Deep Learning Methods for Classification with Limited Training Data

Seminar Report : Spring 2017

submitted by

AVIRAL KUMAR
(140070031)

under the guidance of

PROF. SUNITA SARAWAGI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
April, 2017
Abstract

The human brain has an inherent ability to learn to react to something with just one past experience. The quest for Artificial Intelligence has brought us to the situation where machines simulating the abilities of the human brain are being developed. In this context, the a new flavour of the evergreen classification problem, that is, to classify data having seen few training instances becomes quite relevant. As a part of this report, we will look at some of the state-of-the-art deep neural network based approaches used for solving this problem of classification. We first explore the problem of few-shot learning where we are presented with only a few examples of each of the classes. Some solutions to these problems include meta-learning, usage of MANNs and metric learning. We also explore a problem, quite relevant in computer vision, which is reliable classification of examples in case of skewed class distributions in the train data. We will look at a metric learning solution to this problem.
Acknowledgment

This report is an outcome of my seminar work under the guidance of Prof. Sunita Sarawagi on: Deep Learning Methods for Classification with limited training data. I would like to thank her for her guidance, motivation and constant support throughout the duration of the seminar. My research in this area is driven by her vision and support.

I would also like to thank my colleagues Rishabh, Vihari, Ujjwal, Prashanth, Jigar and Deepak for healthy and useful discussions on the vast depths of deep learning.
# Contents

1 Introduction ................................................. 2
  1.1 Deep Learning Methodologies ................................. 3
    1.1.1 Meta-Learning ......................................... 3
    1.1.2 Transfer Learning ....................................... 4
    1.1.3 Metric Learning ......................................... 4
  1.2 Structure of the report ...................................... 5

2 Deep Learning Models ......................................... 6
  2.1 Artificial Neural Network .................................... 6
  2.2 Convolutional Neural Networks (CNN) ........................ 7
  2.3 Recurrent Neural Networks (RNN) ............................ 8
  2.4 Long Short Term Memory (LSTM) ............................. 10
  2.5 Attention Mechanism ......................................... 11
  2.6 Memory Models ............................................. 13
  2.7 Neural Turing Machines (NTM) ............................... 13

3 Deep Learning Methods for Classification with less examples 17
  3.1 Approaches to One Shot Learning ............................ 17
    3.1.1 One Shot Learning using MANNs ........................... 17
3.1.2 Matching Networks for one-shot learning ........................................ 20

3.2 Approach to robust Metric Learning .................................................. 22

3.2.1 Metric Learning for Adaptive Density Discrimination ...................... 22

3.3 Approaches to optimize gradient descent .......................................... 25

3.4 Handling Class Imbalance Scenarios .................................................. 29

3.4.1 Learning Deep Representation for Imbalanced Classification ............ 30

4 Conclusion and Future Work ................................................................. 34
Chapter 1

Introduction

According to Arthur Samuel in 1959, Machine Learning is a subfield of computer science which gives “computers the ability to learn without being explicitly programmed”. Machine Learning evolved from the study of pattern recognition and computational learning theory in artificial intelligence, and it explores the study and construction of algorithms that can learn from and make predictions on data — such algorithms are “live” in some sense, as they overcome following strictly static program instructions by making data-driven predictions or decisions, through building a model from sample inputs and using it to make decisions in future. [14]

Traditional machine learning algorithms include Support Vector Machines, Regression Models, EM algorithms. In the quest for conquering the peaks in AI, a new approach or paradigm to artificial intelligence was introduced, called **Deep Learning**. Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. Deep learning approaches aim to mimic the way human brains work. For example, the central problem in representation learning is solved by introducing representations that are expressed in terms of other, simpler representations. Deep learning allows the computer to build complex concepts out of simpler concepts. [2]

One of the major drawbacks of neural networks is that usually they have a large number of parameters and tuning all the parameters to reach the global optimum of the loss requires a huge amount of training data. But this is different from the human brain. Hence, one of the current hot topic research problems in Deep Learning currently is to make a computer mimic the human ability to be able to learn to predict instances correctly without seeing many instances of that type.
Classification Problem

Given a set of objects \( D = \{ x_i \}_{i=1}^N \), and a set of classes \( C = \{ c_j \}_{j=1}^K \), give a function \( f : D \to C \), such that \( f(x_i) = c_j \) iff object \( x_i \) belongs to class \( c_j \).

Several methods have been developed by the machine learning community to solve the problem of classification like SVMs, Perceptron, Ensemble Methods, Naive Bayes classifier and the list will go on.

Few-shot learning problem

One shot or few shot learning is an object classification problem in machine learning, which deals with exactly one or maybe, a few training examples contrary to what normal deep learning paradigms need (thousands of data). The motivation behind this is the ability to learn object categories from few examples at a rapid pace which has been demonstrated by humans. Apart from the approaches discussed later on in this report, one shot learning problem can also be solved using Siamese Networks, Bayesian One-shot methods, and has also been of interest in the vision community.

In the next section, we describe a few deep learning techniques or methods which have been used to solve a variety of classification problems in some of the recent literature.

1.1 Deep Learning Methodologies

1.1.1 Meta-Learning

Meta learning is a subfield of Machine learning where automatic learning algorithms are applied on meta-data about machine learning experiments. Meta-learning can also be described by the phrase - “Learning to learn”. Usually, a learning algorithm may perform very well on one learning problem, but very badly on the next. A particular model might work on images (CNNs, as we see in the next chapter) while RNNs work well on natural language data. So, by using different kinds of meta-data like the properties of the dataset, performance measures of the algorithm, gradient values amidst gradient descent, a better overall strategy can be achieved which would give rise to a better learned agent. Few examples of Meta-learning include Hyperparameter Optimization, Ensembles and Algorithm selection.

A meta learning system is characterized by the following three properties.

- A learning sub-system that adapts itself with experience
- Experience, which is gained from previous experiences or from different domains
• A meta-learning agent which takes up the job to infer patterns from the experience of the learning subsystem and makes it learn in a better way.

As we will look into the next few chapters, we shall be discussing one of the papers which deals with how to use meta-learning to improve accuracies in one-shot learning.

1.1.2 Transfer Learning

Transfer learning is a learning paradigm which is used to improve a learner from one domain by transferring information from a related domain. [15] As a practical example, consider two persons who want to learn to play piano. One of them knows how to play guitar and his knowledge of guitar will help him learn piano fast or atleast better than the other person.

Mathematically, a domain \( D \) is divided into two parts, \( \mathcal{X} \) and a probability distribution \( P(X) \), for \( X = \{x_1, \cdots, x_n\} \in \mathcal{X} \). Here, \( \mathcal{X} \) is the space of all possible feature vectors, and \( X \) is a particular learning sample. Given a domain \( D \), a task \( T \) is characterized by \( \mathcal{Y} \) (the label space) and a predictive function \( f : X \rightarrow \mathcal{Y} \), such that it predicts, for \( x_i \), the label \( y_i \). The task of transfer learning is the following: Given a source domain \( D_S \) and a source task \( T_S \) and a domain \( D_T \), task \( T_T \), transfer learning is the task of improving the function \( f_T(.) \) given information about \( D_S \) and \( T_S \).

With respect to the problem this report talks about, transfer learning can play a huge role with rare examples, as the model could be trained on some other data which is abundant and then some features of this second model can be transferred to our first model.

Transfer learning becomes more and more ineffective as the distance between the distributions of the two source and target feature vectors, or source and target data increases. Transfer learning can be visualized in case of deep learning models and we look at some applications into our problem.

1.1.3 Metric Learning

The choice of distance metrics between two complicated structures in a machine learning problem is of utmost importance. A proper chosen distance metric which suits well to the problem means that the problem will always get solved to a good extent. Metric learning is the task of learning a distance function over objects provided, where the notion of distance is in some kind of correlation with the similarity between the objects. Examples of applications of metric learning include Mahalanobis distance learning, Neighbourhood Component analysis (NCA), Large margin nearest neighbours, etc. While we describe examples of this later on, we will talk about the mathematical details.
1.2 Structure of the report

As stated earlier, we focus on the problems of one-shot or few-shot learning along with a flavour of classification under imbalanced scenarios. In the next chapter, we describe the basic models that have been used extensively as a part of the deep learning approach. We look at Neural Networks, and then move on to types of neural networks, like, RNNs, CNNs, LSTMs, Siamese Networks. We also explore the latest trend of Memory Augmented Networks (MANNs) and Neural Turing Machines (NTMs). In the last and the final chapter, we look at how these deep models can be used for solving the particular problems we are interested in.
Chapter 2

Deep Learning Models

In this chapter, we will look at the basic blocks deep learning architectures are made of, and how a combination of such blocks creates a lot of wonders. Deep Learning entirely begins from the idea of Perceptron. Perceptron is an algorithm, developed in the late 1950s, which is used to produce a linear separation boundary given some points in some $n$-dimensional space ($\subseteq \mathbb{R}^n$). It is an online algorithm and can receive data as a stream and can learn from infinite amount of data.

We first define what neural networks are and then move on to the various types of neural networks.

2.1 Artificial Neural Network

An Artificial Neural Network or a Multi-layer perceptron (MLP) is a connectionist model which consists of multiple perceptron cells arranged in the form of a directed graph. The output of a perceptron acts as the input to an activation function (like sigmoid, ReLU, $tanh$), and then the output of this activation function goes into the next perceptron cell this is connected to. Neural Networks as proven mathematically, can actually simulate any function $f : \mathcal{X} \rightarrow \mathbb{R}$. NNs can have many layers, any many configurations. NNs are trained using backpropagation, which is essentially a gradient descent update, but done collectively over the cells of the NN.

Neural Networks have been extensively used for classification problems. Being descendants of the perceptron, using a neural network, we can obtain non-linear separation boundaries of arbitrary smoothness.

Neural Networks usually suffer with the presence of several parameters which require a large amount of data to be trained properly, which otherwise can lead to absolutely incorrect inference.
In the rest of this chapter, we shall discuss the different types of neural networks and majorly those which hold a relevance in the topic this report is based on.

## 2.2 Convolutional Neural Networks (CNN)

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input. (usually applied to images or other 2D input such as a speech signals). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. A CNN is made up primarily of 3 kinds of layers: Convolutional layers, Pooling layers, and Fully Connected layers.

**Convolutional Layers**  The input to a convolutional layer is a $n \times n \times c$ input (c is the depth, number of channels). The layer has $k$ filters of size $m \times m \times q$ where $m \leq n$. The filters create a map of depth $k$, with dimensions $n - m + 1 \times n - m + 1$, where each cell in the output corresponds to the convolution of a local 2D region.

**Pooling Layers** A pooling layer is an downsampling layer, which downsamples the output of the convolutional layer. This motivation behind using a pooling layer is to just pass relevant features to the next layer and thereby, summarizing generated features in the previous convolution layer. There can be many types of pooling operations: max pooling (selects the maximum valued cell out of the receptive field of that neuron), average pooling (passes the average of the cell values in the local receptive field of the neuron). Either before or after the pooling layer an additive bias and sigmoidal nonlinearity is applied to each feature map.

**Fully Connected Layer** At the end of alternating conv and pooling operations, a fully connected layer helps in prediction of the required classification output, from sufficiently downsampled and enhanced features.

**Backpropagation** Backpropagation in CNNs is more or less identical to that of a feed-forward neural net. For pooling layers, error is backpropagated by appropriate upsampling, for example, for average pooling, error is distributed uniformly over all the cells in the local area. For convolutional layers, the backpropagation step is complex, as it involves flipping of the error matrix by $90^\circ$.

---

1A convolution of two functions $f$ and $g$ is a special kind of linear operation and can be described by a weighted average of $f$ over a region such that $f(\tau)$ is weighted by $g(T - \tau)$.
Applications of CNNs CNNs exploit structural locality, and generate good representations of data in lower dimensions. Most of the applications of CNNs are in image segmentation, image labeling tasks. Apart from them, CNNs have been used in metric learning to generate representations in lower dimensional spaces. Standard CNN frameworks like GoogLeNet [12], AlexNet [6] are strong CNNs, and used for many transfer learning tasks on images.

2.3 Recurrent Neural Networks (RNN)

Conventional Neural Networks, lack the ability to retain information spread across time steps. On the other hand, in tasks like sequence predictions, the input and output at time instant $t - 1$ of the network affects the output and input at time $t$. So, there is a need for a network which could have some “memory” to store the past experiences and use it later on. RNNs do exactly this.

As the name suggests, Recurrent Neural Networks (RNNs) have a feedback loop, through which they can examine their output at time $t - 1$ as an input at time $t$. This feedback mechanism brings in them an aspect of memory. In case of sequential inputs, the hidden state of a RNN stores it.

Formally defined, consider an artificial feed forward network $A$, a sequential input $x_t : t = 0, \cdots, T$. Then a RNN can be represented as shown in Figure 2.2. The following mathematical relations hold for a RNN. (Here $h_i$ is the hidden state vector of a RNN at time instant $i$)

$$s_t = \Phi(Wx_t + Us_{t-1})$$

where $W$ and $U$ are two matrices, which are internal parameters of the RNN. RNNs can be visualised by unfolding them or unrolling them in time, over the time axis as shown in Figure 2.2. This way of viewing a RNN leaves it no different from any feedforward neural network.
Backpropagation in RNNs occurs using a special algorithm called BPTT (BackPropagation Through Time). The core idea behind BPTT is same as regular BP on the unrolled network, but the difference is that after unrolling the network, the same vectors $W, U$ need to be updated due to the gradients at each time step.

Applications of RNNs RNNs have been used successfully in sequence based tasks. Sequence-to-sequence models [11] are entirely based on RNNs. Neural Machine Translation, Language Modeling, Generation of text, etc are some more applications.

Disadvantages of RNNs

- RNNs are unable to model long term dependencies. In cases, where there is a need to recall something which happened far back in time, RNNs usually are expected not to work well in such situations.

- Vanishing and Exploding Gradients Problem In the BPTT updates for RNN weights, the information stored in the error function is backpropagated long back and suffers a lot of multiplications. If the value(s) being multiplied are less than 1, then over time, the gradient would tend to 0, while if they are > 1, the gradient can grow immeasurably large. This makes exact gradient descent updates in RNNs difficult, and tricks such as clipping the gradient need to be used.

LSTM (Long Short Term Memory) is a “gated” recurrent neural net, which can preserve long term dependencies and also doesn’t suffer from vanishing and exploding gradient problems.
2.4 Long Short Term Memory (LSTM)

As mentioned above, LSTM networks can help mitigate the problem of remembering long term dependencies and vanishing and exploding gradients. The key to LSTMs is the cell state $c_t$, the horizontal line running through the top of the Figure. The cell state $c_t$ is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged. On the other hand, removal or addition of information to the cell state, is carefully regulated by structures called gates. Gates are composed out of a sigmoid neural net layer and a pointwise multiplication (Kronecker Product $\otimes$) operation. A LSTM cell has 3 gates: a forget gate, an input gate, and an output gate.

- **Forget Gate** Takes as input $h_{t-1}$ (hidden cell state), $x_t$ and outputs $\alpha \in [0, 1]$. A 1 means completely retain this, and a 0 means completely forget this. See Figure 2.3

- **Input Gate** Decides what information is to be stored in the cell state. A sigmoid layer called input gate layer decides, what information is to be updated. A new candidate is generated by the tanh layer and then combination is done by $\otimes$. See Figure 2.4

- **Output Gate** To-forget information is removed from the cell state, new information is added to the cell state and now the cell state is given out. Hidden state $h_t$ is also updated. See Figure 2.5 and 2.6.
2.5 Attention Mechanism

In the case of temporal predictions, it is sometimes a good idea to focus on some particular parts of the space, to give out some predictions. Consider an example of the neural machine translation task. In this way it will be easy to see how attention works.

In NMT, we map the meaning of a sentence into a fixed-length vector representation and then generate a translation based on that vector. By not relying on things like n-gram counts and instead trying to capture the higher-level meaning of text, NMT systems generalize to new sentences better than many other approaches. (Source: WildML)

Usually, NMT tasks are implemented using a RNN encoder decoder module. These modules first of all take in the entire sequence as input into a RNN, and then start producing the outputs one by one. See Figure ??

The decoder is supposed to generate a translation solely based on the last hidden state ($h_3$ above) from the encoder. This $h_3$ vector must encode everything we need to know about the source sentence. But this is not always practically possible. An alternative could be a LSTM, or say some hackish way of passing the sentence in different orientations to the encoder module.

But attention is the solution which doesn’t require any encoding into a fixed length vector. We allow the decoder to "attend" to different parts of the source sentence at each step of the output generation. We let the model learn what to attend to based on the input sentence and what it has produced so far. (Source: WildML) [16]
Figure 2.7: An example of NMT Task Encoder Decoder (Source: WildML)

Figure 2.8: Example of Attention (Source: WildML)
2.6 Memory Models

RNNs are different from other deep net architectures and even, other machine learning methods due to their ability to learn and carry out complicated transformations of data over extended periods of time. RNNs are Turing-Complete [10] and therefore have the capacity to simulate arbitrary procedures, if properly tuned. Yet this is almost always not simple in practice. Therefore, Memory Augmented Neural Networks (MANNs) were introduced, where an additional external memory was augmented to the neural networks and the read-write operations into the memory could be learned by gradient descent and were controlled by the underlying neural network. Memory models have been shown to be quite useful in case of One-shot learning tasks, Meta-learning tasks and have brought us an inch closer in order to achieve the overall aim of AI.

We describe a general model (not specific to any application), called Neural Turing Machines [3], which were introduced in 2014, by the Google DeepMind Team.

2.7 Neural Turing Machines (NTM)

Analogous to a Turing machine from Theory of Computation, which contains a deterministic finite state automaton in the backend and an infinite tape on which the machine can write, a Neural Turing Machine consists of a runner Neural Network (CNN, Feed-Forward Network), and a piece of external memory. There are two heads, one read head, and one write head, which can read from and write to, respectively from the memory.

Reads and Writes Reads and Writes into the external memory module are blurry in nature, meaning that they are smooth functions, and along with it, employ an attention based focus mechanism in order to determine the locations in the memory that are read from or written to.

Addressing Mechanisms in the Memory Memory is a $N \times M$ matrix, where $N = \#$(memory locations); $M = size(Mem[i])$. Broadly two types of addressing mechanisms are used: Content Based Addressing and Location Based Addressing.

- **Content Based Addressing:** This kind of addressing helps the controller network focus attention on locations with similar current values and values emitted by itself. A key vector $k_t$ of length $M$ is produced by the head and it is compared to $M_t(i)$ by similarity measure $S[\cdot, \cdot]$ (For example, cosine similarity). Positive key strength $\beta_t$ is used to amplify or attenuate a focus on some location in the memory. The output is a weight vector analogous to the attention weight vectors we have seen in the previous sections and is given by $w_t^c$.

$$w_t^c(i) := \frac{\exp(\beta_t S[k_t, M_t(i)])}{\sum_j \exp(\beta_t S[k_t, M_t(j)])}$$
Figure 2.9: Neural Turing Machine Architecture

- **Location Based Addressing:** This addressing mechanism is used to focus attention on specific locations in the memory matrix. Location based addressing mechanisms need to handle issues like random-access-jumps and iterations (as we want the neural network to learn any procedure). These issues were solved by introducing a notion of a **shift-weighting**, which is used to spread focus across locations. To generate the attention weight vector for location based addressing, first of all, a gated weighting is generated. A constant \( g_t \in (0, 1) \) is used as a scalar to blend between the weightings \( w_t \) and \( w_{(t-1)} \). Gated weighting is given by

\[
 w^g_t \leftarrow g_tw_t + (1 - g_t)w_{(t-1)}
\]

Now, shifts are applied, using a shift weighting, which basically, defines a normalised distribution over integer shifts (for example, softmax or discrete values). Rotation is applied to \( w^g_t \) by \( s_t \) and can be formulated as a circular convolution:

\[
 \tilde{w}_t(i) := \left( \sum_{j=0}^{N-1} w^g_t(i)s_t(i-j) \right) \mod N
\]

Some sharpening can be applied to generate the final weights:\n
\[
 w_t(i) := \frac{\tilde{w}_t(i)^\gamma}{\sum_j \tilde{w}_t(j)^\gamma}; \gamma_t \geq 1
\]

These mechanisms give rise to different modes of addressing, purely content based addressing, content based + shift (this allows focusing on the top of a specific block and then move to a particular position in the block (arrays, structures)) and location based + shift (this allows iteration through a sequence of addresses by advancing same distance at each time step. (iterators, loops))

**Read operations** Now, any time when the head or the controller gives the key, and a mode of addressing is selected, weights are generated. Now, the read vector \( r_t \) of length \( M \) is defined as convex-combination of row vectors \( M_t(i) \) in memory:

\[
 r_t := \sum_i w_t(i)M_t(i)
\]
Write operations  Drawing motivations from a LSTM cell, a write operation is composed of two steps, erase and add.

[Erase Step ] Read Head emits $e_t$ (erase vector, used to focus on the information to be erased), $\forall i, e_t(i) \in (0, 1)$. Reset memory as:

$$\tilde{M}_t(i) \leftarrow M_{t-1}(i)[1 - w_t(i)e_t]$$

[Add Step ] Write Head emits $a_t$ (add vector, used to focus on the locations to be written to). Reset memory as:

$$M_t(i) \leftarrow \tilde{M}_t(i) + w_t(i)a_t$$

As both the reads and writes are differentiable, the process of learning reads and writes can occur via regular gradient descent methods.

Training  Training of a NTM involves regular gradient descent, with momentum [Cite Appendix], gradient clipping. The model is end-to-end differentiable.

Key experimental results  NTMs have been shown useful in tasks like copying (tests storage and recall of a sequence of information, in the paper, a NTM was trained with sequences of 8 bit randomly chosen vectors, with $|\text{Seq}| \in (1, 20)$. Found to generalize well to longer sequences, which was not observed in case of LSTMs), priority sort (Given a sequence of random binary vectors and a scalar priority for each sequence, learn to output the sequence in a sorted manner. Memory traces suggest that a hash table kind of algorithm was learned by the Controller), and various other tasks like n-gram modeling, associative recall, etc. Overall, Graves et.al. found that the performance of a NTM with a feed-forward controller also beat the performance of a LSTM.
Applications of NTMs  Applications of NTMs include one-shot learning tasks (see Chapter 3), learning to remember rare events. These days, modifications of NTMs have also been shown successful in finding shortest paths in graphs and other problems like hybrid computing. [4]
Chapter 3

Deep Learning Methods for Classification with less examples

In the last two chapters, we broadly discussed the different tricks and techniques in deep learning, and the basic deep learning models and architectures that are widely used in these paradigms. In this chapter, we shall review some of the recent literature on the various ways in which researchers have combined basic models to achieve good results in the original problem of our interest. I would also present some critique of each of the approaches and some of the points which form really good problems for future work.

We start with some of the approaches for performing good one-shot learning. Recall that one-shot learning is the problem where it is required to learn a good model given that there are limited examples of each class in the training data set.

3.1 Approaches to One Shot Learning

One shot learning has been solved mostly using Metric Learning approaches for example, Matching Networks for one-shot learning [13] and using Meta Learning with MANNs, for example, One Shot Learning using Memory Augmented Networks [9]. The former improves over the results of the latter.

3.1.1 One Shot Learning using MANNs

The authors are trying to exploit the storage and recall properties of a Memory Augmented Neural Network in this paper to solve the problem of One shot learning. According to them, the main aim
is to make the network learn to store instances in the memory and access them at query time for accurate predictions. They don’t want that the network learns the weights in order to be able to generate the label and predict the label. Instead they want the network to learn **sample-class bindings** and store it in the memory. The weights should learn higher level knowledge. (for example, how to store data into memory, how to compare query input and the memory, how to output the label) Naturally, this problem falls under the paradigm of Meta Learning.

**Task Description**  Being a Meta-learning algorithm, we wish to get optimal parameters

\[
\theta^* = \arg\min_\theta E_{D \sim p(D)}[L(D; \theta)]
\]

where \(D\) is a dataset. \(D = \{(x_t, y_t)\}_{t=1}^T\). \(y_t\) is a target label and is presented as input along with \(x_t\), in a temporally offset manner meaning that, the network sees the input sequence \((x_1, \text{null}), (x_2, y_1), \ldots, (x_T, y_{T-1})\). At time \(t\) the correct label for the previous data sample \((y_{t-1})\) is provided as input along with a new query \(x_t\). The network is tasked to output the appropriate label for \(x_t\) at the given timestep. Labels are shuffled from dataset-to-dataset to prevent memorization of the mappings. The network is expected to use a random guess for first occurrence of a particular class and there after use the storage in memory to predict 100% correct results thereafter.

**Memory Architecture**  The memory architecture in this paper is inspired from a Neural Turing Machine, which has a neural network controller, a read head and a write head and follows certain addressing mechanisms. All of this is the same here. The only difference is in the way memory is accessed. The authors define a new way of memory access, called the **Least Recently Used Access** (LRUA), something analogous to what the modern Operating Systems do. The LRUA module is a content-based addressing algorithm which writes to either the most recently used or the least used location. New information can be written to a new location, preserving the already written locations or can overwrite existing information.

The attention weights for the memory location is generated by keeping track of usage weights, \(w_t^u\), which are calculated according to the following \(w_t^u := \gamma w_{t-1}^u + w_t^r + w_t^l\). Least recently used weights for a given time-step can be computed as

\[
w_t^{lu} = \begin{cases} 
1 & w_t^u > m(w_t^u, n) \\
0 & w_t^u \leq m(w_t^u, n)
\end{cases}
\]

where \(m(v, n)\) denotes the \(n^{th}\) smallest element of vector \(v\), and \(n\) denotes the number of writes in memory. Write weights are calculated by using a convex combination of previous read weights and previous least used weights.

\[
w_t^w := \sigma(\alpha)w_{t-1}^r + (1 - \sigma(\alpha))w_{t-1}^{lu}
\]

Writing to memory occurs in accordance to these weights. Read operation is similar to the NTM module.
Experiments and Observations

The primary dataset used for testing was the Omniglot dataset. To prevent overfitting, many new images by rotating and translating the already present images were generated. The labels were encoded as a new kind of 5-hot encoding and were provided as inputs to the MANN module. It was found that during the first occurrences of the datapoints, the MANN made educated guesses but then from the second instance onwards the accuracy steeply increased to about 99%. An example of the output is shown in the figure ??.

The baselines were kNN, a feedforward LSTM, the conventional NTM (described earlier). The initial accuracy for the kNN classifier is around \( \frac{1}{N} \), where \( N \) is the number of unique classes in an episode. The MANN clearly outperformed the results of the baselines as shown in the graphs.

Persistent Memory Interference: It was observed that a good strategy was wiping the external memory from episode to episode, as each episode contained unique classes with unique labels. In case of no wiping, learning was affected, and the network performed badly as compared to the original wiping condition.

Curriculum Training: The module was subject to curriculum training where the number of classes was gradually increased in every 10000 episodes. It was observed that the performance

Figure 3.1: Example of training a MANN in this network
gradually started decaying close to 100. (All the earlier experiments were done on 5 and 15 classes)

My opinion of the paper would be that the paper presents a really nice and easy technique to achieve perfectly accurate results in case of one shot learning tasks. But, the experiments seem well suited to the LRUA access the authors proposed. Moreover, there was no label overlap across episodes. In case of label overlaps, I feel that their approach might not do this well as it might suffer from interference from the previous episodes (something similar to the persistent memory interference they observed). Secondly, the entire memory is accessed in the read operation to get the average read vector, which is not that scalable with high memory sizes. Also, if the number of labels or classes go up (here it is in the order $\approx 10^3$) then the memory requirements shoot up.

Follow-Up Work Some follow-up work for this paper includes experimenting with the performance of the method under label overlap and other not-so-well suited tasks. Also, working on new training procedures and algorithms for memory access, is a follow-up work for this paper.

3.1.2 Matching Networks for one-shot learning

This paper improves upon the paper on one-shot learning using MANNs discussed above. The paper presents a non-parametric approach towards solving the problem of one-shot learning. The paper casts the problem of one-shot learning in the set-to-set-framework.

Model Architecture Given a small support set of $k$ examples of image-label pairs $S = \{(x_i, y_i)\}_{i=1}^k$ to a classifier $c_S(\hat{x})$ which, given a test example $\hat{x}$, defines a probability distribution over outputs $\hat{y}$. We define the mapping $S \rightarrow c_S(\hat{x})$ to be $P(\hat{y}|\hat{x}, S)$ where $P$ is parameterised by a neural network.

The overall idea is to build a nearest neighbour classifier, but a differentiable one.

$$\hat{y} = \sum_{i=1}^{K} a(\hat{x}, x_i) y_i$$

where $a$ is a kernel which computes the similarity of $\hat{x}$ and $x_i$, where $y_i$ are one-hot vectors. Now in order to encode $a$, the most natural idea would be to encode $a$ as a softmax of similarities between embeddings of $\hat{x}$ and $x_i$. So,

$$a(\hat{x}, x_i) = \frac{e^{c(f(\hat{x}), g(x_i))}}{\sum_i e^{c(f(\hat{x}), g(x_i))}}$$

Embedding with training examples In order to generate the embeddings, a function $g(.)$ is parametrised by a bi-directional LSTM. Consider $g'(x_i)$ to be the neural network parameterizing the embedding of $x_i$.

$$\overline{h_i, c_i} = LSTM(g'(x_i), h_{i-1}, c_{i-1})$$
\[
\hat{h}_k^b, \hat{c}_k^b = LSTM(g'(x_i), h_{i+1}^b, c_{i+1}^b)
\]

Here ‘b’ denotes the updates for the backward layer of the bi-LSTM.

**Embedding the test examples** The function \( f \) is an LSTM that processes for fixed amount (K steps) and at each point over the examples in the training set. The encoding is the last hidden state of the LSTM. Again, this way we’re allowing the network to change its encoding of the test example as a function of the training examples.

\[
\hat{h}_k, c_k = LSTM(f'(\hat{x}), [h_{k-1}, r_{k-1}], c_{k-1})
\]

\[
h_k = \hat{h}_k + f'(\hat{x})
\]

\[
r_k = \sum_{i=1}^{S} a(h_{k-1}, g(x_i)) g(x_i)
\]

\[
a(h_{k-1}, g(x_i)) = \text{softmax}(h_{k-1}^T g(x_i))
\]

This is just a vanilla LSTM with attention where the input at each time step is constant \( f'(\hat{x}) \).

**Training Procedure** As pointed out in the previous paper, the test-time protocol (given N novel classes with only k examples each (k is very small, and task is to predict new instances to one of N classes) to exactly match the training time protocol. Therefore, in order to create each episode of training from a dataset of examples, we sample a task T from the training data, e.g. select 4 labels, and up to 5 examples per label (i.e. 5-20 examples). In order to form one episode sample a label set L (e.g. \{cats, dogs\}) and then use L to sample the support set S and a batch B of examples to evaluate loss on.

**Experiment Results** This paper presents promising experiment results, and they, in fact seem to beat the previous paper. The primary dataset used for testing was the Omniglot dataset. The paper also proposes a new dataset which is derived from Imagenet, and proves its efficacy over it too. As a consequence, it seems that by learning good metrics it seems that tasks like One-shot learning can be easily cracked.

**My opinion of the paper** As a whole, I would say that the idea of developing a end-to-end differentiable nearest neighbour module is commendable. On the other hand one of the primary concerns is that as the data size grows, the size or the space to be used by this algorithm will go really high, so we cannot use this for the other tasks. There are a few questions like variable dataset sizes, possibilities of online learning, and other concerns which need to be answered here.
3.2 Approach to robust Metric Learning

As we have seen in the previous section that the paper [13] improves upon [9] by using learning suitable transformations of the limited input datapoints into a space where just a simple \(k\)-NN is able to classify them well. This is what metric learning seeks. Several deep learning approaches have come up in metric learning. We will be discussing one paper called Metric Learning for Adaptive Density Discrimination. [8]

3.2.1 Metric Learning for Adaptive Density Discrimination

Existing approaches for Distance Metric Learning (DML) involve usage of either class labels (a single scalar value) to summarize the notion similarity, (that is, equality of class label implies lesser distance between two points in the transformed space and vice versa), which is not justified at all, as a high dimensional vector is being converted to just a single value which decides its fate in the transformed space. This approach can artificially impose a lot of distortion of the internal structure of the points in the original input space.

Other class of approaches towards DML include using local similarity (concerning neighbours of an example and using them to determine the representation of the example in the transformed space), but these approaches preserve the initial distance relations in the input space. But, this contradicts the basic need for DML, which is to learn an appropriate distance metric as existing metrics in input space don’t help us in the global task we aim to solve. Moreover, none of the existing approaches capture inter-class similarity and intra-class variation.

**Existing Objective Formulations**  A lot of the existing approaches on DML use the notion of **Triplet Loss**. (See Figure 3.2) Triplets are 3-tuples of points \((r_m, r^+_m, r^-_m)\), where \(r_m\) is called the seed point, \(r^+_m\) is a positive data point and \(r^-_m\) is a negative data point (assume binary classification as of now). The overall goal of triplet loss is to bring points of same classes together, and hence it would penalize the distance between the seed (usually a sampled positive point) and the positive point, and the negative of the distance between the negative point and the seed.

\[
L(\Theta) = \frac{1}{M} \sum_{m=1}^{M} \left[ \|r_m - r^+_m\|^2 - \|r_m - r^-_m\|^2 + \alpha \right]_+
\]

where \(z = [\cdot]_+\) denotes the hinge loss function. Triplet loss takes only triplets of points to minimize the loss and is hence short-sighted. Not the global structure, but just three points are taken into account while training.

The authors hence propose a new loss for DML which respects globality, inter-class similarity and intra-class variation.
Model Description  Input dataset is given by \( \mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N} \) and the points are spread across \( C \) classes. The task is to output \( r_n = f(x_n; \Theta) \), where \( f \) is parametrized by a neural network (here GoogLeNet [12] has been used; we also mentioned that CNNs are good for learning representations) and \( r_n \) is the representation/transformation of \( x_n \) in some space.

For each class, with an application of K-means algorithm, initial \( K \) clusters assignments are obtained. Assignments chosen to minimize intra-cluster distances.

\[
I_{c_1}, ..., I_{c_K} = \operatorname{argmin}_{I_{c_1}, ..., I_{c_K}} \sum_{k=1}^{K} \sum_{r \in I_{c_k}} ||r - \mu_{c_k}||^2
\]

\[
\mu_{c_k} = \frac{1}{|I_{c_k}|} \sum_{r \in I_{c_k}} r
\]

Magnet Loss for DML  Figure 3.2 Magnet loss takes into account the effect of larger neighborhoods and a neighborhood structure. The magnet loss is now defined over the set of clusters, instead of triples.

\[
\mathcal{L}(\Theta) = \frac{1}{N} \sum_{n=1}^{N} \left\{ - \log \frac{\exp\left(-\frac{1}{2\sigma^2} ||r_n - \mu(r_n)||^2 - \alpha\right)}{\sum_{c \neq C(r_n)} \sum_{k=1}^{K} \exp\left(-\frac{1}{2\sigma^2} ||r_n - \mu_{c_k}||^2\right)} \right\} + \text{ }
\]

where \( \sigma^2 \) is the variance of all the examples from their cluster centers, and \( \alpha \in \mathbb{R}_+ \) is a scalar. \( \alpha \) is the desired inter-cluster gap in variance units. Also, if the cluster centers are sufficiently far from a particular example then it vanishes from its term in the objective. This allows approximation of each term with a small number of closest clusters. Another feature is that magnet loss penalizes the close-by, overlapping clusters heavily. It tries to ensure large distance between clusters. Keeping \( \alpha \) in variance units helps in variance standardization.

Training Procedure  This is something which I found to be new as compared to other papers. Their approach of sampling training examples is something new. During training, we need to sample local neighbourhoods (rather than individual examples) randomly and use it as a mini-batch. The algorithm is given below.
1. Sample a seed cluster $I_1 \sim p_I(.)$
2. Find $M - 1$ nearest impostor clusters $I_2, ..., I_M$ for $I_1$
3. Sample $D$ points from each cluster: $x^m_1, x^m_2, ..., x^m_D \sim p_{I_m}(.)$

$p_I(.)$ and $p_{I_m}(.)$ allow adaptation to distributions of examples in the representation space. $p_I(I) \propto \mathcal{L}(I)$(mean loss) once we have a few examples of $I$. (examples are cached for this use) This is something I found interesting, as sampling from a distribution of the loss means that points which incur high loss will be selected and will be separated due to the training, which should make the training process faster. Stochastic approximation of the objective:

$$\mathcal{L}(\Theta) = \frac{1}{MD} \sum_{m=1}^{M} \sum_{d=1}^{D} \left\{ -\log \frac{\exp \left( -\frac{1}{2\hat{\sigma}^2} || r_m^d - \mu ||^2 - \alpha \right)}{\sum_{\mu : C(\mu) \neq C(r_m^d)} \exp \left( -\frac{1}{2\hat{\sigma}^2} || r_m^d - \mu ||^2 - \alpha \right)} \right\}$$

**Updating the clusters:** The cluster means are refreshed periodically, initialized by K-Means++

**Advantages over Triplet Loss:** Penalizing clusters of points away from one another leads to a more coherent adjustment of each point. Moreover, as it updates the entire cluster at once, the process requires less iterations. Also, the complexity is less than $O(n^3)$ of triplet loss.

**Evaluation Procedure** Label of each example $x_n$ is a function of its representationâ€™s $\{r(x_n)\}$ softmax similarities to its $L$ closest clusters. A label for a data point $x_n$ is chosen as follows:

$$c^*_n = \arg \max_{c=1, ..., C} \frac{\sum_{\mu : C(\mu) = c} \exp \left( -\frac{1}{2\sigma^2} || r_n - \mu ||^2 \right)}{\sum_{k=1}^{K} \exp \left( -\frac{1}{2\sigma^2} || r_n - \mu_k ||^2 \right)}$$

where $\sigma$ is the running average of $\hat{\sigma}$ accumulated during training and $\mu_1, ..., \mu_L$ are the $L$ nearest means to $x_n$. This has been called the $k$-Nearest Clusters algorithm.

**Experiments and Observations** The primary task in the experiments was Fine-Grained classification on datasets like Stanford Dogs, Oxford-IIIT, etc. The results suggest that magnet loss defeats triplet loss by a substantial margin. Sometimes, even the simple softmax classifier surpassed the triplet loss. Moreover, triplet loss achieves a particular accuracy 5-30 times faster than triplet.

Another task was the task of attribute distribution on the ImageNet Attributes dataset. Magnet outperforms triplet and softmax in terms of concentration of attributes at a particular region in the state space. A similar task was of hierarchy recovery, where again Magnet dominated.

**My opinion of the paper** The paper presents some really good ideas, like magnet loss itself, sampling from a distribution proportional to the loss, etc. But there are certain things, which can be improved or at least experimented with. It is not very clear as to why the number of clusters for each class is kept fixed at the same constant. It should have been better if they could have used an
adaptive rate of changing $K$. Secondly, periodic updates are not as meaningful with respect to me. A good idea would have been to represent the transformation function by the CNN and added a new point for the mean of each of the clusters into the dataset, (so, $K \times C$ such new points) and also update these means during backpropagation. Also, suddenly stopping the network, updating the means dynamically, and restarting backpropagation can give non-linear, drastic jumps on the surface of the loss function and it is not very clear if this would go through without any problems, meaning that backpropagation would be easy in such scenarios.

**Follow up work** Some follow-up work could include incorporating a variable index cluster, so extending the algorithm to the point with variable number of means for each class which could be adapted. Apart from this, as it is well known, K-Means clustering is not a very high accuracy-giving algorithm for classification, so probably replacing the K-Means algorithm with other algorithms might give better estimates of neighbourhoods and better convergence.

### 3.3 Approaches to optimize gradient descent

In the previous section, we saw how better metrics can be learned using CNNs, which can indirectly benefit in one-shot learning tasks by improving accuracy of algorithms which use metric learning approaches for solving one-shot learning tasks.

Gradient Descent forms the basis of Deep Learning. Gradient Descent has evolved a lot over time, and is one of the first algorithms that has formed the foundation of deep learning. One drawback of tuning parameters using less amount of data (which usually occurs with rare classes and in one-shot learning tasks) is that the parameters cannot be tuned to optima, and this can lead to catastrophic inference.

Moreover, as already well known, initialization matters substantially in gradient descent optimizations. With certain initialization, it might be easy to reach the global minimum, while with others, completely difficult. For each separate dataset considered, the network would have to start from a random initialization of its parameters, which considerably hurts its ability to converge to a good solution after a few updates.

The following two papers, Learning to learn by gradient descent by gradient descent [1] and Optimization as a model for few shot learning [7] present general and specific, problem directed meta-learning approaches respectively towards (1) learning to do gradient descent in a learner (2) learning to propose a good initialization for the learner. In each of these approaches, there are two learning subsystems: Meta-learner (the network which facilitates gradient descent of the other subsystem) and the Learner (the network which actually predicts the outputs for the input data, and whose gradient descent and initialization we wish to optimize).
Overall Idea of the two papers  To create a meta-learner which would propose gradient descent updates to the learner subsystem and thereby, learn a high level method of predicting updates for the learner. Both the papers model the problem in different manners and hence we present both the formulations one-by-one.

Formulation #1  [1] The gradient descent update is represented as \( \theta_{t+1} = \theta_t + g(\nabla f(\theta_t), \phi) \) where \( g(.) \) is the function that parametrizes the update proposed by the meta-learner. For the meta-learner proposing a good update, means minimizing the expected best case loss over a distribution of functions \( f \), that is

\[
L(\phi) = E_f [f(\theta^*(f, \phi))]
\]

Here \( g(.) \) is parametrized by a RNN whose hidden state will be denoted by \( h_t \). For training the meta-learner (optimizer), it will be convenient to have an objective that depends on the entire trajectory of optimization not just the end result. For example, \( L(\phi) = E_f [\sum_t w_t f(\theta_t)] \) and \( \theta_{t+1} = \theta_t + g_t \) and \( [g_t, \theta_{t+1}]^T = m(\nabla_t, h_t, \phi) \) where \( m \) is the RNN. The weights \( w_t \) can be calculated randomly.

The optimization procedure on this function is easy. The gradient \( \frac{\partial L}{\partial \phi} \) is calculated by sampling a random \( f \) and applying backpropagation. A schematic is given in Figure 3.3. Gradients along dashed edges are ignored. For example, \( \frac{\partial L}{\partial g_t} = 0 \) as this prevents computation of second derivatives.

Formulation #2  [7] This paper has broader goals in order to ensure gradient descent updates for the learner along with proposing suitable initializations for the learner. We first define the notion of a dataset for meta learning. A meta-set \( \mathcal{D} \) consists of datasets \( D \) split into \( D_{\text{train}}, D_{\text{test}} \). There are three different meta-sets: \( (D_{\text{meta-train}}, D_{\text{meta-test}}, D_{\text{meta-validate}}) \). The overall task here is the \( k \)-shot, \( N \)-class classification task, such that \( D_{\text{train}} = k \times N \) examples and \( D_{\text{test}} \) consists of some fixed number of examples.

As the authors say, the main motivation for their work is derived from the LSTM cell update.
According to the authors,

$$c_t = f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t$$

Gradient Descent Update:

$$\theta_t = \theta_{t-1} - \alpha_t \nabla_{\theta_{t-1}} L_{t-1}$$

Both the rules look quite similar and can be made identical by setting $\theta_k = c_k, \alpha_t = i_t, \tilde{c}_t = -\nabla_{\theta_{t-1}} L_{t-1}$. So, it is a good idea to introduce a meta-learner LSTM such that the cell state of the meta-learner LSTM $c_t$ is equal to the parameter $\theta_t$ of the learner. Learning rate of gradient descent will then be given by $i_t = \sigma(W_I \cdot [\nabla_{\theta_{t-1}} L_{t-1}, L_t, \theta_{t-1}, i_{t-1}] + b_I)$. Depending on the parameters $\theta_{t-1}$, loss $L_t$ learn to make adaptive updates. The optimal choice for $f_t$ doesn’t seem to be 1, hence we can model it as to $f_t = \sigma(W_f \cdot [\nabla_{\theta_{t-1}} L_{t-1}, L_t, \theta_{t-1}, f_{t-1}] + b_F)$. Depending on the parameters $\theta_{t-1}$, loss $L_t$ control dependency on previous cell state or the previous parameters, and this allows us to adaptively perform gradient descent. For example, in case of a ravine, we might decide to forget the previous parameter values and jump to a new location. Also learns the optimal initialization $c_0$.

**Model Description** Having given both the formulations, the model descriptions for the two methods coincide to a large extent. So, we discuss them together. There are two main issues, parameter sharing and gradient pre-processing.

- **Parameter Sharing:** Parameter sharing is required in this approach as when applied to few-shot learning, the number of parameters of the meta-learner to optimize should be less.
There is 1 LSTM for each coordinate independently. The parameters of the optimizer/meta-learner LSTM are the same for all parameter coordinates of the learner. But hidden states and function activations for each parameter coordinate are different, leading to different behaviour on different parameter coordinates/parameters. Another benefit is that, this makes the method order-independent as the same update is used for each parameter coordinate irrespective of the order. (which is beneficial for non-sequential data)

• Gradient Preprocessing: As NNs disregard small weights and focus on big ones, there should be a few gradients which shadow the effect of other gradients. Gradients with respect to learner parameters vary a lot. Both the papers, therefore normalize the gradients as given below.

\[
\nabla^k = \begin{cases} 
\left( \frac{\log(|\nabla|)}{p}, \text{sgn}(\nabla) \right) & \text{if } |\nabla| \geq e^{-p} \\
(-1, e^{p} \nabla) & \text{otherwise}
\end{cases}
\]

The training algorithm for Formulation #2 is summarized in the Figure 3.5. An important point to note is that the loss of the learner in each step is the loss on \(D_{test}\), thereby trying to match train and test conditions (relevant in one shot learning scenarios)

Experiments and Observations In [7], the details of the network are as follows. The learner network was a CNN with 4 conv layers, \(3 \times 3\), 32 in number filters, batch normalization, ReLU
activation and finally $2 \times 2$ max pooling. Fully connected layer involved softmax over the number of classes. The loss: $\mathcal{L} = \text{avg negative log-probability assigned by learner to correct class}$. The meta-learner LSTM was a 2-layer LSTM, first layer being a normal LSTM, to take inputs and other bookkeeping, and the second modified meta-learner LSTM which implements the update rule. ADAM optimizer was used over the loss of the learner.

The primary dataset used was the *Mini-Imagenet* dataset [13]. Baselines considered were k-NN, modified k-NN (both use metric learning), and Matching FCNs [13]. It was found that the meta-learning LSTM approach dominates by a small margin in 1-shot classification while by a huge margin in 5-shot classification tasks.

**My opinion of the papers** I feel that the overall idea of both the papers is quite interesting. The papers present an efficient and a promising approach to meta-learning. Firstly, keeping test time and train time approaches same is ensured which is one of the key factors that could help in the success of meta-learning. The number of parameters is effectively kept under control, by constraining the cell state of the optimizer LSTM to be the parameter vector of the learner network. The idea of learning to predict a possible initial values of the parameters of the learner network helps them reach better optima, which again is a very good idea.

The difference between the papers is that the [1] aims at solving a more general meta-learning problem, not specific to one-shot learning, whereas the [7] attempts at solving a specific problem. The [1] just parametrizes the updates as a function of the gradients $\nabla_{t}$ and not the loss itself. This is a bit unnatural, as I am not sure if this can help in situations where loss is high, but the gradient of the loss is almost 0.

On the other hand, [1] doesn’t enforce linear kind of gradient descent updates, by this I mean, that the function $g(.)$ can also involve effect of higher order derivatives, which is not the case with the other paper.

I feel that a good strategy can be developed just using these papers, by combining the good points. We use non-linear updates and along with it, also predict the initializations.

**Follow-up work** A possible follow up work, could be to extend the approach in this paper to general settings, not only few-shot learning, but also scenarios like class imbalance, instance imbalance and try to compete against the state-of-the-art gradient descent update methods.

### 3.4 Handling Class Imbalance Scenarios

Having seen much of one shot learning, we also explore a bit of the recent deep learning approaches on another celebrated problem, which is to learn a model which can predict accurately on all classes
given that there is a huge imbalance in the prior distribution of the classes in the dataset. Here, we indirectly deal with data insufficiency, as the network will get trained in predicting the major classes accurately, while even totally adverse predictions on negative classes will not decrease the overall accuracy (on all data points in the dataset).

We discuss a paper from CVPR 2015, Learning Deep Representation for Imbalanced Classification [5]. The paper focuses on the general class imbalance scenario in two central problems in vision, face attribute recognition and edge detection.

### 3.4.1 Learning Deep Representation for Imbalanced Classification

As we have seen until now, in the previous sections on one-shot learning that the one of the best ways of escaping from the less amount of data is by using proper deep representations, which can be achieved using metric learning.

**Conventional Machine Learning Methods** Conventional Machine Learning methods use tricks like oversampling the minority class data points in order to equalize the class priors to some extent, or undersampling the majority classes. But, a major disadvantage associated with these methods is that either we are overfitting (oversampling) or losing out the information given (undersampling).

Another method has been to penalize the misclassification of points in rare classes more than the majority classes. This method works well for shallow models and has not been tested for deep models.

**Overall Idea of the paper** Minority classes often exhibit highly variable distributions. Therefore, learn an embedding $f(x) \in \mathbb{R}^d, \|f(x)\|_2 = 1$ such that embeddings are distinctly separable in the representation space $\mathbb{R}^d$. In order to do this, the paper proposes a new kind of sampling called **Quintuplet Sampling** and a new loss, called **Triple Header Hinge Loss** which is derived on similar lines as Triplet Loss and Magnet Loss [8]. Further, they propose a max margin version of k-NN called Large Margin Nearest Neighbours.

**Quintuplet Sampling** Quintuplet sampling is a new sampling technique which used to sample quintuplets. Quintuplets are 5-tuples of points $(x_i, x_i^{p+}, x_i^{p-}, x_i^{p--}, x_i^n)$ where $x_i$ is called the anchor, $x_i^{p+}$ is the anchor’s most distant in-cluster neighbour, $x_i^{p-}$ is the nearest within-class neighbor of the anchor, but from a different cluster, $x_i^{p--}$ is the most distant within-class neighbor of the anchor and $x_i^n$ is the nearest between-class neighbor of the anchor, such that the following relation holds:

$$D(f(x_i), f(x_i^{p+})) < D(f(x_i), f(x_i^{p-})) < D(f(x_i), f(x_i^{p--})) < D(f(x_i), f(x_i^n))$$

The intuition behind quintuplets is that two instances are close if they belong to the same class and the same cluster too. This helps build a local classification boundary with the most
discriminative local samples. One more advantage of this method is that, irrelevant samples in a class are ignored for class separation, making local boundary robust and insensitive to imbalanced class sizes.

**Triple-Header Hinge Loss** Triple-Header hinge loss is based on the idea of maximizing the margin between the clusters, just like SVMs. Hence, the optimization problem is framed in the similar way. We introduce three slack variables, $\epsilon_i, \tau_i, \sigma_i$ for each quintuplet and hence have the following optimization problem.

$$\min \sum_i (\epsilon_i + \tau_i + \sigma_i) + \lambda ||W||^2 \text{ such that}$$

$$\max(0, g_1 + D(f(x_i), f(x_i^{p+})) - D(f(x_i), f(x_i^{p-}))) \leq \epsilon_i$$

$$\max(0, g_2 + D(f(x_i), f(x_i^{p-})) - D(f(x_i), f(x_i^{p+}))) \leq \tau_i$$

$$\max(0, g_3 + D(f(x_i), f(x_i^{p+})) - D(f(x_i), f(x_i^{p-}))) \leq \sigma_i$$

$$\forall i, \epsilon_i \geq 0, \tau_i \geq 0, \sigma_i \geq 0$$

$g_1, g_2, g_3$ can be set using geometric visualizations and practically in experiments by using techniques like grid-search. The authors say that the advantage of quintuplet loss over triplet loss is that it is more robust to invasion of imposter neighbours, and doesn’t distort the class structure.

**Training Procedure** The algorithm for training is mentioned below.

1. Cluster for each class by k-means using the learned features from previous round of alternation.
   
   In the first round, hand-crafted features or prior features obtained using pre-training are used. The number of clusters are variable, but the number of elements in a cluster are fixed.
2. Generate quintuplet table using the cluster and class labels from a random subset (50%) of training data.

3. **CNN training**: Sample mini-batches equally from each class, retrieve corresponding quintuplets from offline stored table.

4. Feed quintuplets into 5 identical CNNs, compute the loss with cost-sensitivities and then perform backpropagation.

5. Alternate between steps 1-2 and 3-5 every \( n \) iterations until convergence.

The architecture and the details on the training are described in Figure 3.7. This architecture describes a method for learning a good representation. Now, we need to classify the points based on the metrics learned.

**Nearest Neighbour Classification** A traditional k-NN classification predicts the label of a particular data point, by taking into account the labels of its \( k \)-closest neighbours. However, k-NN suffers from one major drawback, which is that it ignores the distribution of classes in a dataset, and makes the equal – class – density assumption. So the performance in the imbalanced case would degrade.

The authors propose a new modification to the conventional k-NN called **Large Margin Nearest Neighbour**. According to LMNN, a data point is labeled in the following way:

\[
y_q = \arg \max_{c \in [1, C]} \left( \min_{m_j \in \phi(q), y_j \neq c} D(f(q), f(m_j)) - \max_{m_i \in \phi(q), y_i = c} D(f(q), f(m_i)) \right)
\]

where \( c \) is the set of all classes, \( \{m_i\} \) is the set means in the neighbourhood of \( q \) and \( \phi(q) \) denotes the neighbourhood of \( q \). Intuitively, this means labeling \( q \) with the label whose one cluster means creates the most robust boundary. As it operates on the cluster centroids, rather than data points, it is faster than conventional k-NN.
Experiments and Observations  There were two primary experiments, one of face-attribute recognition and the other of edge detection. Both of these are binary classification tasks but exhibit highly skewed datasets. The evaluation metric that was average weighted accuracy, given by

$$\text{accuracy} = 0.5\left(\frac{t_p}{N_p} + \frac{t_n}{N_n}\right)$$

As the imbalance amount increases, their approach outperforms the other competitors at that time and offers an accuracy of 84% in the attribute recognition task.

My opinion of the paper  Overall, I think the broad idea behind the approach is an obvious one, but the modifications they made to suitably adapt it to this problem are really good. The idea of modified Large Margin kNN is quite interesting. However, I am not very convinced with the idea of quintuplets, as they strongly enforce some conditions which seem more than what is needed for a proper classification, therefore, I believe, probably at least as good results can be obtained by relaxing the distance inequality.

Moreover, they claim that their method does not involve re-sampling of data to disturb the prior data distribution, but it seems to me that the generation of quintuplet samples where they ensure that there are equal samples of each class in a minibatch is a bit misleading and contradictory to the fact mentioned.

Follow up work  The current algorithm requires re-sampling of quintuplets periodically which prevents it from being end-to-end model, and therefore, a follow up could be to replace it by a smooth, differentiable method.
Chapter 4

Conclusion and Future Work

In this report, we looked at some of the current hot-topic problems in deep learning research, which includes two main problems, the first one being of few-shot learning, where the goal is to make the learner learn from data, given that the training data is just too limited, and the second one being the problem where the class priors in the training data is skewed to a large extent, and still the learner has to ensure accurate predictions on the minority classes in the test set.

For the first problem of one-shot learning, we looked at methods involving meta-learning using MANNs, where the aim was to make the external memory module store the data it has never seen with the label mappings and retrieve it during test time for predictions. The second approach we saw was the approach of learning a good metric using the support training set given, to match test and train time conditions, which helps the differentiable nearest neighbour network learn to do one-shot learning. This paper of learning good representations improves over the MANN paper. Both these approaches suffer from the drawback that the overall memory/size requirements are too high, for example, in the MANN case, as the data size grows, more amount of memory is needed, and for Matching networks, again the amount of time and space needed for generating embeddings is high.

Further, we looked at how metric learning can be improved to learn better metric transforms which could aid in the process of one-shot learning. We introduced magnet-loss, a new loss for DML, and a \( k \)-Nearest cluster method of training, which seems to be an extension of the NCMC work, but models inter-class variation and intra-class similarity in order to ensure robustness at boundaries and the preservation of the structure.

We then looked at how meta-learning methods could improve the gradient descent process in case of few examples. We looked at two different approaches for the same, in two different, but yet similar papers. We saw that along with a simple LSTM too, but with the modified meta-learned gradient descent, high accuracies as close to Matching Networks can be achieved.
Finally, we moved our focus to the other problem, we are focusing on, which is of classification under imbalanced class distributions. We looked at a metric learning approach of quintuplet sampling and a max-margin loss called **triple-header hinge loss** which aims to achieve robust and separated class boundaries. We also saw a new version of the $k$-NN which discriminates neighbours based on the classes assigned to them, called the Large Margin Nearest Neighbour (LMNN). This approach works great and is able to perform in case of class imbalance scenarios.

Future work in this field would include work on combining meta-learning approaches along with suitable metric learning approaches to achieve better accuracies in case of few shot learning, moreover, it would be worthy to investigate if the task of classification with imbalanced class priors could be solved using the meta-learning framework. More problem-specific loss functions can be proposed and experimented with in case of metric learning, and maybe when all of these methods and techniques fit in synergy with each other, it might give rise to some new state-of-the-art approach in the field of AI.
Bibliography


