# Scaling up Multidimensional Recurrent Neural Networks for image segmentation

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Intelligent Systems

presented by
Marco Romelli

under the supervision of
Prof. Jürgen Schmidhuber
co-supervised by
Prof. Matteo Matteucci

September 2015

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Marco Romelli
Lugano, 7 September 2015

*To my family*

An algorithm must be seen to be believed.

Donald E. Knuth

# Abstract

In the machine learning community, interest in neural networks and deep learning have recently seen a continuous growth. In this context Multidimensional Recurrent Neural Networks (MDRNNs) allowed to bring the power of recurrent neural networks (RNNs) to data other than simple sequences and they have the potential to advance the state-of-the-art in many challenging problems, like image segmentation or video classification.

Given this interest in MDRNNs, the main contribution of this thesis is an efficient implementation of an MDLSTM layer, which is the Long Short Term Memory (LSTM) extension of MDRNNs. Moreover a simple CPU parallelization of the algorithm is implemented and evaluated against the single core version. We also show that this parallelization achieves speedups comparable to what has been achieved on convolutional neural networks with GPU implementations.

Finally the MDSLTM layer is integrated in an hybrid architecture with convolutional layers to perform semantic image segmentation, which is a difficult unsolved problem in computer vision. This architecture achieves near state of the art results on a popular image segmentation benchmark with no need for any pre or post processing.

# Acknowledgements

I would like to thank Jürgen Schmidhuber and Matteo Matteucci for their availability to supervise this thesis. I would also like to thank IDSIA for having provided me with the hardware I needed.

A special thank goes to Klaus Greff and Rupesh Srivastava for their precious tutoring along all the duration of this work.

A thank goes also to all my classmates at Università della Svizzera Italiana, with whom I shared good times and adversities. I'm also grateful to Cinzia and Ivan, for their willingness to listen to my thoughts.

Most of all, I would like to thank my family for their constant love and support, especially in the difficult moments.

# Contents

# Figures

# Chapter 1

# Introduction

Neural networks are a powerful mathematical model widely used in machine learning to tackle problems that are not easy to address with traditional methods. Such problems are typically easy for humans but extremely hard for a machine, so research is still going on to find better ways to accomplish these tasks. Neural networks are particularly useful in those contexts since they are able to automatically extract the information needed to solve the problem and many variants have been proposed in the past years.

In this thesis we focus on multidimensional recurrent neural networks (MDRNNs) [Graves et al., 2007]. This network architecture is an extension of recurrent neural networks which allows to handle inputs that are not limited to a single time dimension, allowing to handle images, videos or even more complex data like magnetic resonance imaging. This allows to bring the power of recurrent neural networks (RNNs) to a new set of problems which have an intrinsic multidimensional structure.

One of such problems is semantic image segmentation, in which each pixel of a given image needs to be labeled into one out of $n$ categories. Many approaches have been proposed in the literature to solve the problem, but none of them is considered the definitive solution. Since MDRNNs allow to handle 2-dimensional data in a natural way, they fit well to this problem and have the potential to advance the state-of-the-art.

## 1.1 Contributions

#### MDRNN implementation

The main contribution of this thesis is represented by an efficient implementation of a MDRNN which is versatile enough to be used by the scientific community.

With this objective in mind, an MDRNN layer is implemented in the open source library Caffe [Jia et al., 2014], which allows to combine it with a vast catalog of other layers.

Parallelization on CPU

To better exploit the power of MDRNNs, a parallel CPU version is implemented in order to be able to train larger networks than what is found in the literature. OpenMP is used to parallelize the code and the performance of this implementation is evaluated in order to quantify the achieved speedup.

Semantic image segmentation benchmark

The realized implementation is evaluated on SIFT FLOW [Liu et al., 2008]; a semantic image segmentation benchmark. The results are analyzed and compared to other state-of-the-art methods. In particular, this thesis gives two main contributions to the field:

- A novel architecture combining the MDRNN layer with convolutional neural networks (CNNs). This approach would allow the network to take advantage from the known benefits of CNNs, while using the power of MDRNNs to extend the capabilities of the network with long term dependencies.

- The training of the biggest MDRNN network in the literature, thanks to the efficient CPU parallelization. Implementations so far have been limited by their single core performance; for this reason we try to use the advantages of a parallel implementation to train a particularly big network and compare its results to the smaller ones.

## 1.2 Thesis structure

We begin in Chapter 2 with an introduction on neural networks and machine learning, starting with a brief history of the multilayer perceptron and the corresponding training with backpropagation. We then focus on convolutional neural networks and the innovations that came with them. Finally this chapter introduces recurrent neural networks and their training algorithm.

Chapter 3 focuses on multidimensional recurrent neural networks. It first explains the general principle of MDRNNs and the corresponding training equations. Subsequently it explains the extension of MDRNNs with LSTM units, called multidimensional long short-term memory (MDLSTM). Finally this chapter gives a brief review of the most important work related to MDRNNs in the literature.

In Chapter 4 we introduce the problem of image segmentation. We start with a definition of the problem with its variants and then we explain conditional random fields, which are the most used method in the literature used to solve this problem. The chapter concludes explaining how neural networks apply to semantic image segmentation and introduces a novel network architecture involving a combination of CNNs and MDRNNs to approach the problem.

Chapter 5 focuses on the implementation details and design decisions made during the development process. The first part explains the choice of Caffe as the supporting tool and gives an overview of the single core implementation. Afterwards a CPU parallelization approach is proposed and the corresponding realization is evaluated against the single core version.

Finally Chapter 6 applies the MDLSTM layer to the problem of image segmentation, focusing on the SIFT FLOW dataset. We first describe the dataset structure and then analyze the results of the best network architectures focusing in particular on the pixel-wise accuracy.

# Chapter 2

# Machine learning and neural networks

This chapter provides an overview on artificial neural networks, focusing in particular on the network architectures that are most related to this work. Section 2.1 contains an overview of neural networks in general, with a brief explanation of their applications in machine learning. In Section 2.2 we discuss convolutional neural networks, which own most of the state of the art results on the computer vision field. Section 2.3 explains recurrent neural networks and the vanishing gradient problem. In this context LSTM is introduced as a way to solve this problem.

## 2.1 Neural networks and machine learning

The first appearance of an artificial neural network is in the work of Rosenblatt [1961] where he introduced the *perceptron*. This type of network tries to imitate the basic working principles of neurons using small computational units combined together through weighted connections. The result of this idea is the mathematical model represented in Figure 2.1. The inputs $x_i$ are composed together (using multiplication) with the

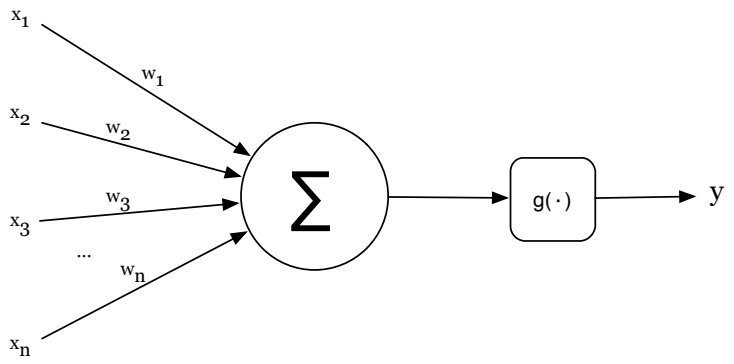



Figure 2.1. Illustration of the perceptron.

weights $w_i$ and summed together. After that, the result goes through a function $g(\cdot)$ which, in the case of the perceptron, is the *sign*[1] function. The overall result is a mathematical function, shown in Equation 2.1.

$$y = g\left(\sum_{i=1}^{n} w_i \cdot x_i\right) \tag{2.1}$$

Since the time of their first appearance, many other architectures were proposed and the success of neural networks has been continuously growing. One of the most successful and famous one is the *multilayer perceptron* (MLP). This type of network is a natural extension of the perceptron in which, instead of having a single node, there are many nodes arranged in a *feed-forward* architecture as depicted in Figure 2.2. As we can see, the inputs on the left are forwarded to the first hidden layer, which computes its activations and forwards them to the next layer and so on until the output layer. From this intuition it's easy to derive the formula in Equation 2.2, in which $h_j$ is the

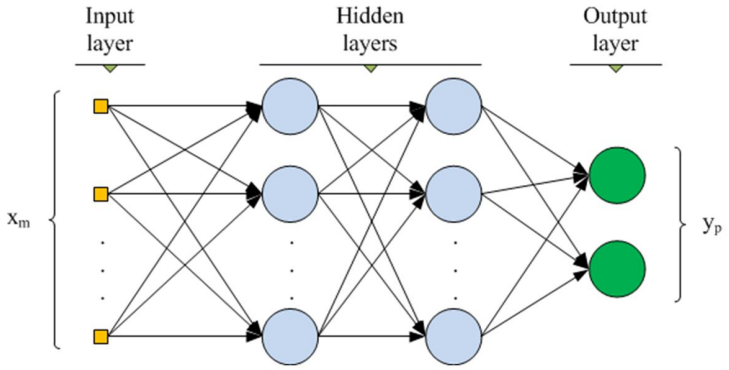



Figure 2.2. Illustration of a multilayer perceptron. Image from [Thiago M. Geronimo, 2013-01-16].

activation of unit $j$, $w_{ij}$ is the weight of the connection from unit $i$ to unit $j$ and $g(\cdot)$ is an activation function.

$$h_j = g\left(\sum_{i=1}^{n} w_{ij} \cdot h_i\right) \tag{2.2}$$

The choice of $g$ is crucial for the correct behavior of the network and it will be discussed in more detail in Section 2.1.1.
It's important to notice that the connections in this architecture go in a single direction, so that there are no loops or self connections. This is a peculiarity of feedforward net-

[1] $sign(x) = \begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$

works and is in contrast with *recurrent neural networks* which are described in Section 2.3.

The reason of the success of neural networks is probably because they work particularly well in practice and they contributed to big improvements in machine learning. In this field we can distinguish three different types of problems:

- *Supervised learning*: given a set of exemplar input-output pairs (*training set*), learn a model that is able to predict the output given a new (never seen) input.
- *Unsupervised learning*: differently from supervised learning, there are no output examples but only inputs. The goal is to find a meaningful structure in the input.
- *Reinforcement learning*: an agent (program) learns to act in an environment by perceiving and receiving rewards for his actions.

Even though neural networks have been successfully applied to all the problems above, this thesis will focus on supervised learning.

### 2.1.1 Network training

In supervised learning we can distinguish between two main tasks:

- Regression: the value to predict is continuous (e.g. a real number). To accomplish this task with the multilayer perceptron we usually set the output activation function to be linear.[2]
- Classification: the value to predict is categorical (e.g. a binary value $[0, 1]$). In this case sigmoidal activations are usually used for output layers.

Despite the specific task, non-linear units like sigmoid or hyperbolic tangent are used in the hidden layers in order to avoid the network to compute a simple linear function.

Neural networks can be applied to both types of problems and the process of teaching the network to predict the right values is called *network training*. More specifically the training process aims to minimize a given error function $E$ with respect to a set of parameters $\theta$.

One of the most commonly used error functions for the regression task is the *mean square error* which, with a single output unit is defined as:

$$E(\theta) = \frac{1}{2n} \sum_{k=1}^{n} (y_k - t_k)^2 \tag{2.3}$$

where $n$ is the number of training examples, $y_k$ is the network output (implicitly defined on the parameters $\theta$ and the input $x_k$) and $t_k$ is the corresponding target value. The

[2] A linear unit simply computes $f(x) = x$, sigmoid is $\sigma(x) = \frac{1}{1+e^{-x}}$ and hyperbolic tangent is $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

argument $\theta$ of the error function is important because it represents the weights of the network and is the set of parameters that we can tune to minimize the error.

As we said, MSE is used for regression while for classification what is usually used is the *cross entropy error function* (binomial or multinomial) which in general takes the following form:

$$E(\theta) = -\frac{1}{n}\sum_{k=1}^{n}[t_k \log y_k + (1-t_k)\log(1-y_k)] \tag{2.4}$$

Despite of this, the network principle remains the same and also the training algorithm is general and doesn't change with the error function.

Many algorithms have been proposed to perform the error minimization, among which *backpropagation* is the most commonly used one. The continuous form of the algorithm was derived in the early 60s [Kelley, 1960; Bryson and Ho, 1970] but its first application to neural networks is attributed to Werbos [1974].

The principle of backpropagation is to alternate forward and backward passes through the network in order to compute gradients with respect to both internal parameters and inputs. Those gradients are then used by a numerical optimization method called *gradient descent*, in order to update the weights of the network in a way that follows the slope of the gradient. The basic update formula for gradient descent is

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \mu \nabla E(\mathbf{W}_t) \tag{2.5}$$

where $\mu$ is called learning rate and is usually a value in the range $[0,1]$. As we can notice everything boils down to the computation of the gradient of the error function with respect to the weights. In practice this is where back propagation comes in. It is an iterative procedure with every iteration being divided into two parts:

- *Forward*: present a set of inputs from the training data to the network and compute the error between the output and the target. This gives the term $E(\mathbf{W}_t)$.
- *Backward*: compute the gradient of the error with respect to all the network parameters and apply the update rule 2.5.

It's important to notice that this is a local search method, which means it exploits local informations about the function but it can't have a global view so it can't find a global minima in the general case.

If the weight update happens after having accumulated the gradient information over all the dataset we talk about *batch* learning. If instead we make the update rule change the parameters after having seen a subset of the training examples we have the so called *stochastic gradient descent* (SGD). Even tough the batch version is more mathematically rigorous, it's often the case that SGD performs better in practice, particularly on large datasets, achieving faster convergence and sometimes finding a better minimum [Bottou, 2012]. Moreover it has been proven that computing the gradient the way SGD does it, gives us an unbiased estimate which approximate the full gradient.

When training neural networks, two opposite problems can arise: *underfitting* and *overfitting*. In the first case the model hasn't got enough expressive power to learn the dataset so the error stays high. In the second case the network learns "too much" by memorizing the training examples and lose its ability to generalize on new data. To fight the first problem usually it's sufficient to increase the complexity of the model, for example by adding more parameters (e.g. hidden units). Models suffering from underfitting are said to have high bias. The second problem is more tricky and many solution have been proposed in the literature. One of the common ways of reducing overfitting is to get more data for the training set. This solution is not always possible or sufficient, so a series of techniques which fall under the name of *regularization* were invented. A model suffering of overfitting is said to have high variance. It's important to note that there is always a bias/variance tradeoff and is important to try to find the best compromise when training a neural network.

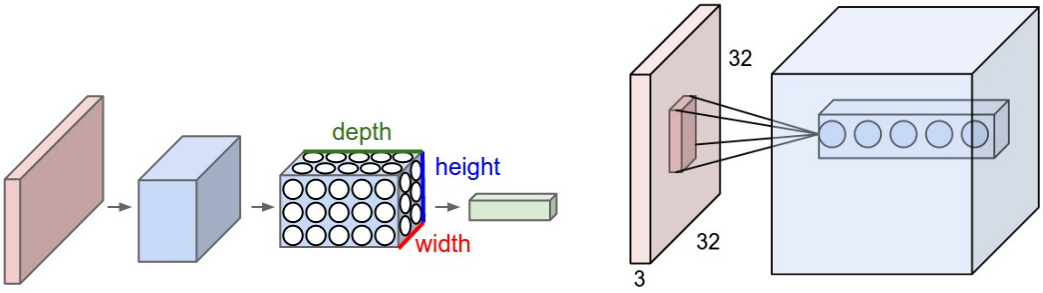
## 2.2 Convolutional neural networks

A particular category of neural network that achieved amazing results in many tasks of machine learning are *convolutional neural networks* (CNNs). They were first proposed by Fukushima [1980] and improved in the following years by many contributors, among which LeCun et al. [1989] was the first to apply backpropagation to this architecture. Thanks to an efficient GPU implementation by Ciresan et al. [2011] they finally improved upon state of the art results on many datasets and they are now among the most widely used classifiers on a vast variety of tasks.

### 2.2.1 Working principle

Convolutional neural networks are a particular instance of feedforward network created with the specific assumption that the input is an image or, more in general, a 3D volume as shown in Figure 2.3a. For example a standard RGB image of size HxW would have a volume with shape $(3, H, W)$ where the first dimension is called depth or channels. The main component of a CNN is the convolution layer. This particular layer uses a receptive field combined with a convolution operator to transform an input volume to an output volume. The depth dimension of the output volume can be seen as neurons, so that each neuron sees a local region of its input as shown in Figure 2.3b. Neurons in the same spatial dimension see the same region of input but compute their output with different weights, also called filters or kernels.
The size of the output volume is controlled by four parameters:

- Depth: as already said, this parameter controls the number of neurons (stacked along the channel dimension) which are connected to the same input region.




(a) Illustration of layers arranged in 3D volumes. (b) Illustration of receptive field.

Figure 2.3. Convolutional network architecture, represented as a sequence of volumes transformations. Images from [CS2, 2015].

- Kernel size: usually a small integer, is the size of the receptive field. For example if the receptive field is a $5 \times 5$ grid, the kernel size is 5.

- Stride: this parameter tells how much we jump along the spatial dimension of the input. For example a stride of 1 means that we move our kernel shifting it by one pixel every time. The larger the stride, the less overlapped are the receptive fields and the smaller is the output volume.

- Padding: additional zeros that we can add to the border of the input volume, usually to make it have the same height and width of the output volume.

The spatial size of the output volume can be computed as

$$O = \frac{W - F + 2P}{S + 1} \tag{2.6}$$

where $W$ is the input size (e.g. height), $F$ is the kernel size, $P$ is the padding and $S$ is the stride.

To compute the value of a neuron we use the standard convolution operator: given a kernel $K$ of size $k$ and an image $I$, the output for a given pixel $(x, y)$ is:

$$G(x,y) = \sum_{i=-s}^{s} \sum_{j=-s}^{s} I(x+i, y+j) \cdot K(i,j) \qquad with \quad s = \left\lfloor \frac{k}{2} \right\rfloor \tag{2.7}$$

Many tricks have been proposed to speedup the convolution process, many of which exploit the Fourier frequency domain transformation and efficient GPU parallelizations (see [Mathieu et al., 2013] and [Vasilache et al., 2014]).

### 2.2.2 Training tricks

Besides an efficient implementation, many "tricks" have been developed to make training of CNNs feasible, especially for deep[3] architectures with millions of trainable parameters. The following are the most commonly used ways to improve CNN's training.

#### Rectified linear units

ReLu activations are a particular kind of non-linearity which was argued to be more biologically plausible and proved to be more effective than the standard sigmoid and hyperbolic tangent functions [Glorot et al., 2011]. Its definition is simply $f(x) = max(0, x)$ which is just thresholding at zero. This type of activation has proven to be particularly effective in terms of speedup of training and overall behavior of the network. Its main advantage, in contrast to other sigmoidal functions, is the non-saturating behavior which allows them to always have a gradient greater than zero.

#### Pooling layer

It's common to put a pooling layer after a sequence of convolution-ReLu layers to downsample the volumes along the spatial dimension, reducing their size. Its principle is similar to the convolution layer, having a kernel sliding on the input volume but, instead of applying the convolution operator, it applies a function that reduces the input matrix to one single scalar. One of the most used functions for this purpose is $MAX$, hence the name *max-pooling*, but also *average* and *L2-norm* have been used.
The main purpose of this layer is to reduce the complexity of the network, resulting in faster training and improved generalization.

#### Dropout layer

Dropout is a technique introduced by Hinton et al. [2012] aimed at improving generalization and preventing overfitting. The main idea is to randomly remove some units with the corresponding input and output connections during training. For example we can have a fixed probability $p$ of keeping an unit and $1 - p$ of removing it (typically $p$ is around 0.5). This can be seen as training $2^n$ many different networks [Srivastava et al., 2014] where $n$ is the total number of units. At test time the virtual models are averaged by multiplying the weights of the network (without dropout) by $p$.

[3] With the term "deep" we refer to the field of *deep learning*, which is a branch of machine learning that tries to train complex multilayer architectures to solve traditional problems [Schmidhuber, 2015].

## 2.3 Recurrent neural networks

So far the focus has been on feed forward neural networks, where there are no loops or self connections in the network topology. This constraint is relaxed in the so called *recurrent neural networks* (RNNs). Many variant of this architecture have been proposed, among which the most simple one is the *simple recurrent network* depicted in Figure 2.4. As we can see there are the usual input, output and hidden layers but in

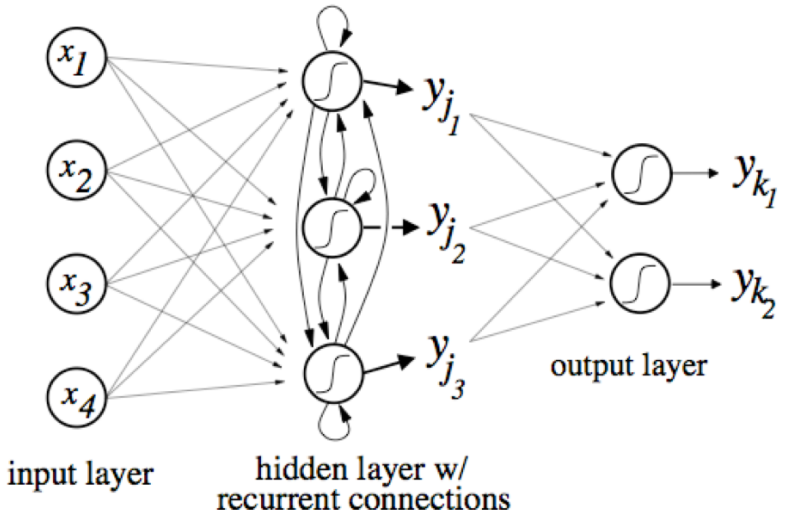



Figure 2.4. A simple recurrent neural network. Image from [Gomez and Schmidhuber, 2013].

the hidden layer we have the additional recurrent connections. This topology allows the network to have *memory* of the past inputs making it suitable for problems where past context is important like sequence labeling (e.g. handwritten text recognition and speech recognition).

### 2.3.1 Backpropagation through time

Many algorithms have been proposed to train a RNN [Jaeger, 2002] but in this thesis we will focus on *backpropagation through time* (BPTT) proposed by Werbos [1990]. This algorithm is based on the concept of unfolding the network in time, which means that we transform the recurrent structure into a more familiar feedforward architecture with as many hidden layers as the length of the input sequence. This operation allows us to use the standard forward-backward procedure like if we are training a simple MLP.

### Forward pass

The forward pass is similar to the one of a multilayer perceptron, with the exception that we need to consider activations coming from one step back in time. Given an input sequence $\mathbf{x}$ to a network with $I$ input units and $H$ hidden units the the activations for the hidden layer can be computed as

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} r_{hh'} b_{h'}^{t-1} \tag{2.8}$$

$$b_h^t = g(a_h^t) \tag{2.9}$$

where $x_i$ is the input $i$ at time $t$, $w_{ih}$ is the weight from input unit $i$ to hidden unit $h$, $r_{hh'}$ is the recurrent weight from hidden unit $h$ to hidden unit $h'$ and $b_{h'}^{t-1}$ is the activation of unit $h'$ at the previous timestep. Equation 2.8-2.9 can be also written in matrix notation, which turns out to be quite helpful for implementation:

$$\mathbf{a}^t = \mathbf{W}\mathbf{x}^t + \mathbf{R}\mathbf{b}^{t-1} \tag{2.10}$$

$$\mathbf{b}^t = g(\mathbf{a}^t) \tag{2.11}$$

In the previous equations $W \in \mathbb{R}^{H \times I}$ is the input weight matrix, $R \in \mathbb{R}^{H \times H}$ is the recurrent weight matrix while $\mathbf{a}$, $\mathbf{x}$ and $\mathbf{b}$ are all vectors of size $H$.

### Backward pass

As in standard backpropagation, we need to compute the partial derivatives of the loss with respect to the weights, knowing the derivatives of the loss with respect to the network output $\left(\delta_k^t = \frac{\partial L}{\partial b_k^t}\right)$. The deltas for the hidden units can be computed as

$$\delta_h^t = g'(a_h^t)\left(\sum_{k=1}^{K} \delta_k^t w_{kh} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} r_{h'h}\right) \tag{2.12}$$

where the meaning of all the components is the same as in the forward pass, $w_{hk}$ is the weight from hidden unit $h$ to output unit $k$ and $g'(\cdot)$ is the derivative of the $g(\cdot)$ function. This rule is applied starting from the end of the sequence ($t = T$) and going back until $t = 0$. Having the deltas for the hidden units, is then possible to compute the update for the weights:

$$\frac{\partial L}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^{t+1} b_i^t \tag{2.13}$$

As for the forward pass, both the previous computations can be vectorized:

$$\delta\mathbf{h}^t = g'(\mathbf{a}^t)\left(\mathbf{W}^T \delta\mathbf{k}^t + \mathbf{R}^T \delta\mathbf{h}^{t+1}\right) \tag{2.14}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{t=1}^{T} \langle \delta \mathbf{h}^{t+1}, \mathbf{b}^{t} \rangle \tag{2.15}$$

In the above equations $\langle \cdot, \cdot \rangle$ denotes the outer product and $\star^T$ denotes the matrix transpose.

### 2.3.2 Bidirectional RNNs

In some kind of problems the concept of time in the sequence is not constrained to follow a particular direction, allowing in principle to use the "future" of the sequence itself. Examples of such problems come out when the input is spatial (e.g. protein secondary structure prediction) or when the output of classification can be delayed to some natural break in the sequence (e.g. in speech recognition we can wait for a pause). Obviously this is not applicable to problems like stock price prediction, where we don't have access to future context.

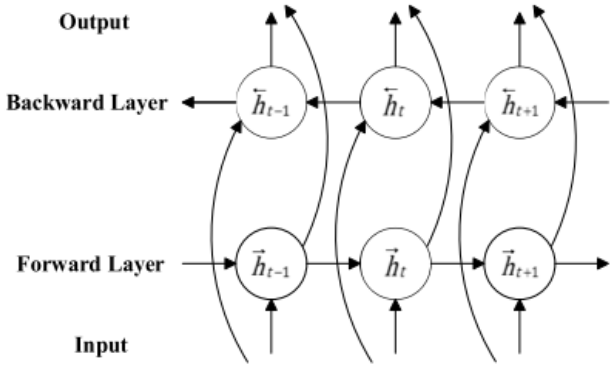



Figure 2.5. Illustration of the hidden layer of a bidirectional recurrent neural network. Image from [ima, 2015].

Standard RNNs process the input sequence in a single direction, but an extension called *bidirectional recurrent neural network* (BRNN) proposed by Schuster and Paliwal [1997] allows to process a sequence starting both from its "start" and its "end". The principle is to apply two standard RNNs independently to the sequence starting from the beginning and from the end and keeping track of the activations of both as shown in Figure 2.5. This simple solution allows the recurrent network to have access to the full context of a sequence, which generally allows a much better performance in term of accuracy.

### 2.3.3 Vanishing gradient problem

It has been proven that a simple RNN with a sufficient number of hidden neurons can approximate any differentiable trajectory [Hammer, 2000]. It has also been shown that, while a standard MLP can approximate any continuous function, a recurrent neural network can in principle learn any algorithm. Even though this is a strong theoretical result, this type of networks don't work as expected in many kind of real problems and are particularly difficult to train correctly. One of the main reasons for this is the so called *vanishing gradient problem* [Hochreiter et al., 2001] which is a behavior of RNNs in which the gradient information coming from the output layer gets smaller and smaller as it goes through the hidden layers and finally vanishes, preventing the weights from being updated correctly. This problem affects all types of networks but is particularly prominent in recurrent neural networks because their output can depend on an input seen many timesteps into the past.

Many solution have been proposed to the problem, among which the most famous one (and the one used in this thesis) is the *long short-term memory* (LSTM) by Hochreiter and Schmidhuber [1997], described in the following section.

#### Long short-term memory

The principle of LSTM is to substitute the standard recurrent unit of a simple RNN with a more complex structure composed of gates and various recurrent connections as shown in Figure 2.6. As we can see an LSTM block is composed of three gates and

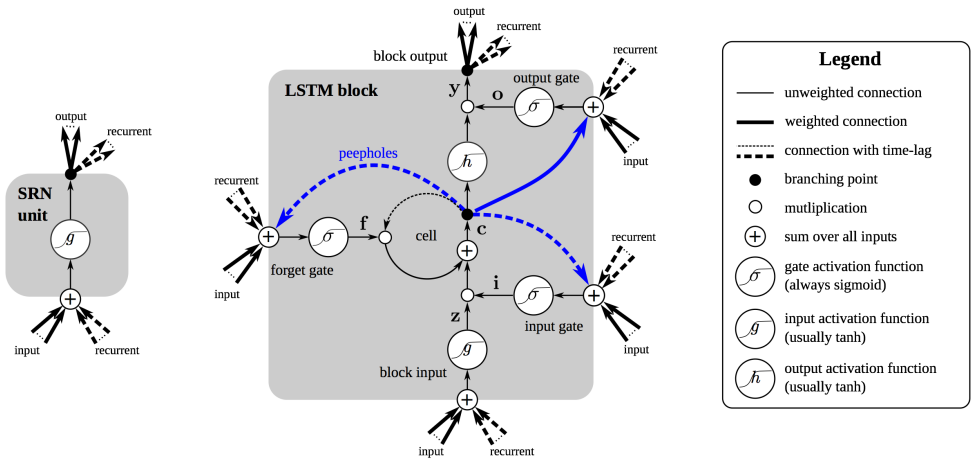



Figure 2.6. A simple recurrent unit compared to an LSTM block. Image from [Greff et al., 2015].

a cell-state in addition to the usual input and output:

- Input gate: controls the information that get into the cell. It gets connections

from the input and the previous output, sum them up and pass the result throw a sigmoid non-linearity.

- Output gate: controls what goes out from the cell. It works the same way as the input gate.

- Forget gate: allows the block to selectively forget its internal state. Again it works the same way as the other gates.

- Cell state: it's the internal state of the cell. It sums the inputs from the *input gate* multiplied with the *forget gate* with the *block input* multiplied by the previous state. Its output goes throw a sigmoidal non-linearity and gets multiplied by the output of the *output gate*.

In Figure 2.6 we can also see peepholes connections (blue arrows), that was introduced by Gers et al. [2000] as a way to allow the cell to learn precise timings. Those connections are useful in problems where the network has to "count" the timesteps elapsed from a particular event in the past. In this thesis they will be ignored since we don't deal with such type of problems. Moreover it was recently shown that they have little influence on practical problems [Greff et al., 2015] and this allows to simplify the forward and backward formulas which are shown below in vectorized form. There $\mathbf{W}_\star$ represents rectangular input weight matrices, $\mathbf{R}_\star$ represents square recurrent weight matrices, $\mathbf{b}_\star$ represents bias weights and $\odot$ is the elementwise multiplication of two vectors.

Forward pass

$$\bar{\mathbf{z}}^t = \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z \tag{2.16}$$
$$\mathbf{z}^t = g(\bar{\mathbf{z}}^t) \qquad (Block\ input) \tag{2.17}$$
$$\bar{\mathbf{i}}^t = \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{b}_i \tag{2.18}$$
$$\mathbf{i}^t = \sigma(\bar{\mathbf{i}}^t) \qquad (Input\ gate) \tag{2.19}$$
$$\bar{\mathbf{f}}^t = \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{b}_f \tag{2.20}$$
$$\mathbf{f}^t = \sigma(\bar{\mathbf{f}}^t) \qquad (Forget\ gate) \tag{2.21}$$
$$\bar{\mathbf{o}}^t = \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{b}_o \tag{2.22}$$
$$\mathbf{o}^t = \sigma(\bar{\mathbf{o}}^t) \qquad (Output\ gate) \tag{2.23}$$
$$\mathbf{c}^t = \mathbf{z}^t \odot \mathbf{t}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t \qquad (Cell\ state) \tag{2.24}$$
$$\mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t \qquad (Block\ output) \tag{2.25}$$

Backward pass

Given the deltas $\Delta^t = \frac{\partial L}{\partial \mathbf{y}^t}$ coming from the layer above, the deltas for the internal components of the LSTM can be calculated as follows:

$$\delta \mathbf{y}^t = \Delta^t + \mathbf{R}_z^T \delta \mathbf{z}^{t+1} + \mathbf{R}_i^T \delta \mathbf{i}^{t+1} + \mathbf{R}_f^T \delta \mathbf{f}^{t+1} + \mathbf{R}_o^T \delta \mathbf{o}^{t+1} \tag{2.26}$$

$$\delta \mathbf{o}^t = \delta \mathbf{y}^t \odot h(\mathbf{c}^t) \odot \sigma'(\overline{\mathbf{o}}^t) \tag{2.27}$$

$$\delta \mathbf{c}^t = \delta \mathbf{y}^t \odot \mathbf{o}^t \odot h'(\mathbf{c}^t) + \delta \mathbf{c}^{t+1} \odot \mathbf{f}^{t+1} \tag{2.28}$$

$$\delta \mathbf{f}^t = \delta \mathbf{c}^t \odot \mathbf{c}^{t-1} \odot \sigma'(\overline{\mathbf{f}}^t) \tag{2.29}$$

$$\delta \mathbf{i}^t = \delta \mathbf{c}^t \odot \mathbf{z}^t \odot \sigma'(\overline{\mathbf{i}}^t) \tag{2.30}$$

$$\delta \mathbf{z}^t = \delta \mathbf{c}^t \odot \mathbf{i}^t \odot g'(\overline{\mathbf{z}}^t) \tag{2.31}$$

The deltas for the input can be computed by summing all the contributions from the LSTM cell:

$$\delta \mathbf{x}^t = \mathbf{W}_z^T \delta \mathbf{z}^t + \mathbf{W}_i^T \delta \mathbf{i}^t + \mathbf{W}_f^T \delta \mathbf{f}^t + \mathbf{W}_o^T \delta \mathbf{o}^t \tag{2.32}$$

Finally the gradient with respect to the weights (input, recurrent and bias) can be computed using dot product the same way as shown with the simple RNN (see Equation 2.15).

# Chapter 3

# Multidimensional recurrent neural networks

This chapter focuses on the main topic of this thesis, which are *multidimensional recurrent neural networks*. In Section 3.1 we describe the most simple version of MDRNN. Section 3.2 introduces the LSTM extension of MDRNNs, the so called *multidimensional long short-term memory*. Since recently MDRNNs are receiving a lot of interest, in Section 3.3 we discuss some of the related work present in the literature, trying to point out their strengths and weaknesses.

## 3.1 MDRNN

It can happen that a particular problem is well described by using multiple dimensions. This is the case for example with images, videos and electron magnetic resonances (EMR). A proposal aimed at exploiting this structure are *multidimensional recurrent neural networks* (MDRNNs) by Graves et al. [2007] which are a particular case of DAG-RNNs [Baldi and Pollastri, 2003] and a special case of structured backpropagation [Goller and Kuchler, 1996]. This type of network extends the concept of sequence to multiple time dimensions and allows to bring the power of recurrent neural networks to a vast variety of new problems. MDRNNs have been successfully applied to image classification, image segmentation [Graves et al., 2007] and offline handwriting recognition [Graves and Schmidhuber, 2009]. Nonetheless in the first two cases they were applied to "toy" datasets which are not suitable to demonstrate the power of the model and there is a lack of experimental results with more challenging problems.

The principle of a MDRNN is to arrange the structure of the hidden units so that we can see the input like a sequence in a multidimensional space. Taking the 2-dimensional case as an example (like an image), as we can see from Figure 3.1, the units are arranged on a grid and connected such that the further away they are from the start of the sequence, the more context is available to the them. This is a desirable property be-

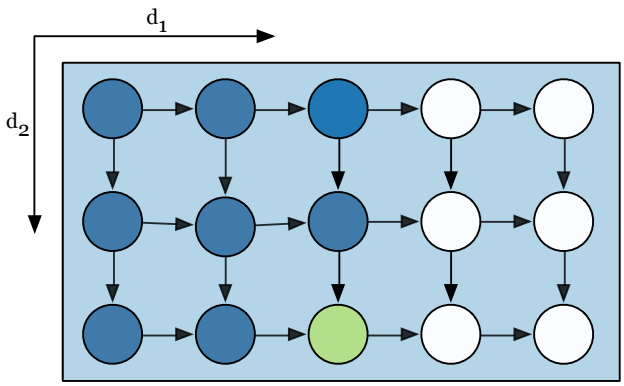



Figure 3.1. Illustration of a 2-dimensional RNN. Following the arrows, we can see that the green unit has context available from all the blue units.

cause, unlike simple or convolutional neural networks, it allows to take decisions based on pixels very far from each other. Notice that every circle in Figure 3.1 is actually a stack of units representing the number of hidden neurons of the MDRNN. Every stack is connected to the next one with a recurrent weight matrix $\mathbf{R} \in \mathbb{R}^{H \times H}$ where $H$ is the number of hidden neurons. The result is that with $d$ dimensions we have $d$ different recurrent matrices (one along each dimension).

Like RNNs can be extended to bidirectional RNNs, MDRNNs can be extended to multidirectional MDRNNs by making the input sequence start at any of the $2^D$ possible different corners which is also the number of MDRNN layers we need. For example in an image we have $D = 2$ so the number of hidden layers we need is $2^2 = 4$ which coincide with the four corners of the image.

### 3.1.1 Network training

The training of a MDRNN with $D$ dimensions can be accomplished by extending the standard BPTT algorithm for RNNs described in section 2.3. The following are the equations for the forward and backward passes:

Forward

$$\mathbf{a}^{\mathbf{p}} = \mathbf{W}\mathbf{x}^{\mathbf{p}} + \sum_{d=1}^{D} \mathbf{R}^{d}\mathbf{b}^{\mathbf{p}_{d-}} \tag{3.1}$$

$$\mathbf{b}^{\mathbf{p}} = g(\mathbf{a}^{\mathbf{p}}) \tag{3.2}$$

Backward

$$\delta \mathbf{h}^{\mathbf{p}} = g'(\mathbf{a}^{\mathbf{p}}) \left( \mathbf{W}^T \delta \mathbf{k}^{\mathbf{p}} + \sum_{d=1}^{D} \mathbf{R}^{T,d} \delta \mathbf{h}^{\mathbf{p}_{d+}} \right) \tag{3.3}$$

In the above formulas $\star^{\mathbf{p}}$ denotes a value at the current timestep $\mathbf{p}$ while $\star^{\mathbf{p}_{d-}}$ and $\star^{\mathbf{p}_{d+}}$ represent respectively the previous and the future value from the current timestep along dimension $d$. As we can notice, there is only one weight matrix $\mathbf{W}$ from input to the MDRNN layer but we have $D$ different recurrent matrices, denoted with $\mathbf{R}^d$.

## 3.2 Multidimensional long short-term memory

MDRNNs suffer from the same issues of simple RNNs including the vanishing gradient problem. For this reason in this thesis we focus on the LSTM extension of multidimensional networks (MDLSTM). The formulas for the forward and backward are very similar to the normal LSTM formulas with the difference that we have $D$ recurrent weight matrices for each gate and also $D$ different forget gates (instead of only one), each one connected to the previous *cell state* only along dimension $d$. The following are the vectorized formulas with the same notation used before:

Forward pass

$$\bar{\mathbf{z}}^{\mathbf{p}} = \mathbf{W}_z \mathbf{x}^{\mathbf{p}} + \left( \sum_{d=1}^{D} \mathbf{R}_{z,d} \mathbf{y}^{\mathbf{p}_{d-}} \right) + \mathbf{b}_z \tag{3.4}$$

$$\mathbf{z}^{\mathbf{p}} = g(\bar{\mathbf{z}}^{\mathbf{p}}) \qquad (Block\ input) \tag{3.5}$$

$$\bar{\mathbf{i}}^{\mathbf{p}} = \mathbf{W}_i \mathbf{x}^{\mathbf{p}} + \left( \sum_{d=1}^{D} \mathbf{R}_{i,d} \mathbf{y}^{\mathbf{p}_{d-}} \right) + \mathbf{b}_i \tag{3.6}$$

$$\mathbf{i}^{\mathbf{p}} = \sigma(\bar{\mathbf{i}}^{\mathbf{p}}) \qquad (Input\ gate) \tag{3.7}$$

$$\bar{\mathbf{f}}_n^{\mathbf{p}} = \mathbf{W}_{f_n} \mathbf{x}^{\mathbf{p}} + \left( \sum_{d=1}^{D} \mathbf{R}_{f_n,d} \mathbf{y}^{\mathbf{p}_{d-}} \right) + \mathbf{b}_{f_n} \tag{3.8}$$

$$\mathbf{f}_n^{\mathbf{p}} = \sigma(\bar{\mathbf{f}}_n^{\mathbf{p}}) \qquad (Forget\ gates) \tag{3.9}$$

$$\bar{\mathbf{o}}^{\mathbf{p}} = \mathbf{W}_o \mathbf{x}^{\mathbf{p}} + \left( \sum_{d=1}^{D} \mathbf{R}_{o,d} \mathbf{y}^{\mathbf{p}_{d-}} \right) + \mathbf{b}_o \tag{3.10}$$

$$\mathbf{o}^{\mathbf{p}} = \sigma(\bar{\mathbf{o}}^{\mathbf{p}}) \qquad (Output\ gate) \tag{3.11}$$

$$\mathbf{c}^{\mathbf{p}} = \mathbf{z}^{\mathbf{p}} \odot \mathbf{i}^{\mathbf{p}} + \sum_{d=1}^{D} \mathbf{c}^{\mathbf{p}_{d-}} \odot \mathbf{f}_d^{\mathbf{p}} \qquad (Cell\ state) \tag{3.12}$$

$$\mathbf{y}^{\mathbf{p}} = h(\mathbf{c}^{\mathbf{p}}) \odot \mathbf{o}^{\mathbf{p}} \qquad (Block\ output) \tag{3.13}$$

Note that writing $\mathbf{f}_n^{\mathbf{p}}$ we denote the forget gate at timestep $\mathbf{p}$ along the $d^{th}$ of the $D$ dimensions.

Backward pass

$$\delta\mathbf{y}^{\mathbf{p}} = \Delta^{\mathbf{p}} + \sum_{d=1}^{D} \mathbf{R}_{z,d}^{T} \delta\mathbf{z}^{\mathbf{p}_{d+}} + \sum_{d=1}^{D} \mathbf{R}_{i,d}^{T} \delta\mathbf{i}^{\mathbf{p}_{d+}} + \sum_{d=1}^{D} \mathbf{R}_{o,d}^{T} \delta\mathbf{o}^{\mathbf{p}_{d+}} + \sum_{n=1}^{D}\sum_{d=1}^{D} \mathbf{R}_{f_n,d}^{T} \delta\mathbf{f}_n^{\mathbf{p}_{d+}} \tag{3.14}$$

$$\delta\mathbf{o}^{\mathbf{p}} = \delta\mathbf{y}^{\mathbf{p}} \odot h(\mathbf{c}^{\mathbf{p}}) \odot \sigma'(\overline{\mathbf{o}}^{\mathbf{p}}) \tag{3.15}$$

$$\delta\mathbf{c}^{\mathbf{p}} = \delta\mathbf{y}^{\mathbf{p}} \odot \mathbf{o}^{\mathbf{p}} \odot h'(\mathbf{c}^{\mathbf{p}}) + \sum_{d=1}^{D} \delta\mathbf{c}^{\mathbf{p}_{d+}} \odot \mathbf{f}_d^{\mathbf{p}_{d+}} \tag{3.16}$$

$$\delta\mathbf{f}_n^{\mathbf{p}} = \sum_{d=1}^{D} \delta\mathbf{c}^{\mathbf{p}} \odot \mathbf{c}^{\mathbf{p}_{d-}} \odot \sigma'(\overline{\mathbf{f}}_n^{\mathbf{p}}) \tag{3.17}$$

$$\delta\mathbf{i}^{\mathbf{p}} = \delta\mathbf{c}^{\mathbf{p}} \odot \mathbf{z}^{\mathbf{p}} \odot \sigma'(\overline{\mathbf{i}}^{\mathbf{p}}) \tag{3.18}$$

$$\delta\mathbf{z}^{\mathbf{p}} = \delta\mathbf{c}^{\mathbf{p}} \odot \mathbf{i}^{\mathbf{p}} \odot g'(\overline{\mathbf{z}}^{\mathbf{p}}) \tag{3.19}$$

### 3.2.1 Complexity

Since the forward and backward passes require one pass each through the data sequence, the overall complexity of MDRNN training is linear in the number of data points and the number of network weights [Graves et al., 2007]. Notice that the number of weights is quadratic in the number of hidden units, so that if the number of hidden units is $n$, the number of weights is $O(n^2)$, making it difficult to scale to an high number of hidden units.

In the case of the multidirectional variant, as we already said, the number of possible directions is $O(2^d)$, which multiplies the previous complexity for a single direction. This exponential scaling is attenuated by the fact that the overall power of the network is dominated by the total number of parameters and so it's possible to use layers with a small number of units without loosing the model power [Graves et al., 2007].

## 3.3 Related work

One of the ways proposed to deal with multidimensional data is represented by *multidimensional hidden Markov models* (MHMM). This approach has the advantage of being a natural extension of HMMs, which have a strong theory behind and well studied properties with known algorithms to solve the most common problems. The drawback is that the computational complexity of the Viterbi algorithm used for inference is exponential

in the number of data samples and the memory required is exponential in the number of dimensions. Even though many approximate solutions have been proposed, none of them is capable of fully exploiting the power of the model. Moreover HMMs build upon the Markov assumption, which make very difficult for them to take advantage of long-term context.

Another approach by Donahue et al. [2014] combines CNNs with LSTM cells to tackle problems like video classification, image captioning and video description. In this work LSTM are used to model the 1-dimensional sequential component of the problem, while CNNs (or MRF) where used for handling images (or frames). The main point is that the LSTM cells used in this architecture are 1-dimensional and applied to monodimensional sequences. The authors were able to integrate the two components into a single architecture that has the advantage of being trainable end-to-end.

A recent approach aimed to extend RNNs to image data is ReNet by Visin et al. [2015] which uses standard 1-dimensional RNNs to perform classification by presenting the image (divided into flattened patches) to many different RNNs that process it in different directions. This approach has the advantage of allowing to use the more simple 1-dimensional RNN architecture and, compared to MDRNNs, allows to use a higher number of hidden units due to its lower computational complexity and higher parallelizability.

Finally Kalchbrenner et al. [2015] propose an architecture very similar to the MDLSTM but with a slightly different way of arranging internal connections. This modification should have the advantage of improving the stability of the network, which is discussed in Section 5.2.1.

# Chapter 4

# Image segmentation

In this chapter we talk about the image segmentation problem and how it is addressed with neural networks. In Section 4.1 we talk about the general problem of image segmentation. In Section 4.2 we talk about the most used classical approach to semantic image segmentation, which involves the use of conditional random fields. Section 4.3 deals with the application of neural networks to image segmentation and proposes a new approach to solve the problem involving a combination of MDRNNs and CNNs.

## 4.1 Image segmentation problem

An image segmentation is the partition of an image into a set of non-overlapping regions whose union is the entire image. The purpose of segmentation is to decompose the image into parts that are meaningful with respect to a particular application [Haralick and Shapiro, 1992]. As can be seen from the definition, the problem heavily depends on the particular application and was tackled in many different ways in the literature.

First of all we should distinguish between three types of problems, which distinguish on what is known about the image and how much the user interact during the process:

- Unsupervised: the segmentation is carried out with no prior knowledge about the image and without the intervention of an operator.
- Semi-supervised: is similar to unsupervised, with the difference that an operator gives some initial hints on the correct segmentation.
- Semantic: the segmentation is performed based on prior knowledge given by a set of pre-segmented (usually by humans) images.

The majority of classical methods belongs to the first two categories. Those algorithms try to perform a meaningful segmentation by considering some local features of the image. For example a simple thresholding algorithm uses the color of pixels to segment an image into two parts (e.g. foreground and background). More complex methods like

*Mumford-Shah* [Chambolle, 1995] aim to minimize some potential functional defined over the image. Other methods like *GraphCut* model the problem as a *Markov random field* and then minimize an energy function defined over it.

For the third category the most used method involves the use of *conditional random fields* (CRFs) as we will see in Section 4.2.

In the following sections we are going to focus on the third category, which is the target of this thesis.

## 4.2 CRFs for image segmentation

As mentioned in the previous section, the semantic segmentation problem has been usually approached in the literature with the help of *conditional random fields*. A CRF is a type of discriminative undirected probabilistic graphical model which encodes the conditional distribution $P(\mathbf{Y}|\mathbf{X})$, where $\mathbf{Y}$ is a set of target variables and $\mathbf{X}$ is a set of observed variables [Koller and Friedman, 2009]. In the context of semantic image segmentation, $\mathbf{Y}$ represent the labels so that $Y_i$ is the label associated to pixel $i$ in an image $I$. It turns out that this conditional probability can be modeled with a Gibbs distribution so that

$$P(\mathbf{Y} = \mathbf{y}|\mathbf{I}) = \frac{1}{Z(\mathbf{I})} e^{-E(\mathbf{y}|\mathbf{I})} \tag{4.1}$$

In the previous formula, $E(\mathbf{y}|\mathbf{I})$ is an energy function also dependent on the structure of the CRF. In the case of fully connected pairwise CRF of Krähenbühl and Koltun [2012], the energy function is a combination of a unary component $\psi_u(y_i)$ that measures the cost of assigning label $y_i$ to pixel $i$ and a pairwise energy component $\psi_p(y_i, y_j)$ which quantify the cost of assigning both label $y_i$ to pixel $i$ and label $y_j$ to pixel $j$.

It's important to note that usually a CRF method is applied as post-processing after an independent classifier has generated an initial prediction. For example some works use manual feature extracted from the images while other (like Farabet et al. [2013]) use a CNN as an automatic feature extractor. Feature extraction is a crucial component of those approaches and most of the time is not applied directly on the image pixels but rather on *superpixels* extracted from the image, which are contiguous cluster of pixels characterized by a strong continuity (within some criterion that can be task dependent). The advantage of working on superpixels instead of pixels is that the computational complexity is heavily reduced, allowing most CRF based methods to be applicable. This approach has the clear disadvantage that if the initial superpixel division is wrong, no matter how good the segmentation method is, the final result will be wrong.

In a recent approach, Zheng et al. [2015] were able to train a CRF based classifier end-to-end by transforming the CRF to an equivalent RNN and combining it with CNNs. This approach achieved state of the art results on image segmentation but due to its nature it's more similar to the approaches that will be described in Section 4.3.

## 4.3 Neural networks for image segmentation

The problem of semantic image segmentation can be modeled in the context of machine learning. In fact we can see it as a particular kind of classification task where, given a set of images and their corresponding ground truth segmentations, we want to train a classifier able to map each pixel of an image into one of $L$ categories.

Given this formulation, it's possible to apply any classifier from machine learning to image segmentation, including neural networks. Despite this, simple NNs like the multilayer perceptron are not very effective for this task. Lets say we want to segment each pixel of a grayscale image of size $128 \times 128$ into 10 possible classes; we would need a MLP with 16384 input units and an output layer of 163840 units. Adding a few hidden layers of reasonable size, this network becomes quite big and difficult to train. Moreover the MLP doesn't offer any advantage to exploit the 2D structure of the problem, since the layers of a MLP are designed to see the input as a 1-dimensional vector. What is usually done to reduce the complexity is to apply the network to a set of features extracted from the image instead of the image itself. This can improve performance but requires to manually chose the image features, which is often a non-trivial task.

This issues with MLP brought researchers to explore other architectures to solve this problem. One of the first successful proposals was the *Pulse-Coupled Neural Network* (PCNN) by Eckhorn et al. [1989] which is a recurrent neural network inspired by the visual cortex of cats. Other, more recent proposals are based on *convolutional neural networks* like [Ciresan et al., 2012; Alvarez et al., 2012; Long et al., 2014]. The advantage of using CNNs is that, unlike the classical MLP, they naturally exploit the 2D structure of images using many layers with receptive fields of different sizes. This allows to extract important relationships between neighboring pixels which is crucial in a task like image segmentation.

Another recent approach involves the use of a combination of RNNs and CNNs called *recurrent convolutional neural network* (R-CNN) [Pinheiro and Collobert, 2013]. This approach has the advantage of being able to exploit large input context using less parameters compared to a feed forward architecture.

### 4.3.1 CNN-MDRNN hybrid for image segmentation

In this thesis we propose a novel method to combine the power of CNNs with the flexibility of MDRNNs. The idea is simply to feed a raw input image to one ore more convolution layers (with optional pooling, ReLu and dropout layers) and then pass the output of those layers to a set of MDRNN layers (usually an MDLSTM for each direction). The output of the MDRNNs is then concatenated and upsampled to match the size of the input image. In Figure 4.1 we can see the general structure of a CNN-MDRNN hybrid. More in details, the raw image of size $H \times W \times C$ goes to the *Conv1* block which represent any combination of convolutional, pooling and non-linearity layers.

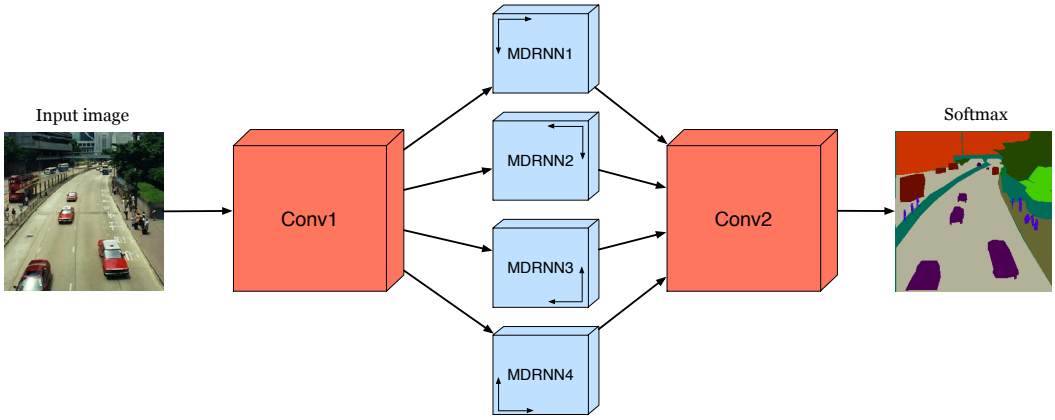



Figure 4.1. Abstract representation of a basic CNN-MDRNN.

Usually it is just a convolution layer without any addition.
The output of *Conv1* is sent to four MDRNN layers (one for each direction). In this thesis we will use MDLSTM layers which are in general better than simple MDRNNs with simple units.

The output of the MDLSTM layers is then concatenated along the depth dimension and forwarded to the *Conv2* block. This block is important because it's responsible to make the output volume match the spatial dimensions of the input image and also have the correct depth dimension to match the number of classes. If the spatial dimensions were shrank during the previous steps, following the idea described by Long et al. [2014], a so called *deconvolution* layer is used to upsample them to the correct size. This layer works by inverting the forward and backward steps of the convolution layer. The advantage of using it is that it can learn any linear or non-linear upsampling scheme, which is clearly more powerful than using a fixed upscaling strategy like bilinear interpolation in the post-processing phase.
Finally a softmax layer over the depth dimension is applied to get the pixel-wise class probabilities which directly map to the final segmentation.

This type of architecture offers many advantages which can be summarized in the following points:

- Since the input goes through convolution layers, the raw image can be used instead of handcrafted features. This is because CNNs are good to automatically extract general image features and features useful to the specific problem.

- Having convolution layers before the MDRNN blocks allows to use the properties of convolution to downscale the input image to a reasonable size. This virtually allows to handle images of any size. Moreover we can use this property to downscale the input to a reasonable size before the MDRNN blocks, which are the more time-consuming layers.

- The MDLSTM layers allow the network to have full context available for the classification of a single pixel. This is an advantage over standard CNNs which are able to see only a local context for any given pixel. Moreover the result of the four MDRNN blocks can in principle be computed in parallel, allowing a substantial speedup.
- The network is trained end-to-end, without any preprocessing or postprocessing required. This is useful because it allows the user to train its own network with low effort and doesn't require to invent particular procedures to adapt the network to this particular problem.

# Chapter 5

# Implementation details

This chapter will discuss the implementation details of the MDLSTM layer used in this thesis. More in details, Section 5.1 discusses the choice of the library in which the layer is implemented. Section 5.2 addresses the implementation of the MDLSTM layer itself and the corresponding design decisions. Finally Section 5.3 analyzes the performance of the layer and proposes a CPU parallelization, comparing it to the serial version.

## 5.1 Implementation environment

One of the purposes of this thesis is to provide a good implementation of the MDLSTM described in Section 3.2. One of the possibilities was to develop a stand alone MDRNN implementation from scratch similarly on what was done by Graves et al. [2007]. This approach has the advantage of allowing full control over the internal behavior of the network and doesn't impose any constraints on the architecture or the choice of programming language. On the other end, there are important disadvantages against such a solution: having to develop everything from scratch, a lot of components have to be written in addition to the MDLSTM layer, like the input layer handling the data and the output layer to compute the pixel-wise loss. This process is time consuming and increases the risk of introducing bugs in the code, possibly compromising the stability of the whole software. Moreover, to implement the CNN-MDLSTM hybrid proposed in Section 4.3.1, many non-trivial additional layers are needed like convolution and deconvolution. Having to implement everything would have been an huge amount of work and certainly error prone.

Given the disadvantages of a custom implementation, we opted for a realization inside an existing library. There are many possibilities in the machine learning community, among which some of the most famous are:

- **Torch7** [Collobert et al., 2011] based on the scripting language LuaJIT and an underling CUDA/C implementation.

- **Theano** [Bergstra et al., 2010] based on Python with a C core.
- **Caffe** [Jia et al., 2014] written in C++ with Python and MATLAB wrappers.

All the above are open source projects under active development and are suitable for this thesis but the choice fell on Caffe. The reason resides in the versatility to build the network architecture and the availability of many good layers like convolution, deconvolution, pooling and non-linearities. In addition the design of Caffe (discussed in Section 5.1.1) allows to focus on the specific layer we want to develop, without worrying about other parts of the network.

### 5.1.1 Caffe architecture

Caffe is an open source deep learning library maintained by the "Berkeley Vision and Learning Center". It is designed to be fast, easy to use/deploy and easy to extend. These goals are achieved through a good modular design and the use of configuration files.

Modularity of Caffe is achieved through three main components which are described in the following sections.

#### Blob

A blob is the memory container of everything that needs to move inside the library. It is essentially a 4-dimensional array used to store everything from input data to layer parameters. For example, in the case of input data its dimensions are $NxCxHxW$ which essentially represents a sequence of $N$ volumes of size $CxHxW$ (channels, width and height). Blobs are stored in contiguous regions of memory (like normal arrays); this is an important characteristic influencing some of the implementation decision described in Section 5.2.
Blobs also keep themselves synchronized across CPU and GPU using particular functions to access their content which trigger the synchronization if needed. In particular each blob has two kind of content: *data* and *diff*. The first one is used to store data generated during the forward pass while the second is used to store the deltas computed during the backward and needed to update the parameters. The functions to access those contents are {cpu,gpu}_data(), mutable_{cpu,gpu}_data(), {cpu,gpu}_diff() and mutable_{cpu,gpu}_diff(). The *mutable* functions are used to access the blob's content and modify it while the others are read-only functions.

#### Layer

Layers are the building blocks of a network and the most important part of Caffe. They are organized in a DAG structure, usually visualized with data at the bottom and layers

stacked on top of each other. Each layer has to follow a particular interface which defines three methods:

- setup: this method is executed once at network initialization and is responsible to do one time operations like dimension and parameters checks.
- forward_{cpu,gpu}: these methods perform the forward step of the layer, taking the input from the bottom layer and computing the output to pass to the top layer.
- backward_{cpu,gpu}: these methods perform the backward pass taking the gradient passed from the above layer and computing the gradient with respect to the internal weights and the input. The first one is stored inside the layer into the *diff* part of blobs while the latter is propagated down to the bottom layer.

### Solver

Once the network is designed, the solver is responsible to carry out the forward and backward passes of each layer, get the gradients and update the weights accordingly. There are currently three solvers available in Caffe:

- Stochastic gradient descent: the standard update rule described in Section 2.1, using the gradient along with a momentum term.
- Nesterov's accelerated gradient: it implements the method described in Nesterov [1983]. It's similar to the previous one but it takes the gradient on weights with added momentum.
- AdaGrad [Duchi et al., 2011]: it uses the whole history of gradients to compute the current one.

The solver is also responsible for keeping track of the training history, taking periodical snapshots of the network and perform network testing.

Besides the above components, another important part of Caffe are the configuration files. Both the solver parameters and the network architecture can be fully specified using configuration files written with the Google *protobuf* format. This allows to easily specify the network structure, enabling the user to change every parameter without being aware of the internal implementation of Caffe.

Finally another important part of this library is represented by *tests*. Every new layer or solver developed for Caffe has to provide a good set of unit tests covering its functionalities. This enforcement allows contributions from the community to be merged with the main project with less effort.

## 5.2 MDLSTM implementation and design decisions

In the context of this thesis a MDLSTM layer was developed for Caffe. The process involved the realization of an MDLSTM layer, the corresponding tests and some little changes in other components that will be described later.

### 5.2.1 Implementation

The first decision made for the development was to focus on the LSTM extension of MDRNNs (MDLSTMs). This is mainly due to the fact that LSTM works better in most of practical cases and it allows to "ignore" the vanishing gradient problem, which can easily come out in long sequences involved with image segmentation.

The second decision was to constrain the MDLSTM to be 2-dimensional. This is encouraged by the strong focus of Caffe towards images, which are intrinsically bidimensional. Implementing a generic MDLSTM would have led to a much more convoluted code with probably the need to change the data (input) layers of Caffe to handle objects more complex than images. Moreover the blob container is fixed to be 4-dimensional which is the number of dimensions needed to handle images. Since this thesis focuses on the problem of image segmentation there was no need for this extra effort.

Finally, since Caffe encourages every layer to be as independent as possible, we decided to not insert the input layer of MDLSTM network into the implementation. Instead we let it to other layers of Caffe (e.g. convolution) which are able to output "arbitrary" volumes. Using a convolution layer with $stride = 1$ and $kernel\_size = 1$ it's equivalent to have a standard fully connected layer as input of the MDLSTM. The drawback of this approach is that we force the MDLSTM layer to expect as input a volume of size $H_1 \times W_1 \times 5C_1$; the depth must be a multiple of 5 because the 2-dimensional LSTM cell has 5 gates (input gate, block input, output gate and two forget gates) and every gates wants its own input volume.

Whenever possible the code was written in vectorized form using various BLAS routines. As stated previously, a new layer in Caffe needs to define three methods (setup, forward and backward); the pseudocode for those methods is shown in the following listings.

```
void LayerSetUp(const vector<Blob<Dtype>*>& bottom, const
 ↪  vector<Blob<Dtype>*>& top) {
        // Load parameters (number of hidden units and direction)
        // Initialize weight blobs
}
```

As mentioned earlier, the `setup` method is run once at layer initialization and is responsible for getting the configuration parameters and preparing some of the internal variables needed by the layer.

```
void Forward_cpu(const vector<Blob<Dtype>*>& bottom, const
 ↪  vector<Blob<Dtype>*>& top) {
  // Loop on the bottom layers
  for (int bottom_id = 0; bottom_id < bottom.size(); ++bottom_id) {
    // Loop on mini-batch
    for (int i = 0; i < batch_size; ++i) {
      for (int x : first_dimension) {
        for (int y : second_dimension) {
          // Compute and store the activations of all gates taking
           ↪  input from *bottom
          // Put in *top the result of the forward
        }
      }
    }
  }
}
```

The `forward` method of the previous code snippet takes the `bottom` blobs as input and computes the activations for all the gates of MDLSTM as described in Equations 3.4, putting the output in the `top` blob.

```
void Backward_cpu(const vector<Blob<Dtype>*>& top,
  const vector<bool>& propagate_down, const vector<Blob<Dtype>*>&
   ↪  bottom) {
  for (int bottom_id = 0; bottom_id < bottom.size(); ++bottom_id) {
    for (int i = 0; i < batch_size; ++i) {
      for (int x : reversed_first_dimension) {
        for (int y : reversed_second_dimension) {
          // Compute and store deltas
        }
      }
      // Propagate gradients with respect to input down to *bottom
    }
    // Reset gradients of internal weights to 0

    // Compute the update for internal weights
    for (int i = 0; i < batch_size; ++i) {
      for (int x : first_dimension) {
        for (int y : second_dimension) {
          // Compute and store gradient of weights
        }
```

```
      }
    }
  }
}
```

The last method is the `backward` described in Equations 3.14. It first computes the deltas for all the internal parts of the MDLSTM cell and then uses those deltas to compute the gradients with respect to the weights.

There are three main types of operations involved in the layer's code: matrix-vector multiplication, matrix-vector addition and element-wise operations.
The first two are performed using the *gemm* and *axpby* routines of BLAS. The third is performed with a simple **for** loop.

```
const int ld_step = bottom[bottom_id]->width() *
 ↪  bottom[bottom_id]->height();

strided_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num_, 1, num_,
 ↪  abs(alpha_x), Whzx, num_, bh + bh_[bottom_id]->offset(i, 0, x -
 ↪  alpha_x, y), ld_step, (Dtype) 1., bz + bz_[bottom_id]->offset(i,
 ↪  0, x, y), ld_step);

strided_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num_, 1, num_,
 ↪  abs(alpha_y), Whzy, num_, bh + bh_[bottom_id]->offset(i, 0, x, y
 ↪  - alpha_y), ld_step, (Dtype) 1., bz + bz_[bottom_id]->offset(i,
 ↪  0, x, y), ld_step);

strided_cpu_axpby<Dtype>(num_, (Dtype) 1., bottom_data +
 ↪  bottom[bottom_id]->offset(i, 0, x, y), ld_step, (Dtype) 1., bz +
 ↪  bz_[bottom_id]->offset(i, 0, x, y), ld_step);

int start = bz_[bottom_id]->offset(i, 0, x, y);
int end = bz_[bottom_id]->offset(i, num_, x, y);
for (int e = start; e < end; e += ld_step) {
  bz[e] = tanh(bz[e]);
}
```

The above listing shows the computation of the $\overline{\mathbf{z}}^{\mathbf{p}} = \mathbf{W}_z\mathbf{x}^{\mathbf{p}} + \left(\sum_{d=1}^{D} \mathbf{R}_{z,d}\mathbf{y}^{\mathbf{p}_{d-}}\right)$ in the forward step as defined in Equation 3.4. More specifically, since the input to the MDLSMT layer is handled by a separate convolution layer, $\mathbf{W}_z$ is the identity matrix. Moreover $\mathbf{x}^{\mathbf{p}}$ is `bottom_data`, $\mathbf{R}_{z,d}$ are `Whzx` and `Whzy`, $\mathbf{y}^{\mathbf{p}_{d-}}$ is `bh` and finally $\overline{\mathbf{z}}^{\mathbf{p}}$ is `bz`.

Given the way the code was vectorized, at a given timestep $(x, y)$ we need to be able to access all the values of the corresponding hidden units both at the current timestep

and in the past: $(x-1, y)$ and $(x, y-1)$. Due to the memory layout of blobs, those values are not stored in contiguous memory regions. To avoid explicit looping over the blobs, the *stride* parameter of BLAS routines was used to access the right memory regions. Even if this approach worked in practice, it is also limiting since it doesn't allow to access any subregion of a given blob. This is the reason why it wasn't possible to avoid the explicit looping over the images of a single batch.

The code for the backward pass was implemented analogously to the forward, always trying to vectorize as much as possible and using a very similar scheme.

#### State explosion problem

After the results obtained on MNIST (see Section 5.2.2) we moved towards a different dataset composed of larger images. Training on such dataset caused the loss to explode towards infinity in a few epochs. After some investigation we found that the source of the problem was the explosion of the values of the cell state computed with Equation 3.12, which contains an unconstrained summation that causes the result to suddenly grow at values too big for a `float` variable. The problem was solved by switching to `double` precision variables which allows to handle much bigger numbers.

Even if this modification allowed to avoid the crashing of the software, it does't solve the underling problem which causes the state to grow to uselessly high values. A possible solution would be to use the modification proposed by Kalchbrenner et al. [2015], which claims an alternative architecture similar to MDLSTM that doesn't suffer from instability problems.

This architecture deserves investigation but it's also worth to explore other possible solutions.

### 5.2.2 Testing

As we said previously, every new layer in Caffe ships with its own set of unit tests. Since the main work of a layer is carried on in the forward and backward methods, those are the parts that need to be tested.

For what concerns the forward part, it was tested against a little manually computed example with the help of a simple Python prototype. This is not ideal but it's the only way to test the forward step alone.

The backward step (together with the forward) was tested using a numerical check of the gradient. Caffe allows to compute the numerical gradient of the network and check if it's close to the computed gradient within a certain threshold.

There are a total of four different tests for the backward (one for each direction) and they are ran with a fixed size input and weight matrices initialized with random numbers drawn from a normal distribution. This makes the tests not reproducible but allows to explore more configurations compared to a fixed initialization.

When all the above tests passed, a simple problem was used to verify the effective learning of the MDLSTM layer. In particular a simple MDLSTM network was trained on the MNIST database [LeCun and Cortes, 2010].

The network was composed of a first convolution layer with stride and kernel size of 1 (equivalent to a fully connected layer). After it we used 4 MDLSTM layers (one for each direction) with 25 hidden units each followed by a final fully connected layer with 10 outputs.

The convolution and fully connected layers were initialized with the xavier method Glorot and Bengio [2010]. The MDLSTM layers were initialized using a zero-centered gaussian with 0.1 standard deviation.

This network achieved 98.5% accuracy on the validation set, which is a good result given the low effort in trying to optimize the parameters and architecture of the network.

## 5.3 Scaling up performance

In this section we evaluate the performance of the MDLSTM layer and we propose a CPU parallelization method which achieves a good speedup compared to the single core version. For all the experiments we used the SIFT FLOW dataset (described in section 6.1) of RGB images of size $256 \times 256$. Since images of such size are quite big to handle, a convolution layer with kernel and stride of 2 is used to downsample the images to $128 \times 128$ which is a more reasonable size. Despite of this, all the measurements consider only the MDLSTM layers and keep out everything else. Every experiment was ran for 10 iterations; this means that if our mini-batch size is $n$, the number of images seen by one experiment is $10n$.
All the measurements have been performed on a 64 core machine with 128GB of RAM.

### 5.3.1 Single core performance

The performance of the single core version is evaluated with a simple architecture with four MDSLTM layers (one for each direction). Each of them was timed independently and the results are then combined.

#### Influence of direction

The performance of the single core version was first evaluated independently for the four directions of the MDLSTM layer. This was because it is possible in theory to have differences among them due to different memory access patterns. As we can see in Figure 5.1, the total time for each square is approximately constant, meaning that there are no significant differences in the performance of layers with different directions. The plot also shows a low variance in the results, which is a bit higher for longer computation times.


Figure 5.1. Influence of network direction on the performance of the network.

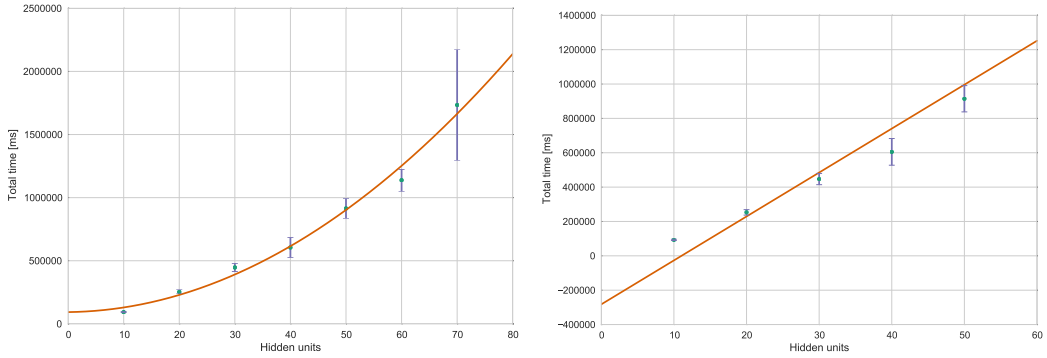
Since the differences in performance are negligible, from now on there will be no distinction between the different directions, considering them all together as a single layer.

Figure 5.1 contains an anticipation of the result with multiple threads, which will be discussed in Section 5.3.2.

### Influence of hidden units

The performance on single core is mainly affected by the size of the input and the number of hidden units. As we already said, the first have been set to $128 \times 128$ and we varied the latter with the values $[10, 20, 30, 40, 50, 60, 70]$. The mini-batch size was fixed to 32. In Figure 5.2a we can see the result of the experiments. As expected the correlation between number of hidden units and execution time is well represented by a quadratic polynomial, which gives an R-squared of 0.988. Despite of this, as shown in Figure 5.2b, a linear fit considering only hidden units less than 50 is pretty good with an R-squared coefficient of 0.982. This is expected because the number of parameter grows quadratically with the number of hidden units and the complexity is linear in the number of parameters.

Those results show that with typical values of hidden units the layer performs pretty well. The overall behavior is quadratic but close to linear in the range of $10-50$ hidden units and since it's uncommon to see more hidden units, it's fair to say that the layer implementation is good.




(a) Quadratic fit of the data. (b) Linear fit of the data.

Figure 5.2. Correlation between the number of hidden units and the performance of MDLSTM layer.

### 5.3.2 CPU parallelization

Even if the single threaded version of the MDLSTM layer is good, it was still too slow compared to other layers of Caffe like convolution. For this reason we explored a parallelization strategy to increase the performance in order to train bigger networks.

There are many ways to parallelize a neural network. The most common approach recently used is GPU parallelization, in particular the CUDA technology offered by Nvidia allowed to achieve incredible speedups on convolutional architectures in the last few years. Even if this approach can offer an important speedup thanks to the big amount of computational units (1000-4900), it is not easily applicable to the MDLSTM layer. This is because such layer has an intrinsic serial component given by the "time" nature of its approach, which limits the achievable parallelism.

A more traditional approach is to exploit multicore CPUs. A recent multicore CPU offers much less computational units (4-64) compared to a recent GPU but it has the advantage that the single units are much more powerful. As a result it's simpler to achieve a good parallelization on CPU compared to GPU.

Among the various candidate parallelization points, we decided to exploit mini-batch parallelization. This means that multiple threads are spawned to compute each image of the current mini-batch independently and then combine the results to get the same output as of the single threaded version. This strategy doesn't decrease the computation time needed by one single image but instead improves the overall throughput of the layer, allowing to process more images in the same period of time.

The desired result was obtained using OpenMP [Dagum and Enon, 1998], which is a C++/Fortran API to develop parallel applications. The main advantage of OpenMP is that it allows to parallelize the code through the use of compiler directives instead of directly modifying the source. In particular the ability of OpenMP to parallelize for loops came really useful for our purpose.

The chosen approach involved the parallelization of the outer loop of both the forward and backward passes. In particular, for the forward pass we parallelized the second loop (over the mini-batches); since there is no dependency between different images the result is that no more synchronization is needed inside the method.

In the backward pass the first part (since it doesn't have any dependencies or conflicts) was parallelized in the same way as the forward. The second part, involving the computation of the gradients with respect to the internal weights, followed the same principle but required some synchronization inside. In particular there are 10 operations in this section, each of them involving a write on one different weight matrix. Since the weight matrices are shared across the different images of the mini-batch, some synchronization was required to avoid race conditions. Each of the 10 operations was wrapped in a different critical section so that if a thread is inside one of them, no other threads can enter it, but they can still enter other critical sections which are not busy.

From the previous description it's clear that in the forward pass we can achieve the best speedup, that in theory can be as high as the number of threads. This is not true for the backward since synchronization is involved.

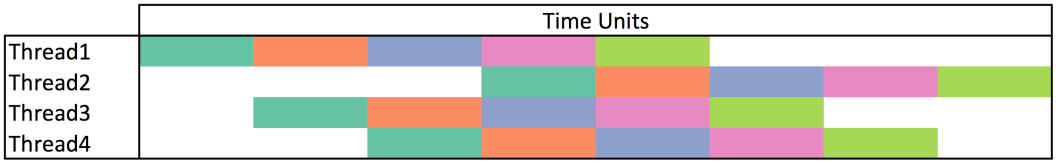



Figure 5.3. Potential thread synchronization pattern. Different colors represent different critical sections.

In Figure 5.3 we can see an example of one of the possible schedules for the update step in the backward pass. We can clearly see that two threads can never be in the same critical section at the same time, but thanks to the fact that we have many of them we can still achieve a good parallelism. In the general case of $n$ threads, batch size of $n$, $m$ critical sections and assuming all the sections require one unit of time, the best theoretical speedup (ignoring super-linear effects) is as follows:

$$S_{best} = \frac{mn}{m+n} \tag{5.1}$$

Since in the implementation of this thesis we have $m = 10$, the best theoretical speedup with this approach is

$$\lim_{n \to +\infty} \frac{10n}{10+n} = 10 \tag{5.2}$$

This result is only theoretical and represents an upper limit to the achievable speedup for the update step in the backward pass. It's also important to note that, having a parallelization over the images of a mini-batch of size $n$, the best number of threads to use

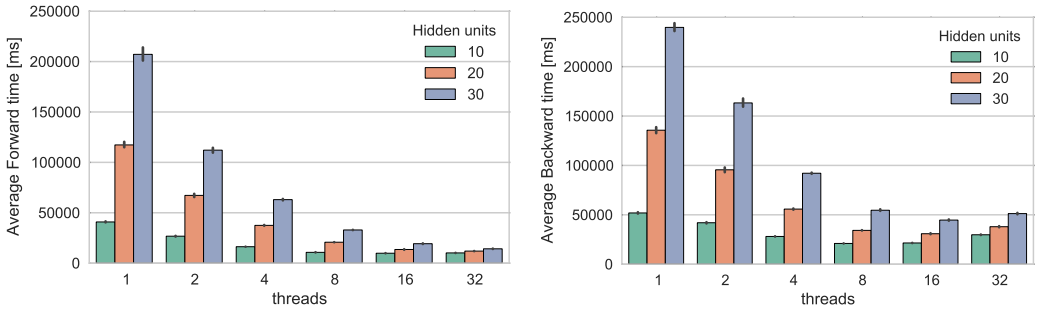
is $kn$ with $k \in \mathbb{N}$. This is because we want the workload of threads to be balanced in order to not let any threads be waiting for long time.

To have a better understanding of the performance improvement, a series of performance measurements was performed and they are shown in the next section.

### Performance comparison

To quantify the performance improvement of the parallelization strategy described in the previous section, a series of experiments was set up. The network used for the experiments was the same as the one described at the beginning of Section 5.3.

In the first set of experiments, we fixed the mini-batch size to 32 and measured the performance of forward and backward passes. We fixed the number of hidden units in the set $[10, 20, 30]$ and varied the number of threads in the set $[1, 2, 4, 8, 16, 32]$. The separated results for forward and backward are shown in Figure 5.4. As we can see,




(a) Results for the forward pass.

(b) Results for the backward pass.

Figure 5.4. Scaling of forward and backward steps to higher number of threads.

increasing the number of threads reduces considerably the execution time for both the forward and backward passes. This improvement is stronger for the forward pass; as we can see, even if the execution times for forward and backward on a single core are similar, moving to 16 or 32 threads gives a better improvement to the forward compared to the backward. This is understandable for the reasons described in Section 5.3.2.

Another observation we can make is that the higher is the number of hidden units, the higher is the achievable speedup. For example with 10 hidden units the speedup on 32 threads is approximately 4x for the forward and 2.5x for the backward. On the other end, with 30 hidden units the same speedups are 14.5x and 5.3x. This is probably because with more hidden units the workload for every single thread increases a lot and so the effect of overheads due to thread management and synchronization are lowered.

Finally we noticed that the backward pass usually achieves better speedups with smaller amount of threads. For example the best number of threads for the forward turned out to be $[16, 32, 32]$ respectively for 10, 20 and 30 hidden units while the best numbers for the backward were $[8, 8, 16]$. This is again explainable by the extra

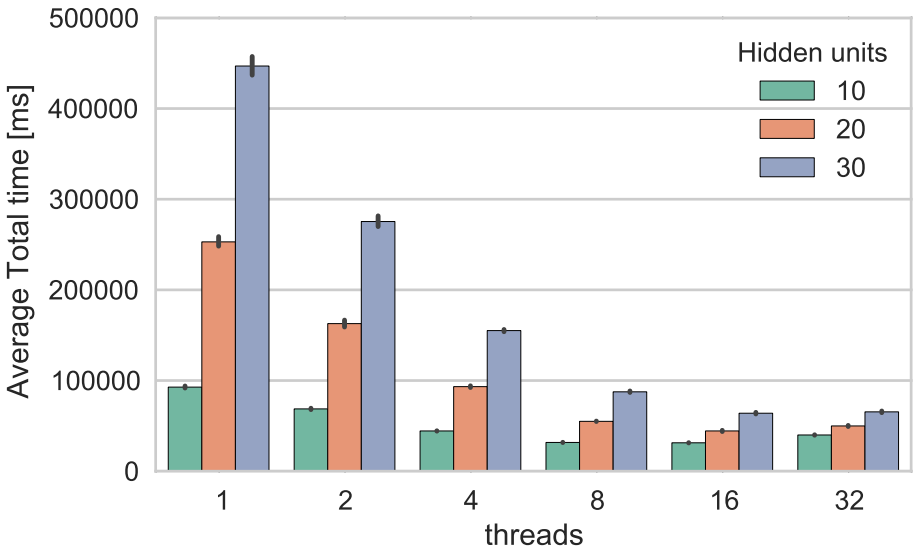



Figure 5.5. Total running time for different number of threads and hidden units.

synchronization involved in the backward, which puts additional overhead given by the fact that more threads have to check the critical sections to see if they can run. The fact of having 10 different critical sections increases this overhead so the workload needed to amortize this effect is considerable.

The total running time combining forward and backward is shown in Figure 5.5. It's clear that the overall behavior seen in the separated plots is maintained. It can also be noticed that an higher number of threads is not always the best solution. For example in the case of 10 hidden units the best speedup of approximately 3x is obtained with 16 threads while using 32 we got only 2.3x. It's difficult to know a priori what is the best number of threads, but it turns out that with higher workloads we need more threads. This means that with more hidden units (more than 30) is safe to assume that the best speedup is achieved with 32 threads.

### Pushing the speedup

Having seen that the achieved speedup increases with the number of hidden units, this number was pushed as high as possible to see what is the maximum achievable gain in performance. We therefore fixed the number of threads to 32 and run experiments for higher values of hidden units to see how high the speedup can become. Figure 5.7 shows the result of those experiments. We can see that, as expected, the speedup grows with the number of hidden units. For the forward pass an apparent limit is reached a bit below the 18x speedup. For what concerns the backward, an apparent limit isn't reached and the best achieved speedup is around 7x. The combination of the two passes

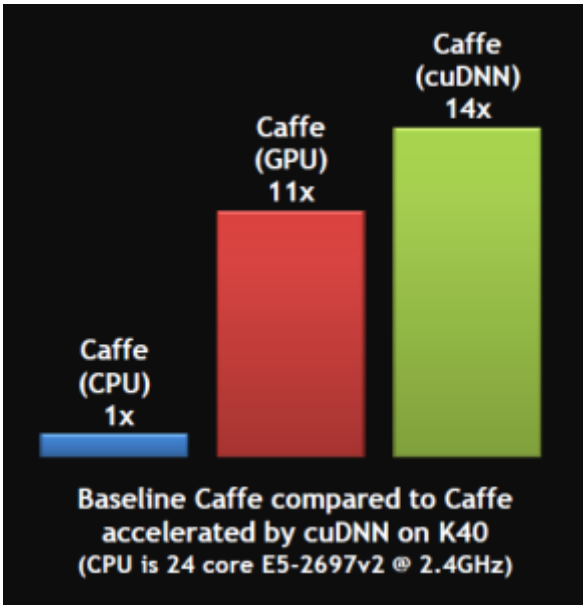
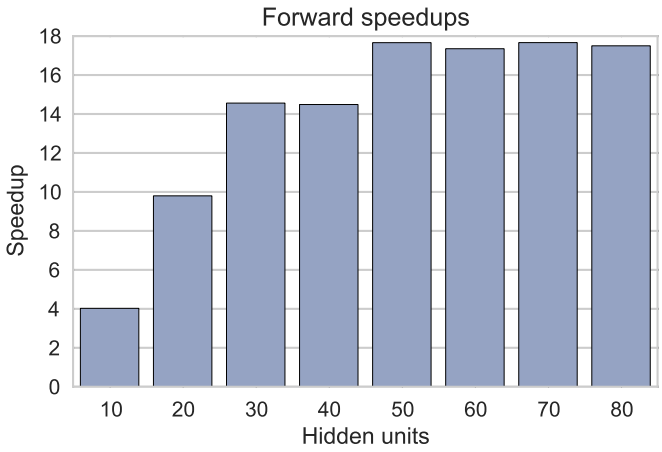


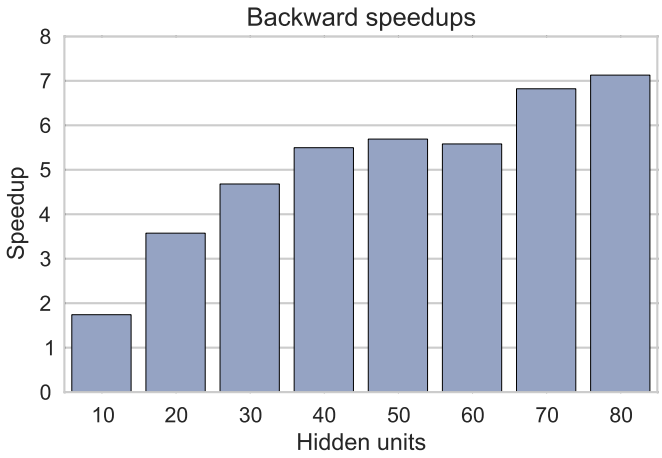

Figure 5.6. Caffe GPU performance. [cud, 2015]

gives an overall speedup of nearly 10x which can be considered good.

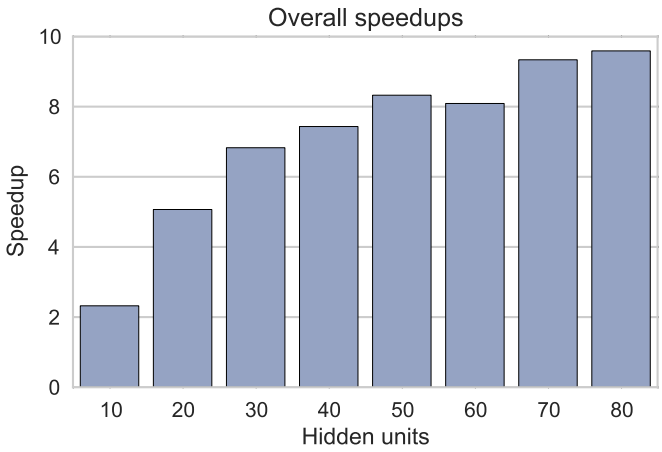
As a comparison metric, as shown in Figure 5.6, the basic GPU implementation of the convolution layer in Caffe on a high-end Nvidia card achieves a 11x speedup which is comparable to our CPU result. The image also shows a 14x speedup with the cuDNN library. This improvement is due to fine optimization at the memory level specific for the CUDA architecture, so it's not comparable to a generic approach like the one we chose for this work.




(a) Forward pass speedups.




(b) Backward pass speedups.




(c) Overall speedups.

Figure 5.7. Improvement of speedup as a function of the hidden units with 32 threads.

# Chapter 6

# Experimental results

In this chapter the MDLSTM layer and the CNN-MDLSTM architecture described in Section 4.3.1 are evaluated on a standard segmentation dataset. Section 6.1 describes the dataset and how it was prepared for the experiments. Section 6.2 presents the results of the most interesting network topologies, comparing them to the state of the art results in the literature.

## 6.1 Sift flow dataset

The dataset used for the experimental evaluation is the SIFT FLOW dataset [Liu et al., 2008]. This is composed of 2688 natural RGB images of size 256 × 256, taken in various types of environments. The macro-categories of images are maritime, urban, forest, mountain and open-country. Each image has been manually segmented into 33 classes (34 considering the presence of a missing or "unknown" label).

For all the experiments the dataset was split into 2200 images for training and 288 for validation. The dataset ships with additional 200 images explicitly meant to be used as test set.

In Figure 6.1 we can see some samples of what kind of images we can find in SIFT FLOW. As we can see there is a strong focus on both natural environments and urban environments, while other types of image like objects or animals are nearly absent.

Figure 6.2 shows the distribution of the labels. We can see that there are some very common classes, like "sky" and "building" which cover most of the pixels. In fact, the 4 most frequent classes cover around 65% of the dataset. On the other hand, many classes are very rare like "bird", "moon" and "cow". These classes are even missing from the test set and, as we will see, this highly unbalanced distribution will strongly affect the classifier learning behavior.

A strange peculiarity of the dataset is represented by the missing (unknown) values. This class represents the $5^{th}$ most common label and this is unexpected since in the original dataset description is not even mentioned. We decided to make no distinction

(a) Coast category. (b) Forest category. (c) Highway category.

(d) Insidecity category. (e) Mountain category. (f) Street category.

Figure 6.1. Examples of images from the SIFT FLOW dataset.

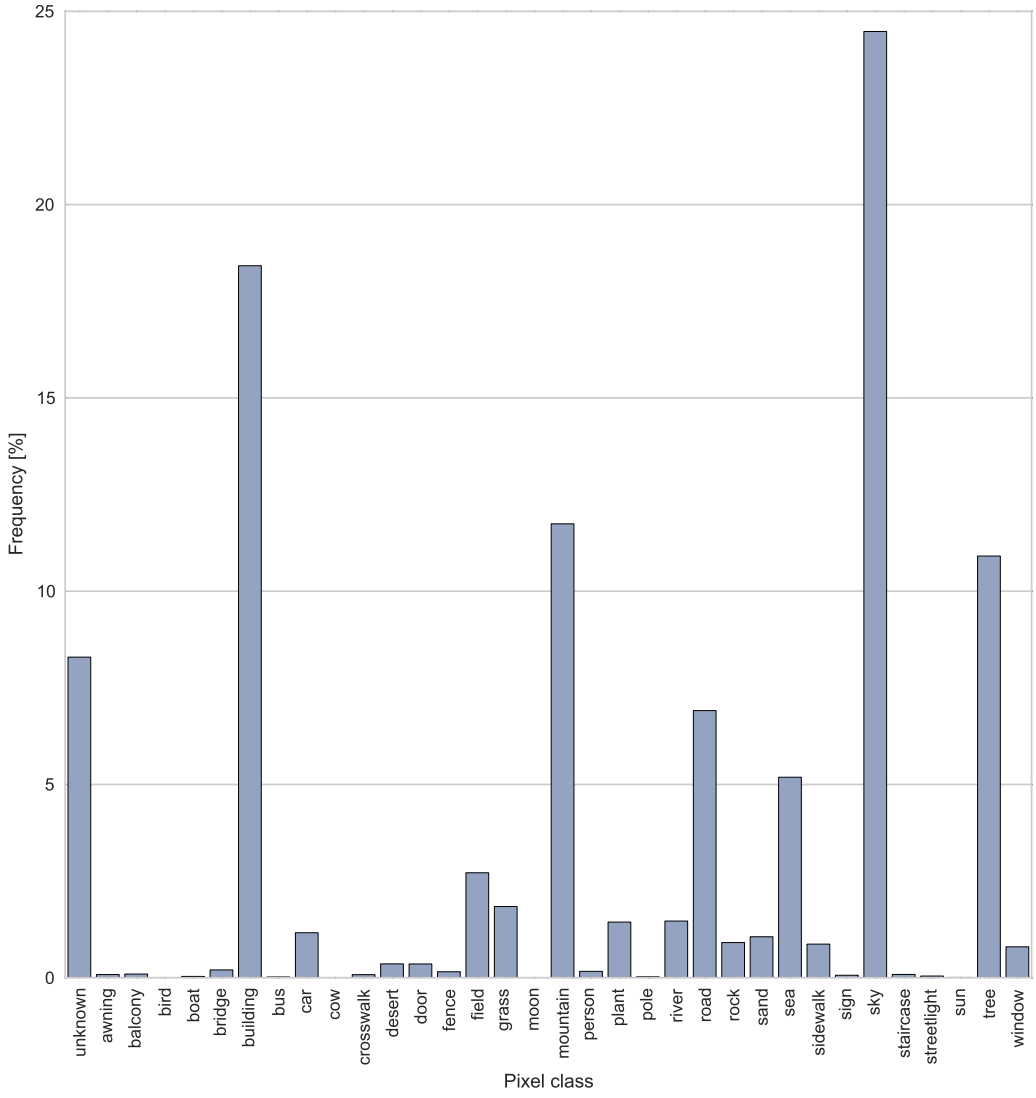



Figure 6.2. Class frequency distribution of labels in SIFT FLOW.

between it and other labels during training, handling the "unknown" pixels as any other pixel of the dataset. Only during test time we ignored all the "unknown" pixels.

## 6.2 Experimental results

In this section we present the results obtained on SIFT FLOW with some of the trained networks and compare them to other methods in the literature.

### 6.2.1 Architectures

All the networks were a variant of the CNN-MDLSTM hybrid (CMDLSTM) described in Section 4.3.1. The input layer was a convolution layer with $kernel\_size = 3$ and $stride = 1$ followed by a MAX-POOLING layer with $kernel\_size = 4$ which downscales the input images to $64 \times 64$.

Every MDLSTM layer was actually a multi-directional variant of MDLSTM with all the four directions. Whenever we used multiple MDLSTM layers, they were interleaved with fully convolutional layers.

At the end we have a deconvolution layer with $kernel\_size = 4$ and $stride = 4$ to make the upsample of the volumes to the original image size, which is $256 \times 256$.

All the layers were randomly initialized sampling from a random gaussian with zero mean and standard deviation of 0.01. The only exception is the last deconvolution layer in which we used the initialization technique from Glorot and Bengio [2010] which is called *Xavier initialization*.

The batch size was set depending on the number of available cores. The machines used for training had 8 or 64 cores, so on the first we used a batch size of 4 while on the second we set it to 32.

The learning rate was fixed to 0.05 when the batch size was 4 and 0.1 when the batch size was 32. The momentum was always fixed to 0.9.

### 6.2.2 State of the art

There are many publications using the SIFT FLOW dataset to report their results. From those we extracted the most important ones and summarized them in Table 6.1.

The result from Byeon et al. [2015] is an MDLSTM with no convolutions. This approach also uses a windowing system to handle the input which downscales the images and a corresponding upscaling method (post-processing) to get the final segmentation.

The rCNN by Pinheiro and Collobert [2013] is a particular architecture which realizes a recurrent convolutional network. This approach feeds a CNN with its previous output and achieves very good results. It has a variable speed depending on the internal parameters, but it ranges from 0.2 to 10 seconds per image.

| Method | Pixel acc. [%] | Class acc. [%] | # Parameters |
|---|---|---|---|
| MDLSTM [Byeon et al., 2015] | 70.1 | 20.9 | 168K |
| rCNN [Pinheiro and Collobert, 2013] | 77.7 | 29.8 | - |
| multiscale CNN [Farabet et al., 2013] | 78.5 | 29.6 | - |
| MRF [Tighe and Lazebnik, 2010] | 76.2 | 29.1 | - |
| 50_CMDLSTM | 71.6 | 20.1 | 216K |
| 70_CMDLSTM | 73.5 | 21.2 | 358K |
| 50-50_CMDLSTM | 74.4 | 24.1 | 366K |

Table 6.1. The current state of the art results on the top, plus the result obtained by the method proposed in this thesis in the bottom part. Pixel accuracy is the overall percentage of correctly classified pixels. Class accuracy is the average accuracy over the different classes weighted in the same way. The results reported for our method ignore pixels labeled as "unknown", which is what we believe it's done by the other methods (even if they are not explicit). In the names "X-Y_CMDLSTM" the first part represent the number of hidden units used for each MDLSTM layer.

The multiscale CNN approach by Farabet et al. [2013] involves the use of a convolutional architecture. The result in Table 6.1 was obtained with an additional post-processing step, required to get the accuracy level reported in the table.

The final result of Tighe and Lazebnik [2010] was obtained with an approach based on Markov random fields and it is also quite slow, reporting a 31 seconds time for the classification of a single image with a parallel implementation.

In contrast, the results of our CMDLSTM approach were obtained with no need for pre or post processing. Moreover our implementation is also relatively fast, achieving a throughput of 1.9 seconds per image with the biggest network.

Focusing on the results of our method, we can see how it generally has a slightly lower accuracy (74.4% vs 78%) and also a lower average class accuracy (24% vs 30%) compared to the state of the art. This suggests that the CMDLSTM has troubles when it comes to predict less frequent classes, while it has performance comparable to the other methods on the most frequent classes. From Table 6.1 we can also see how increasing the number of parameters and/or the number of layers led to an improvement both in overall accuracy and in mean class accuracy. Taking into consideration the "unknown" class would have given around 3% worse pixel accuracy but a slightly better mean class accuracy.

For the above reasons, even if our accuracy results are a bit worse that the current state-of-the-art, we can claim that we are close to it. Furthermore we have room for a lot of improvement since we didn't optimize the architecture both in terms of hyper-parameters and structure. This was mainly due to the short time available for experiments (around two months) and the long time required by the training of a single

network (from a few days to entire weeks) even with the parallel implementation. In the next section we will give more details about these results.

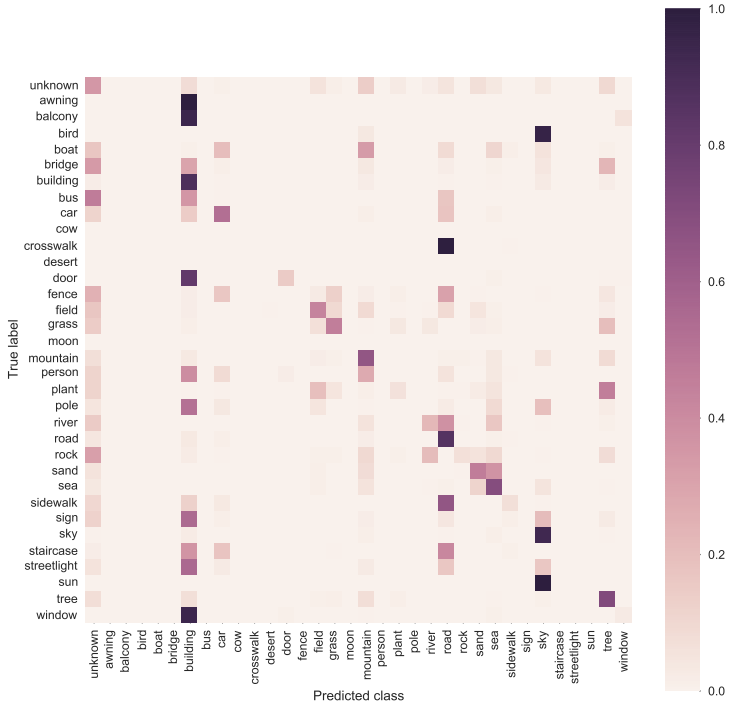
### 6.2.3 Result analysis

As shown in Table 6.1, the best result was achieved with the "50-50_CMDLSTM" network depicted in Figure 6.3, which has two stacked MDLSTM layers of 50 hidden units each. This is the biggest network we were able to train given the time limitations. Moreover in the limits of our knowledge, it's the biggest MDLSTM based network trained in the literature, with approximately 366K parameters. In this section we analyze the results obtained on the test set by this architecture in more details.


Figure 6.3. DAG represention of 50-50 CMDLSTM.

The first interesting plot to analyze is the confusion matrix, which is a table reporting for each true label of the dataset, the percentage of predicted labels for all the classes. As a consequence of this construction, the diagonal of this matrix contains the accuracy result on each class. The overall confusion matrix is shown in Figure 6.4a. In general each row of the matrix sums up to 1 but we can see that some of the rows (*cow*, *desert* and *moon*) have only light values; this is because those classes are missing from the test set.

Another observation we can make is that many classes get labeled as *building* really often. This is understandable in cases of *window* and *door* but it's more strange for classes like *person* and *car*. A possible explanation can be that the *building* class is really common in the dataset and as a consequence the network learns to predict it very well, sacrificing accuracy on other classes that come together with it. The same behavior doesn't show up for the other very common class *sky*. This can be because it's easier to detect and usually it's not contaminated by many other classes like in the case of *building*.




(a) Whole confusion matrix.


(b) Focus on the most common classes and some of the others.

Figure 6.4. Confusion matrix on test set for the best network.

As we said the *building* class is source of many errors, but also other classes cause systematic confusion to the network:

- Prediction of *building* instead of *awning*, *balcony*, *window* and *door*.
- Prediction of *sky* instead of *bird* and *sun*.
- Prediction of *road* instead of *crosswalk* and *sidewalk*.

These types of errors don't influence much the accuracy since they all involve rare classes, but on the other hand they have a big influence on *mean class accuracy* which doesn't take into consideration the frequency distribution of labels.

Looking at Figure 6.4b the differences between common and rare classes are even more clear. For example the network predicts *sky* and *building* with an accuracy around 90% but totally fails to predict *person* and *sun*.

Figure 6.4b also shows that we achieve a fairly good accuracy on the "unknown" class. This suggests that probably it would have been better to ignore it at train time and just consider pixels with that class as "unlabeled". This would allow to avoid predicting it at test time and would have probably given us a slightly higher accuracy.

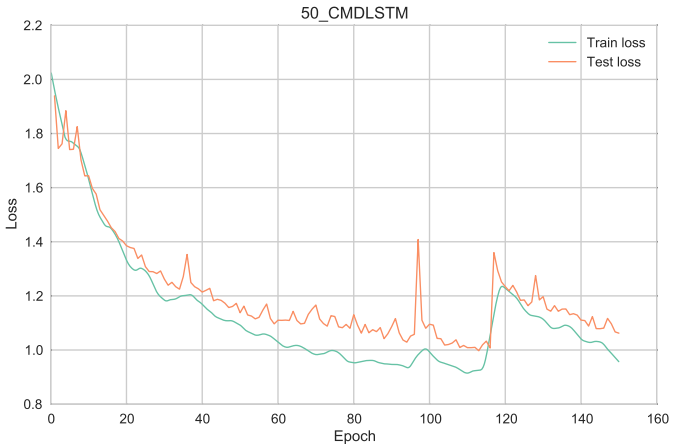
Figure 6.5 shows the training and test losses over time of both the network with best accuracy and a much smaller one. We can clearly see that the bigger network achieves better results and also has a faster convergence, meaning that having more parameters actually allows the model to learn more.

Another interesting observation is that the bigger network runs into a bit of overfitting, as showed by the substantial distance between the train and test losses. This is an expected behavior with a big network and suggests that some regularization strategy should be adopted to reduce this effect. Techniques like *dropout* [Srivastava et al., 2014] have shown to allow great improvements and should be tested in this context since they can potentially allow a better generalization and, as a consequence, an improvement in accuracy. Unfortunately it was not possible to do that due to time constraints.

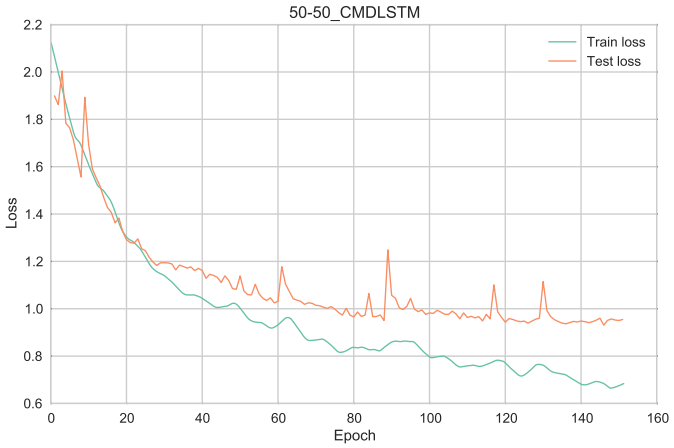
To conclude Figure 6.6 shows some segmentation examples produced by our method. In the first row we have an example of good segmentation, in which the network correctly distinguishes between *water*, *sky* and *mountain*. The only little problems are in the foggy region on the left, where the network predicts *sky* instead of *mountain*.

The second row shows a particularly difficult image, in which a wood is reflected by the water of a river/lake. First of all the human labeler made a distinction between *plants* (green) and *trees* (dark gray) while our network understandably classifies the whole wood as *trees*. On the other hand the human had no trouble classifying the water as *river*, while the network had problems due to the reflection of trees on the water. Despite of this it was still able to recognize most of the water in the image.

The third row shows a problem with human labeling, where all the people together with the road on which they are walking on are labeled as *person*. This is a clear




(a) Single layer network with 50 hidden units (216K parameters).




(b) 50-50 CMDLSTM (366K parameters).

Figure 6.5. Plot of training loss vs test loss of best network and a smaller one.

Figure 6.6. Some interesting examples of segmentation with corresponding true labels. From left to right: original image, human labeled ground truth and network output. Same color corresponds to same label.

mistake by the human which can cause problems during training. On the other hand our network gets confused in the bottom region and is not able to correctly classify the people as *person*, but at least it partially understands that they are walking on a *road*.

Finally the last row shows another weird labeling from the human operator, which decided to classify the bell tower with the label *unknown*. On the other hand our method correctly classify the whole construction as a *building*.

From those examples (and other not shown here) we can infer that our method works really well on images that have a clear distinction between classes and without superpositions, like panorama images or skyscrapers. Conversely it has problems when the image has many different uncommon classes in it, like in a urban environment with cars and people on roads and buildings.

# Chapter 7

# Conclusion

The aim of this thesis was to investigate the power of MDRNNs, focusing in particular on the problem of semantic image segmentation.

We first developed a MDLSTM layer for the open source library Caffe in order to have an efficient and versatile implementation to use in tests. To achieve an even faster performance we used OpenMP to parallelize the code. Our measurements showed that the achieved speedup is near to 10x on a 64 cores machine, which is comparable to speedups achieved through the use of GPUs.

Afterwards we integrated the MDLSTM layer with convolution layers in order to create a convolutional MDLSTM (CMDLSTM), which was able to combine the advantages of CNNs with the power of MDLSTMs. This architecture was also able to tackle the problem of image segmentation without the need of any pre or post processing and allowed the end-to-end training of the model.

Thanks to the full integration of our layer with Caffe, the effort to implement this architecture was minimal and we were able to test it on a well known semantic image segmentation dataset. Chapter 6 shows the results obtained on this dataset. We show that the obtained results are very close to the current state of the art and we present ideas for improving them.

### 7.0.1 Future work

The first thing we would like to investigate is how to prevent the state explosion problem described in Section 5.2.1. In our implementation we went around this issue by switching from single precision to double precision floating point variables. This is a temporary solution which doesn't tell us why the problem happens and how to avoid it.

Another interesting extension to this work would be to explore the possibility of implementing an efficient GPU parallelization of the MDLSTM layer and compare it to the OpenMP CPU implementation. A GPU version of the layer would have the potential

to achieve very high speedups, with the advantage of being usable with the other GPU layers of Caffe.

Since we had little time for experiments, we would also like to explore the potential of the CMDLSTM architecture more thoroughly, as we believe that it has the potential to advance the state of the art in the field. This boils down to trying more combinations of MDLSTM and CNN layers, together with ways of preventing overfitting as the network gets larger.

# Bibliography


Convolutional neural networks for visual recognition, July 2015. URL https://cs231n.github.io/.

Accelerate machine learning with the cudnn deep neural network library, 2015. URL http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/.

2015. URL http://zhaoshuaijiang.com/2014/12/15/paper_rnn_tts/.

Jose M Alvarez, Theo Gevers, Yann LeCun, and Antonio M Lopez. Road scene segmentation from a single image. In *Computer Vision–ECCV 2012*, pages 376–389. Springer, 2012.

Pierre Baldi and Gianluca Pollastri. The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *The Journal of Machine Learning Research*, 4:575–602, 2003.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

Arthur E. Jr. Bryson and Yu-Chi Ho. Applied optimal control: Optimization, estimation, and control: Arthur e. bryson, jr. and yu-chi ho: Blaisdell publishing company (a division of ginn and company), waltham, mass. (1969), 481 pp. *Automatica*, 6(6):825 – 826, 1970. ISSN 0005-1098. doi: http://dx.doi.org/10.1016/0005-1098(70)90033-6. URL http://www.sciencedirect.com/science/article/pii/0005109870900336.

Wonmin Byeon, Thomas M Breuel, Federico Raue, and Marcus Liwicki. Scene labeling with lstm recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3547–3555, 2015.

Antonin Chambolle. Image segmentation by variational methods: Mumford and shah functional and the discrete approximations. *SIAM Journal on Applied Mathematics*, 55(3):827–863, 1995. doi: 10.1137/S0036139993257132. URL http://dx.doi.org/10.1137/S0036139993257132.

Dan Ciresan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2843–2851. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4741-deep-neural-networks-segment-neuronal-membranes-in-electron-microscopy-images.pdf.

Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237, 2011.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. *arXiv preprint arXiv:1411.4389*, 2014.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12: 2121–2159, 2011.

Reinhard Eckhorn, Herbert J Reitboeck, Martin Arndt, and Peter Dicke. Feature linking via stimulus-evoked oscillations: experimental results from cat visual cortex and functional implications from a network model. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 723–730. IEEE, 1989.

Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1915–1929, 2013.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

Felix Gers, Jürgen Schmidhuber, et al. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE, 2000.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

Faustino Gomez and Jüergen Schmidhuber. Sequence learning and recurrent neural networks, 2013.

Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 545–552, 2009.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Multi-dimensional recurrent neural networks. *CoRR*, abs/0705.2011, 2007. URL `http://arxiv.org/abs/0705.2011`.

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

Barbara Hammer. On the approximation capability of recurrent neural networks. *Neurocomputing*, 31(1):107–123, 2000.

RM Haralick and LG Shapiro. Robot and computer vision (vols. 1 and 2), 1992.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

Herbert Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. GMD-Forschungszentrum Informationstechnik, 2002.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Philipp Krähenbühl and Vladlen Koltun. Efficient inference in fully connected crfs with gaussian edge potentials. *arXiv preprint arXiv:1210.5644*, 2012.

Yann LeCun and Corinna Cortes. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Ce Liu, Jenny Yuen, Antonio Torralba, Josef Sivic, and William T Freeman. Sift flow: Dense correspondence across different scenes. In *Computer Vision–ECCV 2008*, pages 28–42. Springer, 2008.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *arXiv preprint arXiv:1411.4038*, 2014.

Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

Pedro HO Pinheiro and Ronan Collobert. Recurrent convolutional neural networks for scene parsing. *arXiv preprint arXiv:1306.2795*, 2013.

Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document, 1961.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45(11):2673–2681, 1997.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Fernando de Souza Campos Paulo R. Aguiar Eduardo C. Bianchi Thiago M. Geronimo, Carlos E. D. Cruz. 2013-01-16. URL `http://www.intechopen.com/books/export/citation/BibTex/artificial-neural-networks-architectures-and-applications/mlp-and-anfis-applied-to-the-prediction-of-hole-diameters-in-the-drilling-process`.

Joseph Tighe and Svetlana Lazebnik. Superparsing: scalable nonparametric image parsing with superpixels. In *Computer Vision–ECCV 2010*, pages 352–365. Springer, 2010.

Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

Francesco Visin, Kyle Kastner, Kyunghyun Cho, Matteo Matteucci, Aaron Courville, and Yoshua Bengio. Renet: A recurrent neural network based alternative to convolutional networks. *arXiv preprint arXiv:1505.00393*, 2015.

Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE,* 78(10):1550–1560, 1990.

Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip Torr. Conditional random fields as recurrent neural networks. *arXiv preprint arXiv:1502.03240*, 2015.