bit just in case there's a better minimum nearby. Could be interesting to put a stupidly high momentum like 1.5 to double check that jittering indeed gets even bigger. Who knows, this might settle the network in another minimum.

Reduce Learning Rate $train_output.txt$ contains data to plot this. Good to put it in to show that you have checked everything meticulously, and to show that it's evidence for the hypothesis of stuck in corner solution.

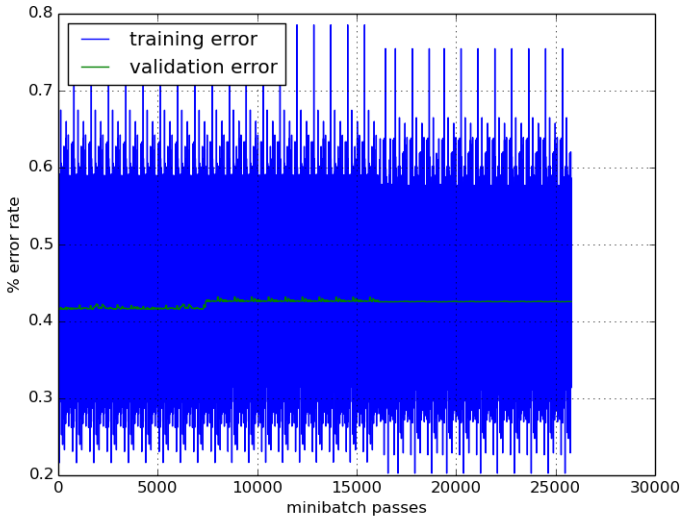


Figure 27: test and validation error rates for clamp detection after setting momentum to zero

7.3.2 Class Imbalance

Here you just need to show that there is class imbalance.

Stuck in Sampling-Induced, "fake" Corner Minimum I trained a neural network to detect clamps in the images. It very quickly converged to logprob 0.4, i.e. 10% classification error, as can be seen in attachment.

But what is surprising is that the training error remains in the neighbourhood of the test error, without converging to zero and without the network overfitting (should converge to zero since neural networks are 'good' universal approximators).

I played around with momentum and the learning rate, which typically help to deal with plateaus, tight bowls, and poor local minima. (you can see it on the graph, the test error jitters a bit more, and then becomes completely constant, and the amplitude of the training error varies a bit). It didn't help.

The reason for why the network converges quickly to logprob 0.4 minimum, and stays there, without ever overfitting, is because severe class imbalance has introduced a "fake" minimum in a "corner" of the error surface. (By error surface, I mean error on the z axis, and parameter values on all the other axes). This local minimum is very hard to get out of because it's quite deep (you get 10% error rate - it might even be the global minimum), and it's far away from where the "real" minima are.

To find evidence to support this claim, we can use the periodicity in the training error to verify the claim that the network is stuck in this deep, fake, corner minimum: take the batches that consistently score very high train error, take those that consistently score very low, and look at the images. Between the top scoring ones and the low scoring ones, is there a significant different in the proportion of "clamp detected" images? Or is there a significant difference in something else (eg unsuitable photos, mislabelling)?

Intuitively, the network goes like "wait a second, if I output 'clamp detected' every time, then I get 10% error, that's awesome, let's just do that."

sutop is a list of the absolute best performing batches in terms of training error top is a list of top performing worst is a list of worst suworst is a list of absolute worst top performing batches: 280, 543, (195, 185, 177, 157, 156, 147, 143, 82, 67, 53, 19, 18 and others) worst performing batches: 152, (302, 162, 105, 87, 60, 39, 736, 710, 643, 631 and others) 280, 543 achieve 0.18-0.22, the others are in the 0.20s. 152 achieves 0.7-0.8 error, the others achieve in the 0.60s.

Training Error	"Clamps Detected"	"No Clamps"	"Clamps Not Sufficiently Visible"
18-22%	0.96484375	0.0234375	0.01171875
20-29%	0.94161184	0.03371711	0.02467105
60-69%	0.81534091	0.10795455	0.07670455
70-82%	0.765625	0.15625	0.078125

Table 3: Class Proportions Across Batches that Score Different Training Errors

Would be nice to add the plot of the errors and point to which batches the spikes correspond to.

This suggests that the only thing that determines error is proportion of non-"clamp detected" cases. this is strong evidence that the net has learned to output "clamp detected" all the time. More evidence can be obtained by looking at which mis-classifications occur: ten mis-classification results such as the one below were processed, and in all cases the mis-classifications were the same: "Clamp Detected" was wrongly output, with maximum confidence.

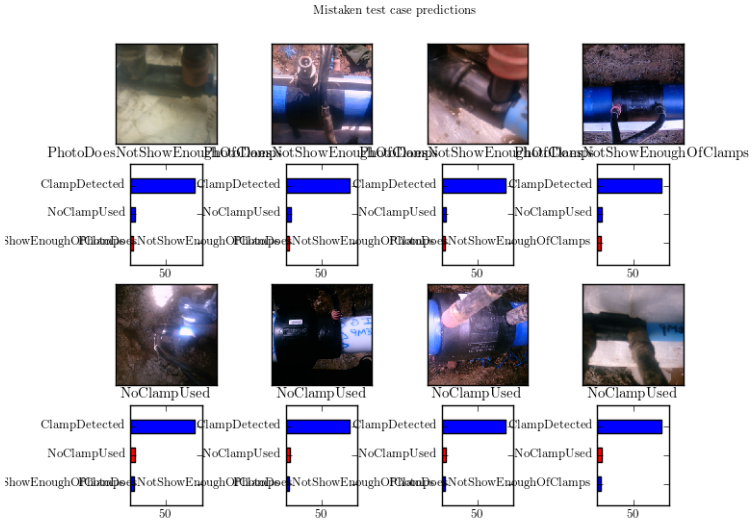


Figure 28: test and validation error rates for clamp detection after setting momentum to zero

It can also be interesting to notice that no meaningful features are learned: for example, the filters learned at the lowest convolutional layer in optimised networks usually resemble edge or contrast detectors: in this case, they look noisy.

7.3.3 Mislabelling

The discovery of this is detailed somewhere above, fetch it.

Only Bluebox images were chosen for the rest of the project, in order to limit the impact of mis-labelling and photographically poor images on training, and to limit the number of possible explanations for poor performance if that were to occur. According to ControlPoint the mis-labelling rate greatly decreased through time; since Bluebox images are the most recent, one would hope the mis-labelling rate would be lowest. However, this heavily reduced the amount of training data, requiring transfer learning to come into play. Still, just as a benchmark – for the record – randomly initialised AlexNets were trained from the ground up.

It is likely that recent Redbox images also have good labelling. By ranking them chronologically, it may have been possible to determine the optimal cut-off date by using a network trained on Bluebox and testing it on Redbox data, with the cutoff date going steadily back. One would expect test error

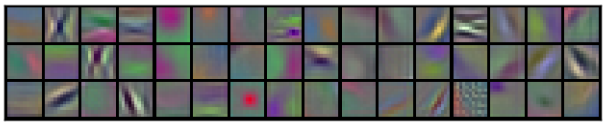


Figure 29: filters learned from a successfully optimised lowest convolutional layer

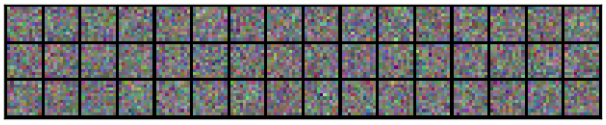


Figure 30: filters learned at lowest convolutional layer of this network

to rise as the cutoff date is brought earlier ie as the proportion of mislabelled images in the test set rises.

7.3.4 Data Complexity

Simply provide examples that describe. I think this has also been written up somewhere, if not, it is in the emails.

8 Task 2: Transfer Learning

8.1 Motivations

Because training set restricted. The objective here is to optimise transfer learning. 2 aspects: re-initialise weights, freeze backprop.

8.2 Implementation: Caffe

Caffe is a framework for convolutional neural network algorithms, created by Yangqing Jia, and is in active development by the Berkeley Vision and Learning Center. It was chosen in order to implement transfer learning, or pretraining. Not like standard unsupervised pretraining, since this time, training was supervised, but on another dataset. So end result is pretty much the same, though it would be interesting to compare the two approaches. (unsupervised pretraining on ImageNet vs supervised pretraining on ImageNet).

The network pretrained on ImageNet is licensed for academic research, and is for non-commercial use only. Therefore, if ControlPoint wishes to make commercial use of a network whose weights were initialised from it, it would have to pretrain its own net on a dataset on which there are no such licensing restrictions.

Caffe Reference ImageNet Model: Our reference implementation of an ImageNet model trained on ILSVRC-2012 is the iteration 310,000 snapshot. The best validation performance during training was iteration 313,000 with validation accuracy 57.412% and loss 1.82328. This model obtains a top-1 accuracy 57.4% and a top-5 accuracy 80.4% on the validation set. [22]

Contributed to Caffe by making it more space efficient: instead of the data being copied to the task directory, it is symlinked. The batching code tests for whether files are symlinks, and follows the link if that is the case. Caffe has a low-level tier written in C++ and a high-level tier written in Python; in this case, the modification was in C++. Note that before, it could get really inefficient to make copies of the data for every task. (One could use the same directory for several tasks, but if we want to keep control of class imbalance, we need flexibility in taking subsets of the training set. Therefore, one usually ends up requiring multiple different data directories, which makes the case for symlinked data all the more compelling.)

Caffe benchmark: https://github.com/soumith/convnet-benchmarks

leveldb More efficient implementation that does not require to create batches. Describe a bit what is going on under the hood, it seems that lookup tables are created for each image so that they can be dynamically pulled in (in constant time) to create batches on the fly. But find out more, it's cool.

Class Imbalance Solver Note that class imbalance ratio i.e. size of largest class relative to smallest class is not the right metric to consider. The right metric to consider is the proportion of the largest class, since this is what provides a bad/fake local minimum. This may not seem to be significant but it can be. Consider the 3 class case where we have 10, 100, 500 images for each respective class. Class imbalance ratio is 0.05, and proportion of largest class is approximately 0.82. If we wanted to attain a class imbalance ratio of 0.2, no class would be permitted to have more than 5*10=50 images, and we would be left with a training set of size 110. On the other hand, if we wanted to attain a proportion of largest class of 0.8, we would merely have to remove 12 images from the largest class, and we would be left with a training set of size 598. Interestingly, in the two class case, optimising with respect to either of the two metrics is equivalent.

Ok, but do we care about this for this project? Are we ever going to be training multi-class classifiers? Yes, if we think that it may help training to distinguish multiple cases. Intuitively, if two classes contain similar semantic content (e.g. inadequate clamp fitting and no clamp detected both involve clamps), then it could be better to train a single classifier on a 3-class task rather than 2 binary classifiers, because in the latter case, we lose potentially useful information that the images contain about clamps. In other words, to detect whether clamps are fitted properly, it helps to know what a clamp looks like, i.e. it helps to know which images do and don't have clamps.

t: target bad min N: total number of current images n: number of majority class images k: number of majority class images to kill

when n-k is optimal, we have t = (n-k)/(N-k) ¡=¿ t*N - t*k = n - k ¡=¿ (1-t)*k = n - t*N ¡=¿ k = (n-t*N)/(1-t)

so solution is to randomly delete $n - n^*$ images from the majority class.

8.3 Experimentation

This section is a fucking mess. To bring order, you might have to train nets again. Structure should be as follows: - pick a task: clampdet (easy) or hatch markings (hard) - learn on top of conv feature space of fc feature space - freeze backprop 1/2/3/4/5/6/7 layers - import weights or re-initialise them

8.3.1 Test Run

This should just be the simple case of transfer learning (no weight re-initialisation, backprop everywhere) vs no transfer learning.

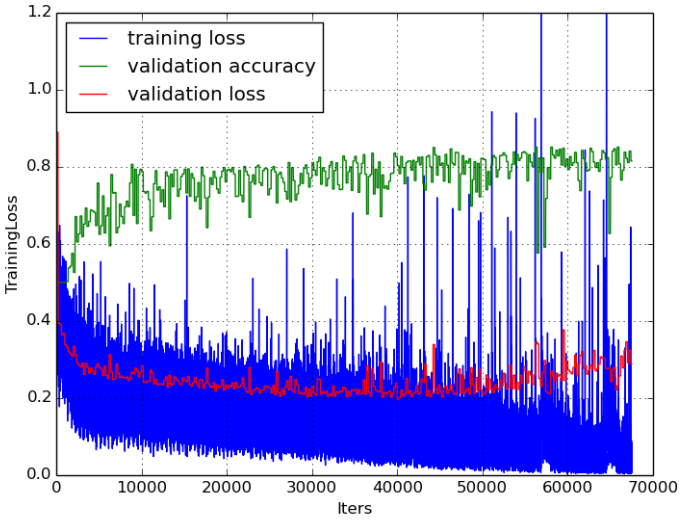


Figure 31: Without transfer learning

Without transfer learning, impossible to train. With transfer learning, it is! (at least not with this task). The conclusion is that initialisation of the parameters is very important, even in convolutional neural networks, at least when the dataset is not large! (Quote deep sparse rectifier network paper and maybe another, just to show that your observations are consistent with the literature). Ideally, show the features or filters or something.

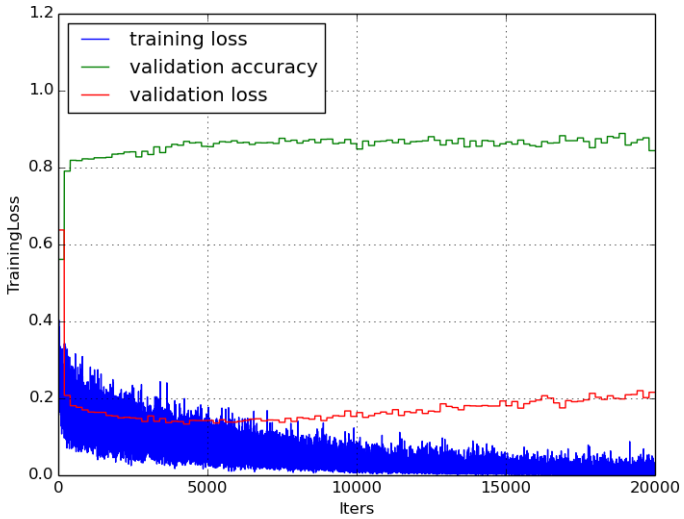


Figure 32: With transfer learning

8.3.2 Initialising Free Layers

Another hyperparameter in transfer learning is whether to initialise the layers in which backpropagation has been enabled to the transferred network's weights, or to randomly initialise them. A way to think about this intuitively is to wonder: do we want to keep the features learned at the given layer or not?

The features on convolutional layers are easy to visualise, and the success of transfer learning in computer vision rests on the fact that pixel features good for one computer vision task are also good for others. This makes intuitive sense for low level shapes: our visual world is made up of combinations of edges, corners, textures and shapes. But what about the features in the fully connected layers?

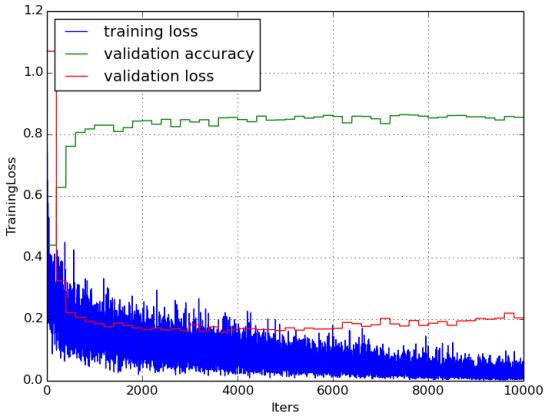
There currently exists no way of visualising the semantic content in fully connected layers: the Intruiguing properties paper suggests that an individual neuron does not contain a semantic feature, but that it is distributed among the units of the layer. So it is not obvious whether or not the features in the fully connected layers are of any use for transfer tasks. That is the motivation for experimenting with weight re-initialisation.

Are below comments still true once retrained no mistake long time for both?

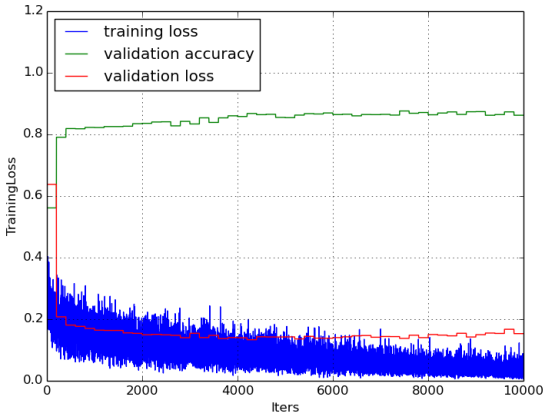
Re-initialising the weights has other effects too. It initialises the network in an area of the parameter space which is further away from the optimal region, which is why the training error starts higher. Secondly, the training error is more volatile: this is a result of the network having "more space to move". As a result, overfit occurs sooner too. What is slightly surprising is that re-initialising the weights does not increase performance overall, despite advice from Soumith Chintala to the contrary (any paper?). Perhaps, if data had been more plentiful, re-initialisation would have proven superior. However, given that the backpropagation is fully enabled in such a network that is very large relative to the dataset (remember this network size was trained on ImageNet), overfit occurs quickly.

8.3.3 Freezing Backprop on various layers

Note: under-sampling, weights initialised to AlexNet for all nets in this section. Assume these aspects cannot be optimised.

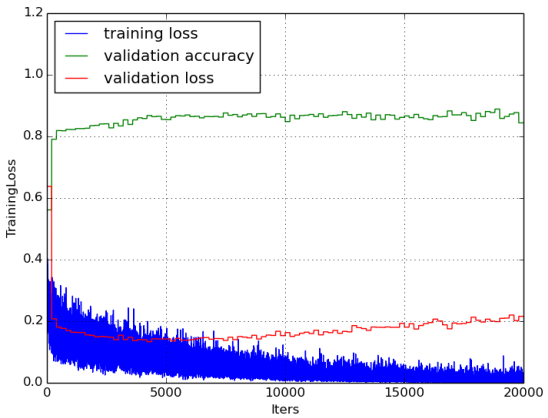


(a) backprop on all layers

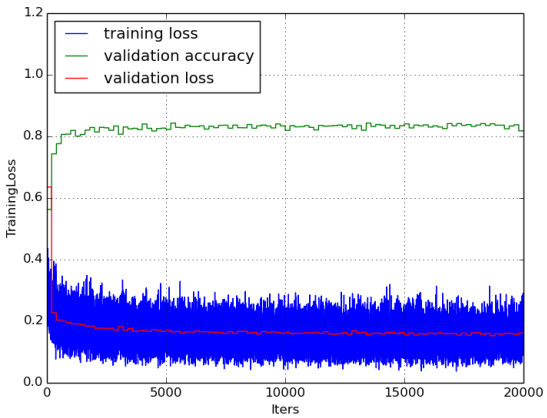


(b) backprop on top layer only

Figure 33: Another figure



(a) backprop on all layers



(b) backprop on top layer only

Figure 34: Another figure

Do NOT take nets in which weights have been re-initialised.

Illustrates how the extent to which backrpop is freed up alters the expressive power of the network and its tendency to overfit (have you got a section in your paper explaining overfit? use the polynomials from Bishop, or the MLNC slides from A. Faisal to illustrate it, then link up with).

Is there an optimal middle ground (i.e. non-monotonous error) number of layers to free up. In our case, freeing up [???] layers is optimal. This middle ground is surprising: I'm surprised, I would expect more overfit when more layers are freed for backprop. Could anyone explain this?//

Soumith's opinion: Are you resetting the weights in the layers FC6,FC7 or are you just backpropping from the AlexNet weights? I think if you reset the weights, you will see what you are expecting.
Me: Yeah, weights initialised to alexnet's. Why do you think that makes a difference? The local minima from alexnet's region leave less room for the net to learn meaningful patterns and push it more quickly into overfitting?
Soumith: when doing transfer learning, if you kept alexnet weights you are already in a well-settled local minima for that layer, so there wont be much movement I think.

The middle ground is the result of the usual expressiveness tradeoff: movement in function space versus over-fitting of the model.

8.3.4 Parametric vs Non-parametric

Motivation: Still on the topic of whether features in the fully connected layers are of transferrable use, the results of 'CNN features off the shelf' offer guidance. The paper achieves quasi state of the art performance by trianing a very simple linear SVM on the feature space of the penultimate fully connected layer (obviously not the highest one, since this one is specifically trained for distinguishing among the classes is the source task).

The difference between training a linear SVM and a standard softmax layer are: - SVM fits the best separating hyperplane. this assumes linear separability. - softmax performs logistic regression. does this assume linear separability?

Expound: Softmax logistic regression: SVM maths + illustration

optimise margin-based loss rather than log probability of the data cite arxiv.org/pdf/1306.0239.pdf

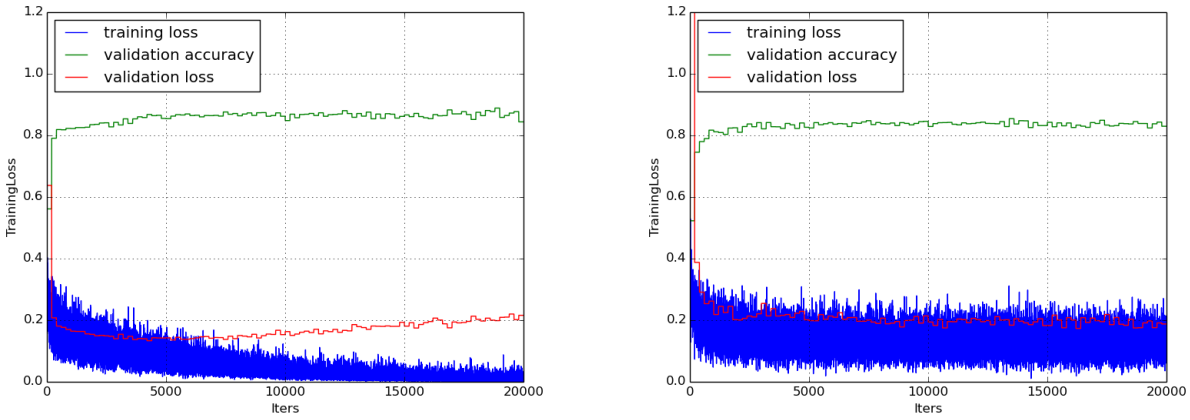
Implementation: - for linear SVM: a single inner product layer with elementwise product activation functions (to achieve linearity), then a hinge-loss layer. - for standard deep neural network: we take the best performing network obtained from previous experimentations.

Note that the linear SVM model was not equipped with a hinge loss accuracy layer, but was rather given the standard per class accuracy layer, which outputs the log probability as validation error.

Results: Comment in light Are All Loss Functions The Same? http://arxiv.org/pdf/1306.0239v1.pdf

We observe that the training error does not converge to zero with the hinge loss, which means that the training data projected on fc7 space is not linearly separable. This is understandable.

The errors between these two models are not comparable since they are of different nature. Just because the hinge loss is higher in absolute terms than the log probabity does not mean that test run performance is lower. The way to compare the two models is to look at classification performance on the test set.



(a) log prob cross entropy loss

(b) hinge loss

Figure 35: hinge loss

9 Task 3: Class Imbalance

9.1 Motivations

The first challenge posed by the Bluebox dataset was its small size, the second one was class imbalance. Although transfer learning by itself delivered strong results for clamp detection (including dealing with class imbalance), this was not the case for the following tasks: [...]

Therefore, additional approaches for dealing with class imbalance were sought out. However, instead of trying them out on the unsatisfactory tasks, they were tried on a subset of the clamp detection dataset where class imbalance was increased (simply by throwing out training cases without clamps). This way, one knows that the only thing preventing performance on the task is class imbalance, and the impact of an approach on classification performance is a good measure of the impact of the approach on class imbalance.

Were this tactic not adopted, it would not be possible to ascertain whether an approach failing to deliver a performance increase would be due to its inability to tackle class imbalance, or to the fact that the task is difficult or impossible for other reasons. For example, water contamination risk is defined by the presence of droplets on the pipe fitting. However, when images are downsized to 256x256 pixels for AlexNet, droplets become invisible to the naked eye. This task could therefore be impossible to learn with networks the size of AlexNet, for a reason not related to the strong class imbalance that the task also presents (92.5%).

9.2 Implementation

c++ per class accuracy layer, threshold layer. bayesian softmax loss layer. bayesian layer abstract class with OOP from which the previous two inherit, which has access to minibatch labels and can compute the prior for the specific batch. More precise than assuming each minibatch has the same prior as the entire training set.

$C++$ per class accuracy layer as an indicator for bad min: do the maths to explain how bad min necessarily implies 0.5 per class accuracy. A flaw is that 0.5 per class accuracy can be caused by other things than bad min - and this did throw me off guard during the project on an occasion. Therefore, it is not a detector! To make sure that we are indeed in bad min, we can run the net and look at the output probabilities.

It will be crucial to keep in mind throughout that the per class accuracy layer provides per class accuracy, but the log prob which it outputs is not

that's also why we thought the per class accuracy was "fucked up": it was outputting per class accuracy, but OVERALL softmax loss hence why the 2 didn't always vary in the same direction!

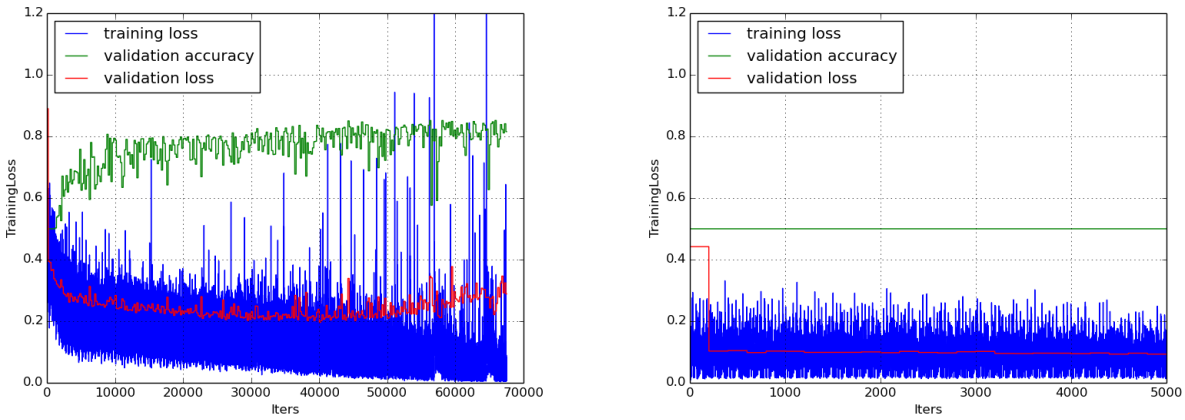
python script.

increase class imbalance: $target = (max_num)/(total_num - delete_min) target * total_num - target * delete_min = max_numtarget * delete_min = target * total_num - max_numdelete_min = total_num - (max_num/target)$

9.3 Experimentation

Several approaches were taken to tackle class imbalance. Successful ones accumulated as experimentation went along.

A fixed architecture and training strategy are kept throughout the experiments: backprop on all layers, all weights transferred.



(a) 88% class imbalance

(b) 98% class imbalance

Figure 36: no transfer learning

The level of class imbalance to impose on the clamp detection task was chosen such that the optimal transfer learning setup obtained from previous experimentation no longer enables successful learning of the task, and puts the network in bad local minimum instead. This corresponds to 98%, from an original imbalance ratio of 0.88%. This corresponds to going from 1323 minority class training cases to 193.

9.3.1 Test Run

To evaluate the difficulty of the clamp detection task with increased class imbalance, a model was trained from scratch without transfer learning, to be benchmarked with the equivalent trained on the entire dataset.

Whereas 88% imbalance made it difficult to learn without transfer learning, 98% imbalance makes it impossible. The model becomes a constant function. Note that although the

9.3.2 Transfer Learning

Transfer learning keeps the net's parameters from bad fake local minima, so the model is not a constant function. But little is learned. Transferring the fully connected layer weights seems to help a little as well. Note that validation error is very low because it is not a per class average: the transfer learning models quickly obtain high precision on the majority class, but struggle to learn the minority one.

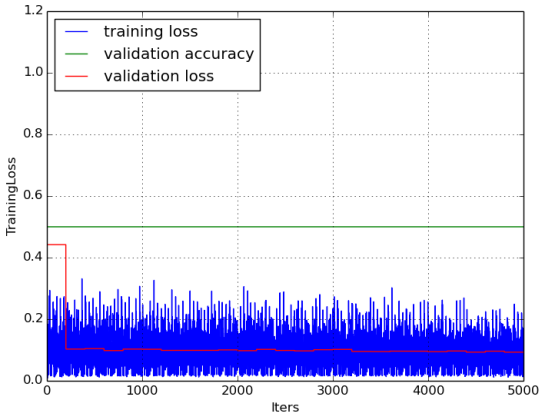
The question is whether the difficulty in learning the minority class is the inevitable consequence of having few training examples to learn from, or whether the imbalance is penalising the minority class from being learned. If the latter case is correct, one should strive to find a way to remedy this.

9.3.3 Batch Size

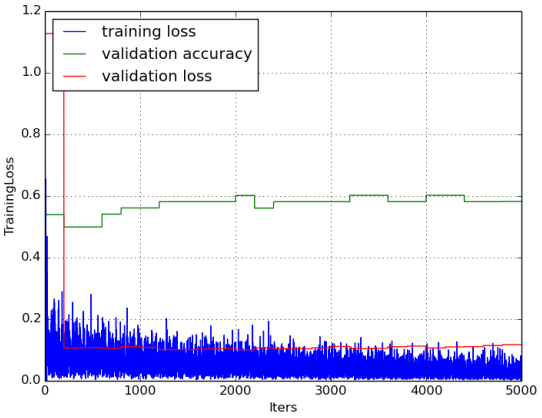
Notice what happens when validation batch size too small:

(training files for above now in clampdetCI/BULLSHIT)

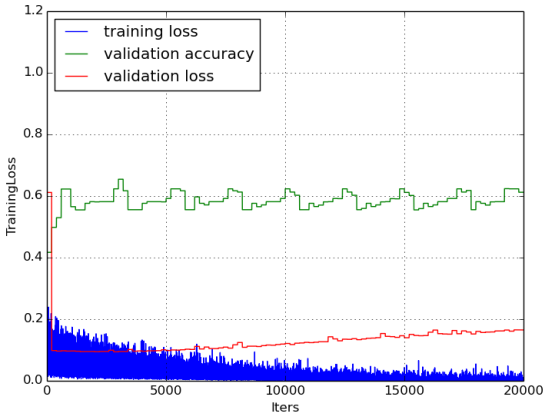
Class imbalance is so high that it is sometimes not present in the subset of the validation set that is being tested. In that case, if the model is in bad min, its per class accuracy will be equal to its accuracy on the majority class. We obtain the confusing output above. (Still got to do some maths to prove that assigning majrity class randomly with probability p (not 0.5) will lead to per class accuracy being 0.5 or p depending on whether minority class present). Hence this confusing output,



(a) no transfer



(b) convolutional layer transfer



(c) full net transfer

Figure 37: 98% imbalance, varying levels of transfer learning

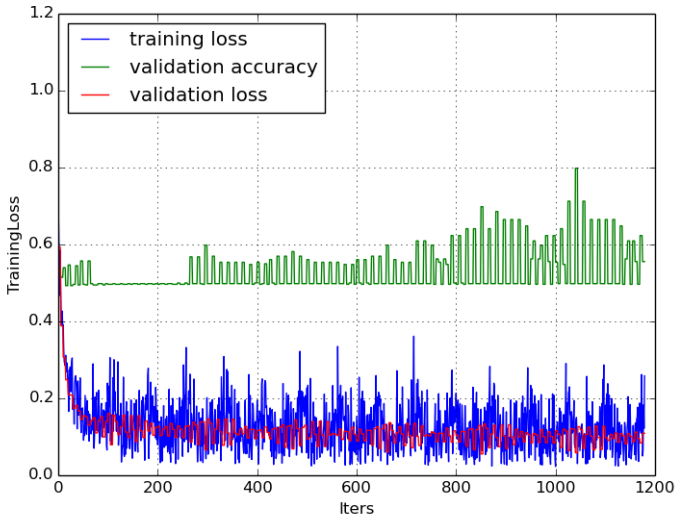
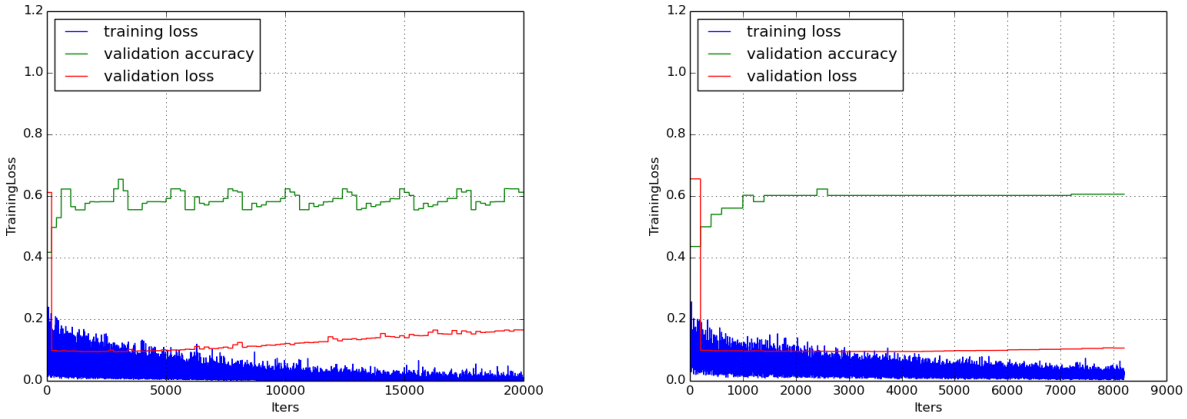


Figure 38: Clamp Detection, Full Backpropagation



(a) mini-batch size 128

(b) mini-batch size 256

Figure 39: 98% imbalance, different mini-batch sizes

which could wrongly be interpreted as the network steadily learning but sometimes getting stuck in bad min (and getting back out thanks to momentum).

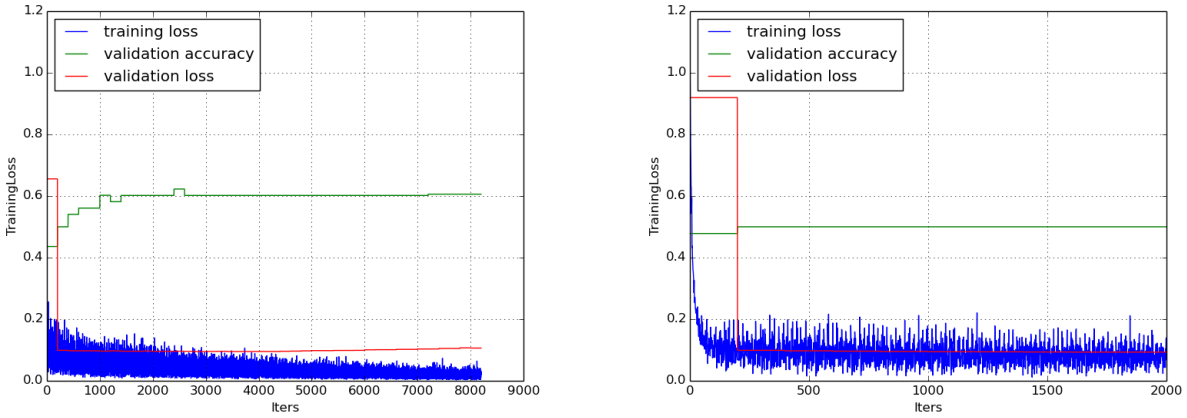
(training files for that in clampdetCI/none on graphic06)

Increasing batch size certainly helps, which shows that the gradient is noisy, so the network parameters head in a suboptimal direction. Noisiness of the gradient is understandable given the class imbalance.

9.3.4 Learning Rate

Decreasing the learning rate helps if it prevents a good minimum from being overshot (elaborate, illustration?).

With a lower learning rate, the net learns the same thing, but a lot slower. This confirms that there was no overshooting, the model converges towards the same minimum. So 0.0001 is a good learning rate for this task.



(a) learning rate 0.0001

(b) learning rate 0.00001

Figure 40: 98% imbalance, different learning rates

9.3.5 Bayesian Cross Entropy Cost Function

If hypothesis supported, maybe rebalancing costs with a different cost function can allow us to under-sample less severely?

Motivations I also wonder whether it should be used in conjunction with a modified cost function. the threshold renormalisation won't change the fact that most of the error will come from c_2 examples, so the net would still be encouraged to learn c_2 more than c_1. [Provide a simple mathematical example of this]. If we want the stochastic gradient to reflect as much of the c_1 errors as the c_2 errors on average, then we need to renormalise the errors based on priors as well. This is done with bayesian cross entropy: [formula here]

When viewed as MLE estimation, the Bayesian cross entropy can be interpreted as follows: instead of trying to maximise the joint probability of the data under the assumption that each observed event has the same probability, we are assigning higher probability to minority class events.

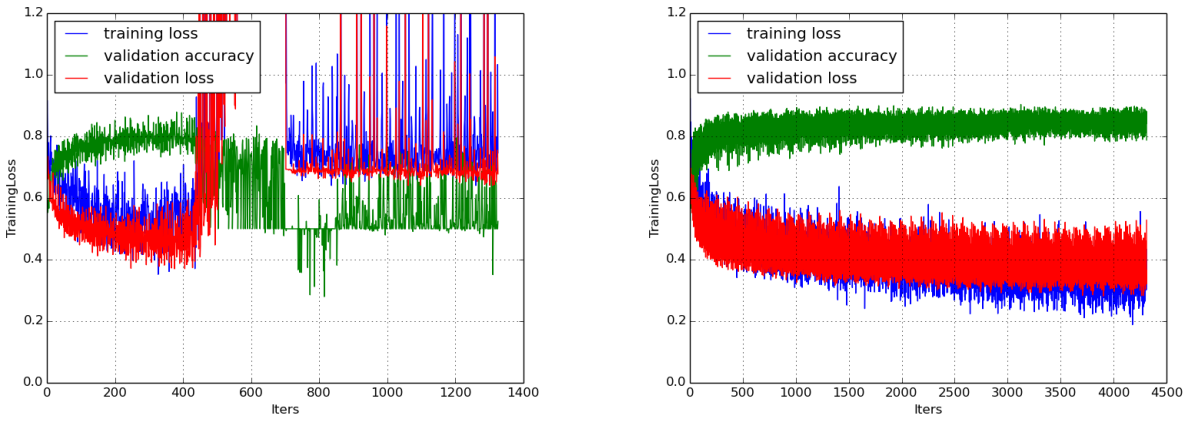
Recall that the error being shown is the negative of the log probability of the likelihood function:

$$-\frac{1}{n}\sum_{i=1}^{n} ln(f(W|x_i)) \tag{14}$$

Where f is the learned function. This is the cross entropy, but it also elegantly evaluates to MLE. In other words, we are choosing the parameters of the model to maximise the likelihood of the data (to make the observed events i.e. the training set as highly probable as possible). Do the proof with joint probability of independent events etc.

Blow us away with some nice maths here! Also need the background maths for cross entropy that is the same as MLE.

Implementation Implement a softmax bayesian loss layer, and a per class bayesian accuracy layer, in C++.



(a) Incorrect gradient propagation

(b) Correct gradient propagation

Figure 41: Ground Sheet detection, different backpropagation formulae

A difficulty implementing this was small batches with a missing class, the loss would be infinite and weight updates would screw up the net.

For testing and debugging, the cost function was used on one of the simpler tasks with little imbalance: ground sheet detection.

When gradient not correctly propagated, backpropagation does not converge.

Makes gradient a noiser because class imbalance from one minibatch to another varies. This could be improved by making sure same class proportion in every batch. Note that validation error is noisy here because 1 validation batch is tested at a time (for faster evaluation of implementation).

I had thought that the "bayesian softmax loss" had failed because the models trained on it were not hitting as low a validation error minimum as were the models trained on normal softmax loss HOWEVER scarcity of minority class cases makes the class harder to learn, no matter what tricks you use to get the net to "focus on it more" since softmax bayesian loss is a kind of per class average softmax loss, it will always be higher than sotmax loss when there is class imbalance even if it achieves in getting the net to learn the minority class better!

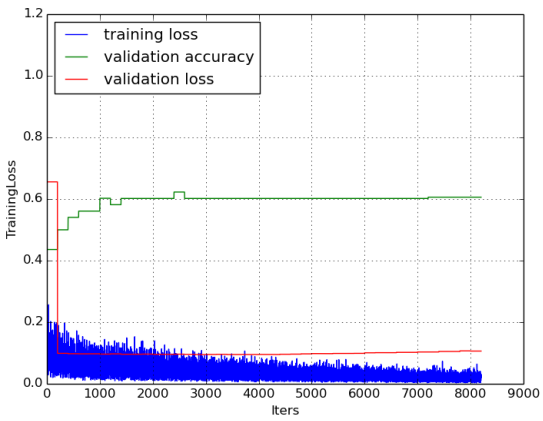
I'm going to run the models on the test set look at the accuracy on positives and negatives and if I get better accuracy on the positives, it's great.

no free lunch: if you're making minority class cases stand out disproportionately, you're telling the net "screw overall performance, try to get the minority class good" "at the expense of performance on the majority class" which, when you're taking an unweighted measure of performance such as softmax loss, lowers performance.

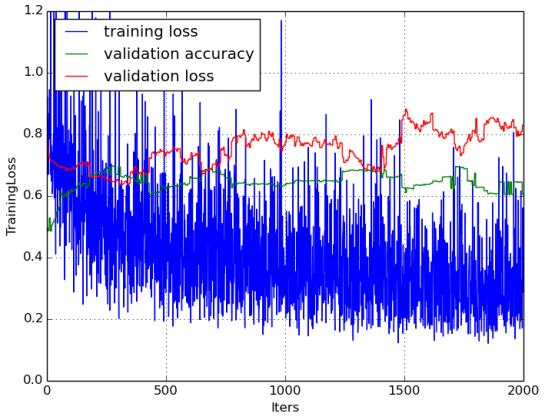
Results Full transfer learning, backpropagation on all layers, batch size 256.

With softmax bayesian cross entropy, the training is far less clean, but the highest per class classification accuracy reached is nearly 10 percentage points higher.

The training is less clean because the modulus of the gradient is strongly influenced by imbalance within the mini-batch, and this value can greatly vary from a mini batch to the next, since a mini batch with 3/256 minority class cases has minority class proportion 3 times that of a mini batch with 1/256 minority class cases, and both are likely to occur in sampling.



(a) softmax cross entropy, 0.0001 learning rate



(b) softmax bayesian cross entropy, 0.00001 learning rate

Figure 42: 98% imbalance, different cost functions

Need to mathematically derive how imbalance affects the modulus of the gradient, and how the standard deviation of the modulus with SBL is much greater than with SL.

Compare the two nets with run classifier script, to see whether they have same sig values. If they do, then SBL is just doing thresholding, nothing more. Otherwise, it's doing something better.

Performance improvement could be obtained by enforcing the same class proportions at every iteration.

9.3.6 Under-Sampling

Imbalance is obviously reached because most mini-batches do not contain any minority class examples, so the net is only ever being told to output 0.

Overfitting consequences of under-sampling Here we evaluate impact of under-sampling on clampdet. There is no need to under-sample on clampdet because class imbalance is tackled by transfer learning. But that is the reason for why clampdet is used to illustrate impact on overfitting: apply under-sampling in a case where learning occurs regardless, to see what impact it has on the health. We can't be certain about this impact on fitting proximity, since it just doesn't learn anyway.

Comparison 1: without weight re-initialization
Notice how under-sampling makes overfit take place so much sooner. As a result, the model cannot pick up any of the higher order generalising patterns, and the performance is not as high (go down from 94% to 85% or something right?). This begs for other class imbalance tackling tricks to be combined with under-sampling, in order to limit the extent to which we under-sample, in order to fragilise robustness as little as possibe, so that we can use more expressive models, i.e. enable backprop on more layers.

9.3.7 Over-Sampling

And now for an alternative to under-sampling.
For the implementation: (min + copy) / (max + min + copy) = target
min + copy = copy*target + target*(max+min)

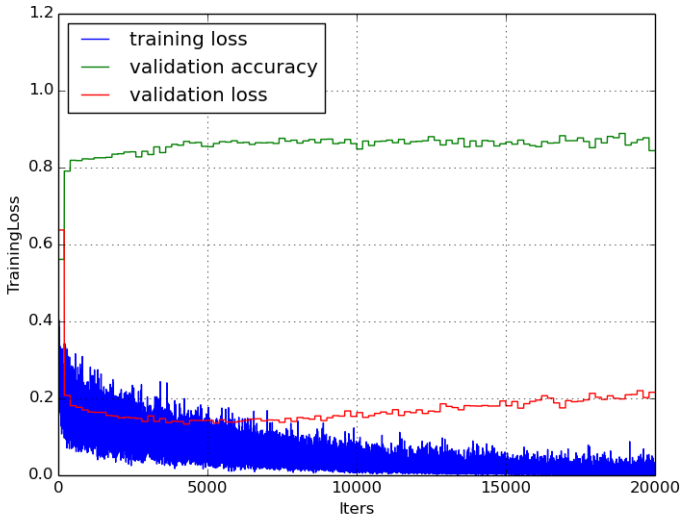


Figure 43: Clamp Detection, Full Backpropagation

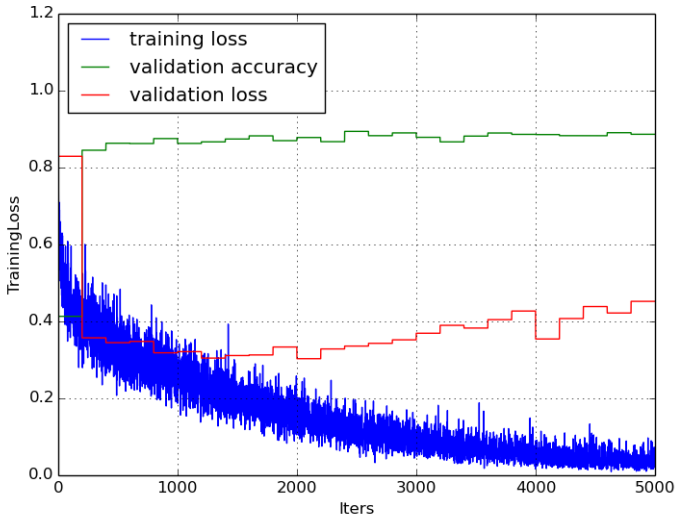


Figure 44: Clamp Detection, Full Backpropagation

copy = copy*target + target*(max+min) - min
(1-target)*copy = target*(max+min) - min
copy = (target*(max+min) - min) / (1 - target)

Duplicates are created in training set, but we don't want them in the validation or test sets as this will slow down the algorithms (though it wouldn't bias the results aka metrics).

Oversampling completely fails.

Observations: - very weak performance - per class accuracy goes below 0.5 - entire error time series are shifted up by a constant - overfit occurs sooner

Interpretations: How can per class accuracy be below 0.5? that's worse than random! It may be due to the fact that copies are made in the training set, but not in the test and validation sets. When comparing these networks, should we not be using the same validation set for all? No, we should be using the same test set (and testing will come in the next section). We should choose the validation set that we deem best for early stopping. In the case of over-sampling, makes no sense to include duplicates.

There might be bugs in the implementation. Check in this paper: http://sci2s.ugr.es/keel/pdf/algorithm/artic

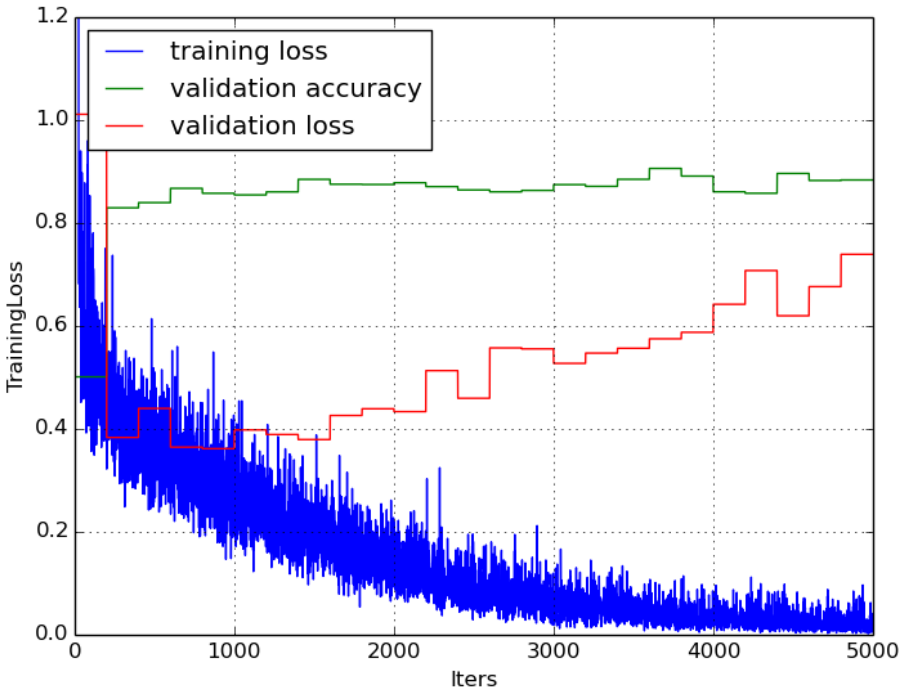


Figure 45: Clamp Detection, Full Backpropagation

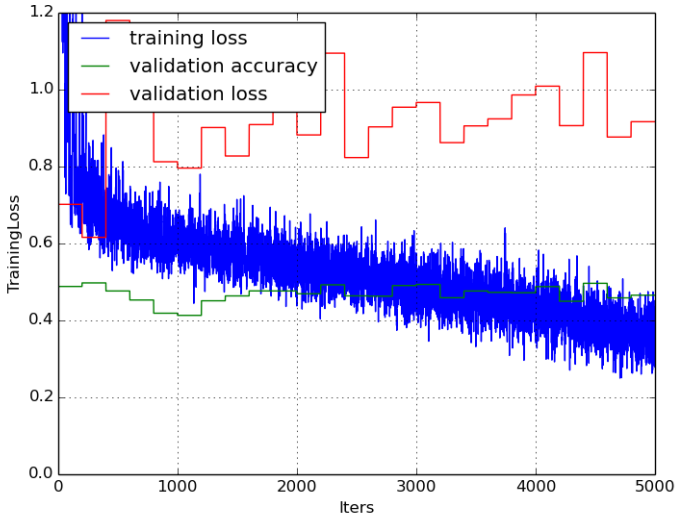


Figure 46: Clamp Detection, Full Backpropagation

what it says can be linked to what you got. Otherwise, just omit it from your report.

9.3.8 Test-time threshold

This is great because it enables to set different costs to false positives, which is what ControlPoint is interested in.

Papers: http://sci2s.ugr.es/keel/pdf/algorithm/articulo/2006http://www.eiti.uottawa.ca/ nat/Workshop200 icml03-wids.pdf

It also allows you to assign different costs to the different misclassifications (i.e. if the algorithm

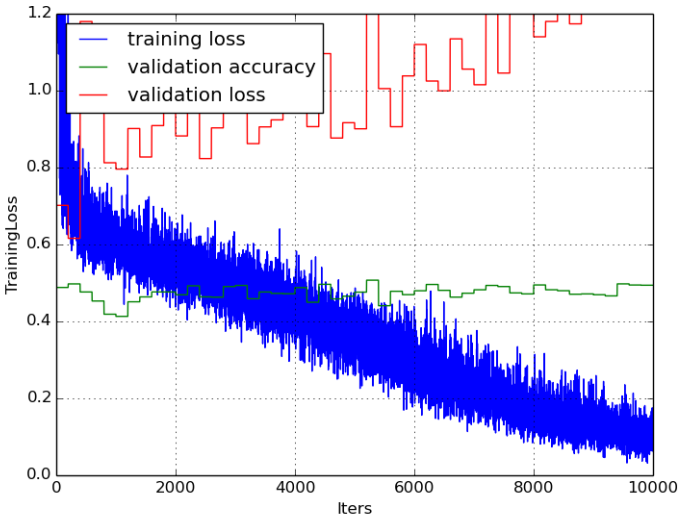


Figure 47: Clamp Detection, Full Backpropagation

is only supposed to flag potential problems then a false negative is much more expensive than a false positive) the idea is that, if we have 2 classes and assign equal cost to both misclassifications, then we would want to pick the label with the biggest p(data — label) probability however, the NN will compute the probabilities p(label — data) p(data — label) p(label) and a class imbalance means the priors p(label) are different so you just need to correct for that by multiplying, for instance, the probability of p(label = 1 — data) outputted by the NN by the factor p(label=2) / p(label=1) and renormalizing

I get why it's called threshold: suppose $p(c_1) = 0.1$. the threshold for classifying as c_1 moves from 0.5 to 0.1. Illustrate this point with one or two simple examples ("intuitively, "). (that's pretty much what the "sig level" script I mentioned intends to do, except script has a flexible sig level.) Also, I think that if we wanted to implement the threshold directly in the net, we might need to put the threshold layer after the softmax layer (and have it renormalise by $1/num_classes$). cos it's non-linear so you can't be sure that their output will be renormalised in the same way.

Rather than have a fixed threshold hard-coded into the network, I propose to use a sig level script, for greater flexibility.

10 Task 4: Conserving Spatial Information

10.1 Motivations

Some tasks depend on picking out visual characteristics in specific areas of the pipe installation: - soil contamination risk - water contamination risk - scraping or peeling - (scrape zone detection?)

Whereas pooling in convolutional layers condenses the visual information and achieves translation equivariance, it also loses spatial information (where the feature was seen). This can be severely detrimental to the tasks above. For example, there is risk os soil contamination when traces of soil can be seen on the pipe fitting, but not if they are anywhere else. Therefore, in this case, it does not suffice to ascertain that traces of soil can be seen on the image.

10.2 Implementation

The architecture of the network was easily modified thanks to the prototxt interface featured in caffe.

10.3 Experimentation

Test Run Note that above batch size 96 saturates gpu memory. The reason for why this model cann take up more space is because without pool, the partial derivatives between conv5 and fc6 are twice as large, and that already constitutes the largest Jacobian in the model: we go from a (bSize*256*6*6) x (bSize*4096) Jacobian to a (bSize*256*13*13) x (bSize*4096), i.e. four times as large. However, the reason for why the GPU is unable to support is probably due to CUDA implementation details, and how many cores have been allocated to that specific Jacobian. After all, the Titan Black has 6GB of memory, which should be ample space. In the interest of time, the issue was not explored further.

Remove pooling and an fc layer

Softmax Bayesian Cross Entropy

11 Final Results

This presents best results (accuracy overall, on pos, on neg, sig level) for each binary classifier and discusses further improvements. All successfull tricks were accumulated to produce these results.

To sum up, the two greatest challenges with the Bluebox dataset is small size and class imbalance. Small size can be dealt with by transferring knowledge from another task by initialising weights. Class imbalance can be cruelly dealt with using under-sampling, but this decreases the size of the dataset even more, making robustness ie generalisation the second challenge. Therefore, we use [..] and [..] to help with dealing with class imbalance and mitigate the extent of under-sampling.

This is where you mention the multi-query issue, the mis-labelling issue. Also mention the things you could not experiment with: the entire list in your .md file.

11.1 Merging Classes

CNNs were trained on all of the "No Clamps" or "Clamps Not Sufficiently Visible" images, and a random sample containing 15% of the "Clamps Detected" images, in order to obtain an acceptably balanced dataset. This fix was easiest to implement and served as a benchmark for the more sophisticated subsequent approaches. Naturally, one would also expect this approach to be least effective: 85% of the training data was lost, data which could intuitively serve not to teach the network to distinguish classes, but to learn features that are good encoders of pipe weld images, from which an effective discriminative setup of the parameters could ensue.

Note that the two minority classes were merged: because it is suspected that there is barely any semantic difference between them (will have to draw random samples to verify this), and because this reduces the number of majority class instances to remove, since it makes it easier to reach a good balance ratio.

11.2 Learning Rate

Step Caffe code: Return the current learning rate. The currently implemented learning rate policies are as follows: - step: return base $lr * gamma^(floor(iter/step))$

These are all heuristics. No 2nd order methods involved. The intuition is that... Theory suggests that... .

Improvement: 94.5% validation error. The test error is: ??.

Exp [...]

11.3 Soil Risk Contamination Task

Imbalance ratio: 96.1%.

Instead of transferring AlexNet, optimal clampdet model is transferred. The underlying assumption is that the way in which clampdet model finetunes is also useful for soil contamination.

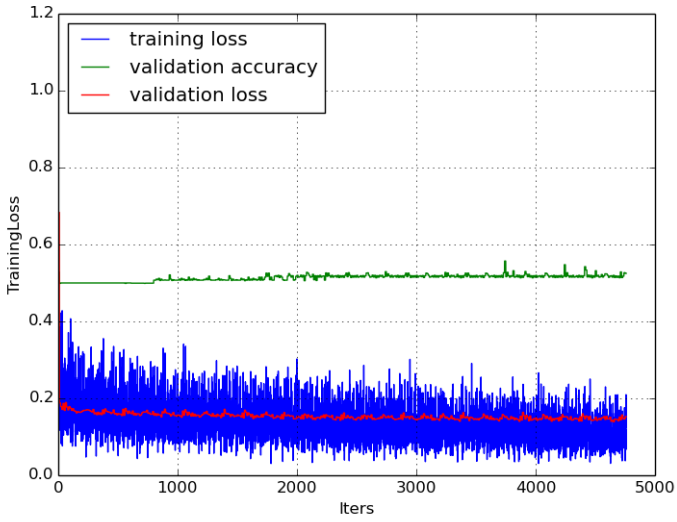


Figure 48: Clamp Detection, Full Backpropagation

Sadly, hardly learning anything. Notice the training error takes a lot of time (does it even reach it?) to reach 0, suggesting it is very difficult to even fit the training data. We can be even more precise than that thanks to ours insight into gradient descent and Chebyshev polynomials: we know that we require an order 'train-iters-to-fuly-fit' polyomial to create regions that cover and discriminate the training set properly, which means that the features learned are inadequate for separating the data. Qualitatively, we know that soil contamination depends on a very specific part of the image, and pooling loses this information. A speck of brown in a specific area means soil contamination, but the same speck elsewhere does not. Therefore, we can say a convnet is not suited for this task. (Unless we use segmentation to crop out the area of the image - the fitting - which we wish to examine for contamination).

Test Run Class imbalance is very sharp, with only 3.9% of soil contamination in the Bluebox images. So even with good initialisations from transfer learning, quite likely that network will not learn anything.

One should keep in mind that there could be a lot fake minima corresponding to "output majority class every time". The parameter space and the function space associated with every single parameterisation of a given deep neural network is not isomorphic; i.e. for a function, there exists many different parameterisations that can represent it. (Need to find a paper that goes into more depth on this.)

So even if backprop is frozen on all the lower layers to ensure that they are good feature detectors, it is still possible with just the use of the higher layers to generate a classifier that is nonsense from the point of view of learning representations of the classes at hand. As a result, fake minima can exist in the sweet spot region (sweet spot region as described by Yann LeCun's research). Since class imbalance is so sharp in this case, these fake minima are even deeper (they correspond to 96.1% success rate), so they are likely to be deeper - and therefore accessible via gradient descent - than the true minima close to the point on the surface at which the network 'lands' with transfer learning.

Training Results Validation error 0, at initialised weights: Classification % success: 0.125977 Cost function score: 1.35429

Validation error 1, after 50 mini-batch passes: Classification % success: 0.964844 Cost function score: 0.179662

Training error 0, at initialised weights: Cost function score: 1.35429

Training error 1, after 1 mini-batch passes: Cost function score: 0.179662

Training error 1, after 1 mini-batch passes: Cost function score: 0.179662

(just have a plot of this! first 200 mini-batch passes).

Note no classification % success because for efficiency reasons it is not computed by caffe for training batches.

Observations Converges extremely quickly to 96%. To confirm this, would have to run validation after every single minibatch pass. If that is indeed the case, then it tells us about how class imbalance litters the error surface with fake minima (one could emit the conjecture that the fake minima are as dense as the injectivity between parameter space and function space).

Stays in the 96% region for the remaining mini-batch passes (have plot of all of those iterations). However the validation error is not always exactly the same - but if the validation is always using the exact same data and the net is indeed stuck at a fake minimum, shouldn't it always be the same value, i.e. the one corresponding to the exact proportion of majority class instances in the validation data? Check this!

Get more evidence Looking for evidence that in a given pot, there is a high number of fake minima all mapping to the same function that spits out majority class every time. If so, then the injectivity of parameter space into function space is not uniform: some functions can be represented by a lot more different parameterisations than others, and sadly for us, the bad function is one of the densely represented ones. Would be fantastic to come up with a toy example of this with a very simple network, but mathematically prove just how much more numerously certain functions can be represented than others.

One could empirically verify this by looking at what the network converges to with and without transfer learning. Also take very different initialisations for non transfer learning. If converged error rate same for all, and indeed sending in a batch of random images to each network spits out same or similar outputs every time, then we've got a few large scale examples as well.

transfer learning doesn't work as well with caffe: https://github.com/BVLC/caffe/issues/642 works better with overfeat: http://cilvr.nyu.edu/doku.php?id=software:overfeat:start

11.4 Hatch Markings

72.5% class imbalance.

Probably gains to be made from keeping information here.

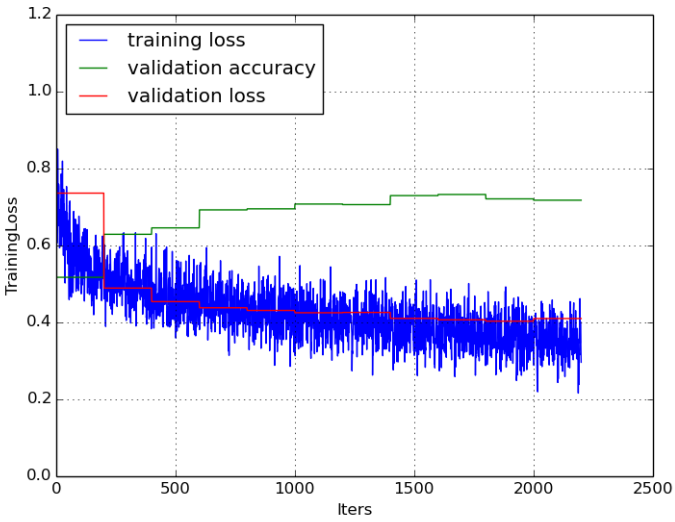


Figure 49: Clamp Detection, Full Backpropagation

12 Conclusions and Future Work

The realisation that weird sampling can heavily dent the approximation of the error surface has raised questions:

How does the training set provide an approximation of the true error surface? Since it can introduce dangerous minima, does that mean that, formally, it does not always provide an extrema-conserving approximation of the true error surface? Theoretically, what are the conditions for obtaining an extrema-conserving approximation (i.e. that doesn't introduce fake minima)? Practically, can we perform transformations on the cost function, or do stuff to our data (sampling), to limit the introduction of fake minima?

Could we answer these questions if, instead of facing the usual problem of having a training set sampled from an unknown distribution, we ran a completely artificial experiment where we start with a known distribution? That way, we know the true error surface, and as we draw samples from the distribution, we can look at how each one approximates the true error surface, how it introduces bad minima?

How does mis-labelling alter the error surface? Intuitively, it would seem that it lifts up the true minima only, making the false minima even more attractive (e.g. "labelling is so bad it's too confusing, there's just nothing to distinguish them, so I might as well go for the blind strategy of outputting clamp detected all the time").

because exists injection of parameter space into function space, and because of what Yann says, can view the error surface as a replication of side-by-side identical pots with jagged bottom. However, seeing as we run into imbalance-induced fake minima with transfer learning as well, despite Conjecture that imbalance-induced fake minima are littered across the entire error surface, also present in the sweet spot zone. Conjecture further that they are as dense as the relative size between parameter space and function space. Could be interesting to think about how the choice of a network architecture affects this injectivity.

Unsupervised Learning to correct mis-labelling, with encouragements to create clusters initialised by those resulting from supervised learning (maybe could only work well with many false negatives, few false positives, as is the case here?).

References

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

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

[3] Lowe, David *Object Recognition From Local Scale-Invariant Features* The Proceedings of the Seventh IEEE International Conference on COmputer Vision 1999

[4] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*
ILSVRC 2013

[5] Nair, Vinod; Hinton, Geoffrey; *Rectified Linear Units Improve Restricted Boltzmann Machines*
ICML 2010

[6] 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

[7] Fergus, Robert; *Tutorials Session A - Deep Learning for Computer Vision* NIPS 2013

[8] Olah, Christopher; *Understanding Convolutions* URL: `http://colah.github.io/posts/2014-07-Understanding-Convolutions/`, last accessed 1st September 2014.

[9] 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

[10] Joan Pastor-Pellicer, Francisco Zamora-Martnez, Salvador Espaa-Boquera, Mara Jos Castro-Bleda; *F-Measure as the Error Function to Train Neural Networks*
Advances in Computational Intelligence Volume 7902, 2013, pp 376-384

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

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

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

[14] Zeiler, Matthew; Fergus, Robert; Visualizing and Understanding Convolutional Networks arXiv 2013

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

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

[17] Saenko, K., Kulis, B., Fritz, M., and Darrell, T.; *Adapting visual category models to new domains*
ECCV, 2010

[18] Bengio, Yoshua; *Google+ Deep Learning community post* URL: `https://plus.google.com/+YoshuaBengio/posts/GJY53aahqS8`, last accessed 1st September 2014.

[19] Bay, H., Tuytelaars, T., and Gool, L. Van; *SURF: Speeded up robust features*
ECCV, 2006

[20] Sermanet, Pierre; Eigen, David; Zhang, Xiang; Mathieu, Michael; Fergus, Rob; LeCun, Yann; *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*
arXiv:1312.6229

[21] URL: `https://code.google.com/p/cuda-convnet/`, last accessed 6th June 2014.

[22] URL: `http://caffe.berkeleyvision.org/getting_pretrained_models.html`, last accessed 6th June 2014.

[23] URL: `http://research.microsoft.com/apps/video/default.aspx?id=206976&l=i`, last accessed 6th August 2014.