Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische
Friedrich-Wilhelms-Universität Bonn

Master thesis

Recurrent Convolutional Neural Networks for Object-Class Segmentation

Author:
Mircea-Serban Pavel

First Examiner:
Prof. Dr. Sven Behnke
Second Examiner:
Prof. Dr. Simone Frintrop
Advisor:
Hannes Schulz

Submitted: 29th January 2015

Declaration of Authorship

I declare that the work presented here is original and the result of my own investigations. Formulations and ideas taken from other sources are cited as such. It has not been submitted, either in part or whole, for a degree at this or any other university.

Location, Date	Signature

Abstract

Object-class segmentation is a computer vision task which requires labeling each pixel of an image with the class of the object it belongs to. Deep convolutional neural networks (DNN) are able to learn and exploit local spatial correlations required for this task. They are, however, limited by their small and fixed-sized filters, which limits their ability to learn long-range dependencies. Recurrent Neural Networks (RNN), on the other hand, do not suffer from this restriction. Their cyclic interpretation allows them to model long-range dependencies. This property might be especially useful when labeling video sequences, where both spatial and temporal long-range dependencies occur. In this thesis, we inspect several promising RNN architectures for this purpose. We investigate how we can consider past and future context in the prediction process by comparing networks that process the frames one by one with networks that have access to the whole sequence. For implementing the proposed models we use the CUV and CUVNET libraries, built on top of the CUDA architecture, to speed up the training and evaluation process. We will evaluate our models by comparing to non-recurrent (i.e. single frame) models on the NYU Depth v2 video sequence database.

Contents

List of Figures

List of Algorithms

1. Introduction

In the attempt to build autonomous systems capable of interpreting the environment and of extracting relevant information from it, researchers often resort to using biologically inspired models. Just as the human vision system is able to work on different levels of abstraction by building high level features from low level ones, segmenting an image consists in partitioning the pixels into distinct groups (segments) which define the new set of features that are to be analyzed further.

Object-class segmentation is a Computer Vision task, aiming to label each pixel of an image with the class of the object it belongs to.

Our task is to perform object class segmentation on RGBD video frames. An RGBD frame consists of color and depth information, captured by Microsoft Kinect cameras, providing a three dimensional perspective of the environment. In addition to the color and depth information, we also have access to a temporal context consisting of frames which are temporally close to the current frame.

1.1. Approach

The idea of a convolutional neural network was introduced first in Fukushima, 1980 but refined especially in the last decade, when the interest in such models increased, due to the fact that they were suitable for parallelization, in the context of a rapid development of parallel architectures such as Graphical Processing Units (GPUs).

Our approach starts with a convolutional model to which we add recurrent connections, transforming it into a Recurrent Neural Network (RNN). Such recurrent connections empower the model with the capability of processing sequences of inputs, and thus of accumulating information over time, exploiting temporal dependencies. Such a dynamic model relies on several measurements taken over time, allowing it to derive a more detailed perspective. Moreover, noisy measurements have a smaller impact compared to the case when the model only relies on the current time step. Another advantage is in the ability to better understand the dynamics of the environment (e.g. motion of the objects).

In this work, three different neural architectures will be implemented and tested. In the first version, the network processes all inputs together. The second one processes each input in one time step, in the order given by the sequence. The third architecture, in addition, processes the inputs in the reverse order as well and combines the two results. Moreover, we test several extension ideas that can be applied to one or more of our networks in order to, hopefully, improve the learning capabilities.

For testing our network we will use the NYUD dataset (Silberman et al., 2012), consisting of RGBD video sequences. Our results will first be compared to those of the convolutional model, which we use as baseline, and then to the other model's results as well.

1.2. Structure of the thesis

In the next chapter we discuss related work in object-class segmentation and deep neural networks, summarizing several approaches. Chapter 3 consists of the theoretical foundations on which the model was design, followed by a chapter describing in detail the networks used. The experiments carried and their result are presented in Chapter 5. Chapter 6 concludes this thesis, also describing further ideas that could improve the model.

2. Related work

In Grangier et al., 2009 a deep convolutional network is trained in a greedy manner, starting by training the first layer, and adding the rest of the layers, one at the time, re-training the whole network in every step. This approach was able to outperform models such as Conditional Random Fields and Random Forests in the task of object-class segmentation.

Socher et al., 2011 describes the recursive neural network (should not be confused with recurrent neural network) as a deep network, receiving as input an over-segmented image and recursively merging the segments. This model minimizes a loss that encourages semantically similar segments to be aggregated together.

In Farabet et al., 2013, the authors propose a convolutional network able to learn feature vectors that represent regions of different sizes centered on each pixel. These features are afterwards post-processed in order to generate the final labeling of the entire scene.

A convolutional network is also used in Schulz and Behnke, 2012, where the input consists of ZCA (Zero Component Analysis) and HOG (Histogram of Oriented Gradients) channels. Training is done also layer-wise, one layer at a time, but the outputs of the already trained part of the network are reused. Another improvement is the use of pairwise class localization filters, a convolution with a large filter applied on top of the output layer, which is able to repair the prediction to some extent from spurious observations.

Recent work of M. Jung, 2014 uses a recurrent neural network for recognizing human actions. They introduce the multiple timescale recurrent neural network (MTRNN), as a network able to process images at different scales, but also able to work at multiple timescales by incorporating a time constant in each layer which adjusts the timescale. They use a larger value for this constant in the top layers (slower timescale dynamics) than in bottom layers.

One example of task where deep neural networks currently achieve state of the art results is the digit recognition task, where the lowest classification error on MNIST dataset was obtained using a model based on several convolutional neural network classifiers (Ciresan et al., 2012).

Another notable result, this time in unsupervised learning, is of deriving concepts out of unlabeled data consisting of random video frames from Youtube (Le et

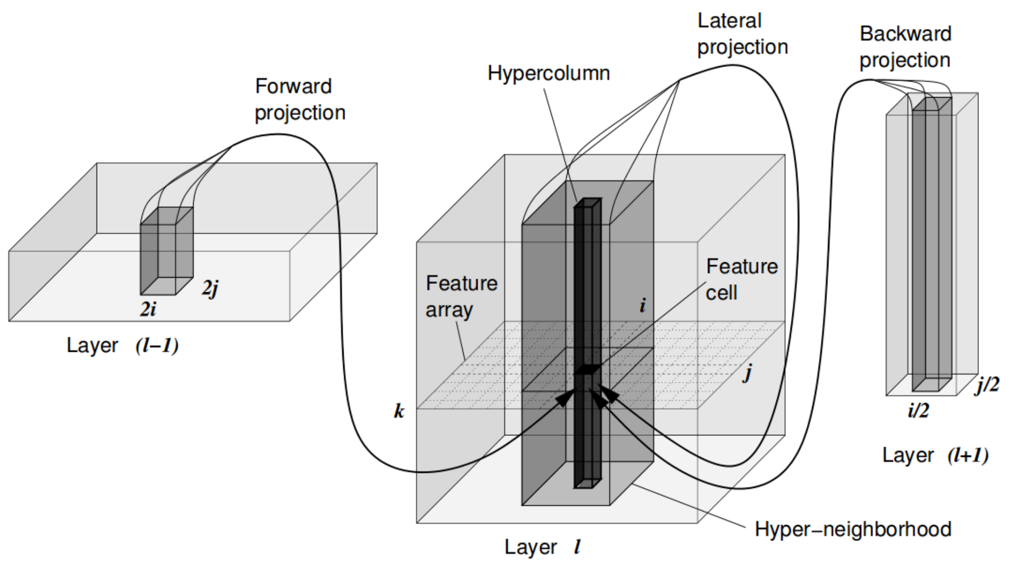


Figure 2.1: Neural Abstraction Pyramid showing the three types of connections: forward, backward and lateral (Behnke, 2003)

al., 2012). The authors state that they obtained neurons that function as detectors for faces, human bodies and cat faces.

In the ImageNet large-scale visual recognition challenge (ILSVRC) of 2014, GoogLeNet (Szegedy et al., 2014), a nine-layer deep neural network was able not only to win the classification and detection competitions, but also to double the results quality on both tasks, compared to those of the last year.

We further describe some of the related work that significantly influenced and inspired our approach or parts of our design. For each we will outline the ideas that we use, and also ways to adapt those ideas to our needs.

2.0.1. Neural Abstraction Pyramid

The Neural Abstraction Pyramid is defined in Behnke, 2003 as "a neurobiologically inspired hierarchical neural network with local recurrent connectivity". This architecture will be the foundation of our design. Figure 2.1 depicts the overview of the network together with its three different types of connections: forward, backward and lateral.

The forward and backward connections capture the correlation between high-level and low-level features, while the lateral ones are responsible for the correlations between a feature and its neighboring features.

The number of maps is doubled with every new level of abstraction. This however does not double the memory consumption of the new layer since the size of the representation is at the same time halved in both dimensions. In other words, as the features become more and more abstract, their number grows as well.

The output of the network is obtained on its bottom layer. This gives another strong advantage to such a recurrent architecture, because compared to a simple feed forward network which constructs the output in the top layer, the pyramid generates a full scale representation of the output, without needing to further upscale it when evaluating it against the ground truth. This is especially useful for tasks where we have to produce a label for each individual pixel such as our object-class segmentation task.

The network was tested on the tasks of digit recognition and face localization. In the task of digit recognition, the MNIST dataset was used and small occlusions were simulated. The network showed capability of filling the occlusions and thus of keeping a long range memory.

For the task of face localization, the BioID (Jesorsky et al., 2001) dataset was used. Primarily the task was to detect, for each face in the dataset, the position of the eyes. This experiment was repeated afterwards while simulating the movement of the input, teaching the network to track the eyes of the person.

For training the network, RProp was used, together with back propagation trough time. RProp has the property of counteracting the effects of vanishing gradients and exploding gradients. More information about those phenomena are provided in Section 3.2.2.

As previously stated we use this model as base for our design, and we further outline the ideas that will also be present in our network. We use all three connections types, but we implement the lateral connections differently, in a way that allows us to benefit from parallelism. We also construct the output in the bottom layer. Training is done similar, but using a RProp method suitable for mini-batch learning, namely RMSProp. One difference is that in our design we keep the number of filters constant for all abstraction levels. This is a consequence of observing the fact that doubling the number of filters every new abstraction layer results in a too complex model for the dataset that we use.

2.0.2. Bidirectional Recurrent Neural Network

Bidirectional Recurrent Neural Network (BRNN) were successfully used for the tasks of speech recognition in Graves et al., 2013 and for the task of image segmentation in Graves, 2012 The Bidirectional Recurrent Neural Network (BRNN) processes sequences of inputs in both directions. This way, the network is able to make use of both the past and the future context. The brief idea behind the BRNN is of using two different networks, each processing the sequence in one direction.

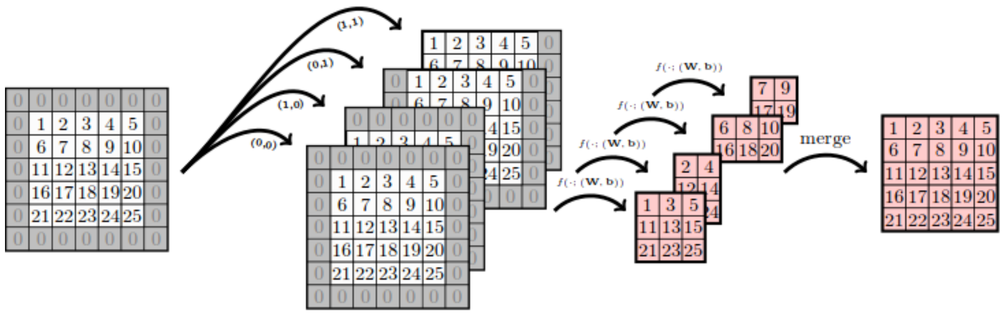

(1,1)
(0,1)
(1,0)
(0,0)
$f(\cdot;(\mathbf{W},\mathbf{b}))$
$f(\cdot;(\mathbf{W},\mathbf{b}))$
$f(\cdot;(\mathbf{W},\mathbf{b}))$
$f(\cdot;(\mathbf{W},\mathbf{b}))$
merge

Figure 2.2: Scene Inference (Pinheiro and Collobert, 2013)

The final output of the network is obtained by combining the two outputs.

Since we work with sequences of video frames as input, using such a model might prove beneficial. We apply the idea differently. In Graves, 2012 the BRNN is applied to learn long range spatial dependencies. We already have such a mechanism for long range spatial dependencies through the lateral connections and thus we are more interested in exploiting the temporal dependency in both directions, the future and the past context.

2.0.3. Scene Inference

Pinheiro and Collobert, 2013 approach to the object-class segmentation task is to use a recurrent neural network with a structure similar to Jordan networks, presenting the same input image to all the temporal copies, but at different scales. This is motivated by the resolution decrease of the input, due to the pooling layers, after being forwarded through the network.

In order to obtain the labeling of the full-scale image, the authors use a technique called “scene inference” which requires an extension of the input set with shifted versions of the input image and a interleave operation on the outputs.

This network structure is able of learning long range spatial dependencies between pixels, but at the same time of keeping a small filter size, speeding up the involved calculations.

This idea inspired us in implementing an operator that upscales the output of a hidden layer such that we can propagate it backwards. While scene inference might be more accurate then our simple upscaling operation, it also slows down both training and prediction, by a constant factor which depends on the depth of the network and the downscaling factor of the pooling operations.

Also, the purpose for using such an operator is different in our case. While they build the full scale output by interleaving high level representations obtained in the highest level of the network, we already have a full scale output since we obtain it on the bottom level of the network. We instead use upscaling as part of the backward and lateral connection, mapping pixels from a high level representation into a region of a lower level representation.

3. Foundations

3.1. Convolutional Neural Networks

Convolutional Networks are multi-layered learning models inspired from the biological visual cortex. LeCun et al., 2010 describe each stage in a convolutional network as being composed of “a filter bank, some non-linearities, and feature pooling layers”.

The modern architectures of convolutional neuron networks are based on the architecture of the neocognitron (Fukushima, 1980), that uses S (simple) and C (complex) layers which have similar functionality to the filter-bank and the pooling layers respectively, while hyperbolic tangent units are used for providing non-linearity.

Due to the sparse connectivity between layers, such networks are well suited for vision tasks, since they exploit the local spatial correlation between pixels. Also, by having a larger number of layers they allow a better generalization, low level features derived in the lower layers of the network being reused to create high-level features.

Filter-bank layer

This layer is a collection of filters, each being responsible for learning one feature. A filter is a map of neurons that responds to a feature. For this, the neuron is

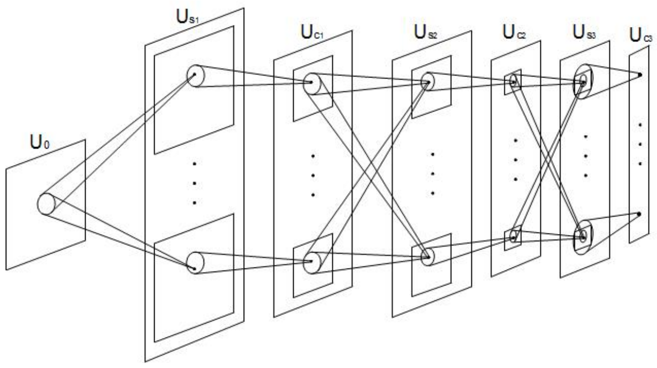


Figure 3.1: Structure of neocognitron. Fukushima, 1980

connected through plastic connections to all cells located in a rectangular patch (receptive field) centered at the position of the neuron. During the feed forward step, a weighted sum is calculated for each neuron, and the bias term is added. The output of a neuron is:

$$y_j = b_j + \sum_i k_{ij} * x_i \tag{3.1}$$

where y_j is the output feature map j, x_i is the patch of the input feature map i, k_{ij} are the weights connecting the two feature maps (kernels) and b_j is the bias. Since the whole filter responds to the same feature, it is able to detect it in any position. Another consequence to this fact is that all the neurons of a filter share the same weight set. Apart from the weight sets and biases, a filter-bank is also defined by structural parameters such as:

- **Number of Filters**
  The number of features learned by this layer.
- **Filter Size**
  The size of the receptive field.
- **Stride**
  Distance in pixels between successive applications of the filter.
- **Padding**
  When applying a filter to the whole image, the size of the image would normally decrease with the filter size due to the border effect. To counter this effect the image can be padded with zeroes (no activation) before applying the filter

Feature Pooling Layer

Pooling layers are responsible for collecting each feature within a receptive field. They do not derive new features, keeping the original feature set, and thus the number of output filters is equal to the number of input filters. The activation of a neuron in such a filter depends on the presence of the corresponding feature in the receptive field. However, the way this activation is calculated makes the difference between different pooling methods. Two such methods are max-pool and average-pool which calculate the maximum activation within the receptive field P and average respectively. One effect of this operation is that it allows a certain shift-invariance, since the features are active even if they are not exactly in the expected position as long as they are inside of the receptive field. Another effect is the resolution decrease, a more abstract representation, with higher-level features, being produced.

$$y = \frac{1}{|P|} \sum_{x \in P} x \qquad (3.2) \qquad\qquad y = \max_{x \in P} x \qquad (3.3)$$

Non-liniarity Layer

In their analysis of nonlinearities effect in deep neural networks, acknowledge that although sigmoid neurons are "more biologically plausible than hyperbolic tangent neurons", in training multi-layer neural networks hyperbolic tangent used as transfer function leads to better result . As a consequence, in the traditional architectures hyperbolic tangent (tanh) was the default nonlinearity unit used.

However, lately this aspect was investigated, research studies pointing to the rectifying units as better replacements for the sigmoid and tanh units. Rectifying units apply an activation function called rectifier (Equation (3.4)) that is an approximation of the softplus function (Equation (3.5)):

$$y = max(x, 0) \qquad (3.4) \qquad\qquad y = log(1 + e^x) \qquad (3.5)$$

Using such activation units leads to a network producing sparse representations, which argue to be more biologically plausible and also to have the mathematical advantage of being "more likely to be linearly separable, or more easily separable with less non-linear machinery".

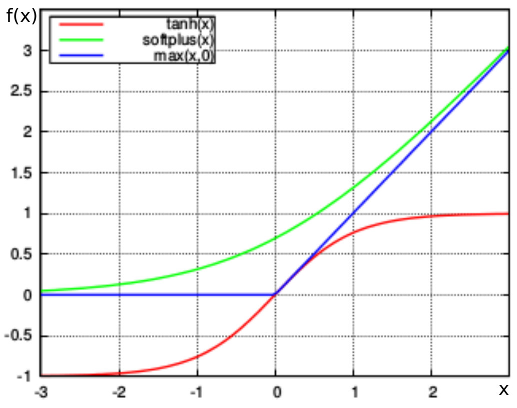


Figure 3.2: Comparison of the three nonlinear function: tanh, softplus, rectifier

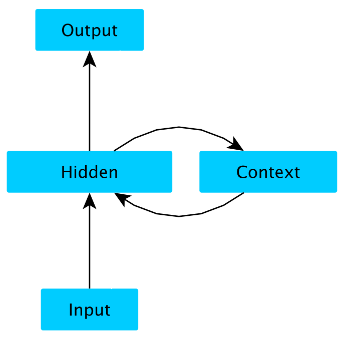
3.2. Recurrent Neural Networks

Recurrent Neural Networks (RNN) were introduced in late '80, several structures for such networks being proposed in Jordan, 1986; Elman, 1990; Hopfield, 1982. They are able to model dynamical systems, with applications in signal, speech and image processing. Our focus is the area of image processing. The recurrent connections allow the past inputs to be retained inside the network, thus providing the capability of learning temporal correlation between inputs and exploit those correlations during the prediction step.

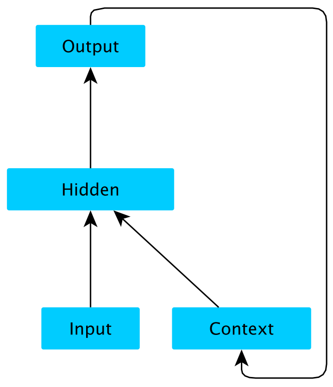
In addition to the parameters that regular neural networks have, a RNN also carries its own state derived after processing the previous inputs of the sequence. This state can be seen as a separate input.

Both Jordan and Elman networks make use of context neurons which propagate the received input only in the next iteration. While Jordan networks exhibit output-to-context and context-to-hidden connections, propagating the output again through the whole network, Elman networks use hidden-to-context and context-to-hidden connections, propagating the output of one layer as input to the same layer, in the next iteration.

In the context of video sequences with a sufficiently large frame rate, temporal dependencies are strong since we expect that one object in a frame to be in the vicinity of its previous position, making RNN a candidate for learning segmentation tasks on video sequences. However, not only temporal dependencies can be exploited in RNN. Pinheiro and Collobert, 2013 use a recurrent convolutional neural network to exploit long range spatial dependencies between pixels, while keeping the filter size small.



(a) Elman Network



(b) Jordan Network

3.2.1. Training Recurrent Neural Networks

The standard method for training Recurrent Neural Networks, and also the method used in this thesis is the Back Propagation Through Time (BPTT). The idea is to unfold the network a constant number of times by making copies of the network (temporal copies) and replace each recurrent connection with a forward connection to the next temporal copy. Thus, the network's structure is transformed in a direct acyclic graph, being trainable with the regular backpropagation method. One thing to note is that the weight changes are accumulated over all temporal copies. Since each temporal copy emulates the state of the network at a time-step, each copy receives the input corresponding to the respective time-step.

3.2.2. Vanishing and Exploding Gradient Problem

One of the reasons why recurrent neural networks are not as popular as other models is the difficulty of training them. Two phenomena that make training neural networks a challenging task are the vanishing and the exploding gradient. Both phenomena were first analyzed formally in Bengio et al., 1994. To summarize their analysis together with a more recent analysis of Pascanu et al., 2013, we start from the equation of the gradient (we consider the partial derivative of the weights) in the context of using BPTT. $\mathcal{E}_t$ denotes the error at time t, s_t denotes the hidden state of the network at time t and W the weight set.

$$\frac{\partial \mathcal{E}_t}{\partial W} = \sum_{i=0}^{t-1} \frac{\partial \mathcal{E}_t}{\partial s_t} \frac{\partial s_t}{\partial s_i} \frac{\partial s_i}{\partial W} \tag{3.6}$$

Further, the term $\frac{\partial s_t}{\partial s_i}$ can be rewritten by chain rule as $\sum_{k=i}^{t-1} \frac{\partial s_k+1}{\partial s_k}$. This, along with the fact that the recurrence weight matrix determines the transition from current

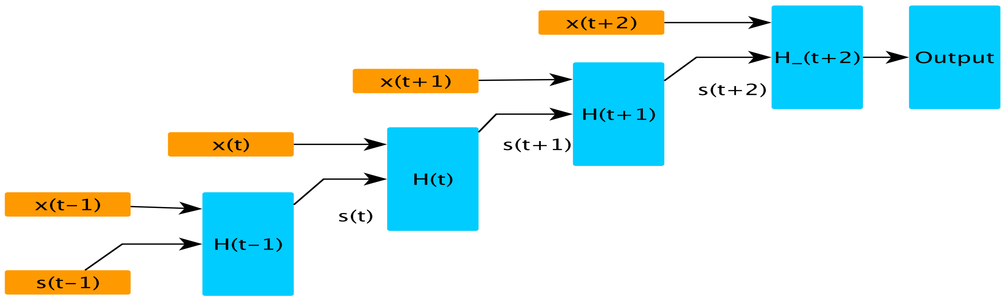


Figure 3.4: Back Propagation Through Time

state to the next state lead to raising the recurrence weight matrix to the power of $t - i$.

The conclusion of their analysis is considering the highest eigen-value λ of the recurrence weight matrix. Having λ less than one is a sufficient condition for the long term components to vanish. Having a λ higher than one is a necessary condition for exploding gradients.

During the last two decades, many solutions have been proposed for coping with those issues. Long short term memory units, introduced in Hochreiter and Schmidhuber, 1997, keep the weights of recurrent connections fixed to one, and instead learn how long to remember an input in the network, replacing it with the new value when the old one is no longer considered significant.

Another approach is the use of Hessian-free Optimization, which is a second-order optimization method, which was shown in Martens, 2010 to be successful in training deep auto-encoders. Experimental results have shown that this method deals well against the vanishing/exploding gradients issue, although a formal analysis to confirm this does not exist at the moment.

R-Prop is a method in which only the sign of the gradient is considered. The update value is calculated based on a learning factor that updates itself during training independently for every parameter. A more detailed description is presented in Section 3.4.1.

3.2.3. Exploiting Temporal Dependencies

As previously discussed, Recurrent Neural Networks (RNN) provide the capability of learning temporal dependencies between inputs, making them suitable for processing sequences.

We start by analyzing such dependencies in different areas in order to motivate the benefits of learning them.

Speech Recognition

In the task of speech recognition, the input consists of a set of utterances (units of speech bounded by silence) that are to be translated to sets of phonemes (basic units of speech that distinguish words from each other). This is challenging due to the fact that the speech is produced by different speakers with different accents and dialects which requires the network to generalize well enough to learn the structural rules that map the features derived from processing the speech signal to phonemes.

Another challenge is the fact that the network is responsible for deciding where along the speech signal a phoneme begins and where a phoneme ends. Learning the duration of each phoneme is not an opinion due to its high variance, the network having to decide based on the structure of the signal. While a deep neural network with sufficiently large number of inputs receiving the sequences as input could learn temporal dependencies, long-term dependencies would require a too complex model, unlikely to produce a good generalization.

One natural solution is the use of dynamical systems, keeping an internal state that tracks the structure of the signal. In a recurrent neural network, the internal memory is represented by the recurrent connections. The structure of the signal can be learned by observing and remembering patterns along it, that together define a phoneme. In Räsñen, 2013 a detailed analysis of such patterns that occur in natural speech is presented. Provided that the learning model is able to learn long-term dependencies, avoid the phenomena presented in Section 3.2.2, the sequence of patterns detected along the speech signal will be successfully mapped to phoneme.

Regarding the network architectures used in speech recognition, Graves et al., 2013 make use of LSTM units in a Bidirectional Recurrent Neural Network (Figure 3.5). This structure allows making use of the future, and not only of the past, context by using two hidden layers which process the data in both directions, each being considered when computing the output of the network. They showed that such an approach is able to outperform an architecture that consists of only one hidden layer.

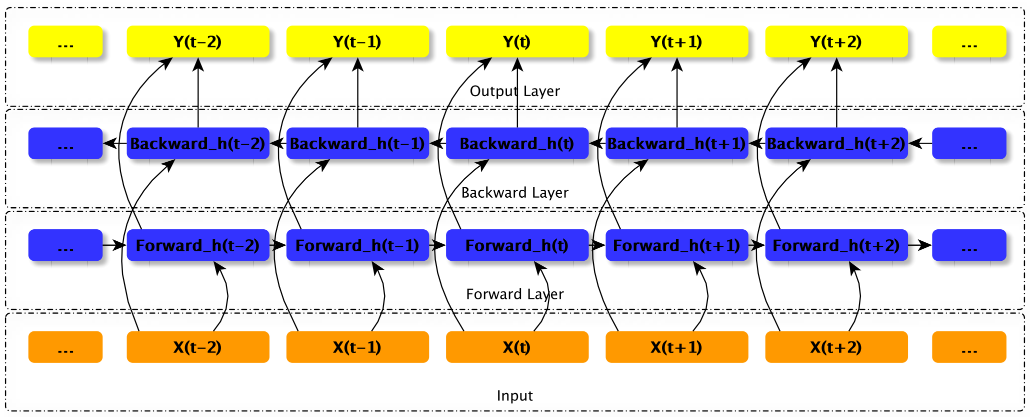


Figure 3.5: Bidirectional Recurrent Neural Network

Language Understanding

The automatic analysis of text allows the extraction of the relevant information from it, having numerous applications. One first step into analyzing a text is to derive a higher and easier to understand representation of it, that can be further on investigated depending on the application. This can be done by associating a semantic meaning to each word, determining relationships between related words.

In this scenario, dependencies can extend over a variable and large number of words which makes recurrent neural networks a suitable learning model candidate. In Yao et al., 2013, an Elmann network is used, receiving a bag-of-words representation of the input, thus each input of the network represents a small context that will be related to neighboring context. They proved the superiority of their model by comparing their results obtained on language understanding datasets with the results of models such as Support Vector Machines and Conditional Random Fields.

Image Segmentation

Dependencies in images follow both of the dimensions corresponding to the axes. In Graves, 2012 an architecture able to exploit such a multi-dimensional dependencies was presented under the name of Multi-Dimensional Recurrent Neural Network (MDRNN). Such a network connects each neuron to the neurons that precede it with respect to every dimension. In the forward step, the neurons are processed in a topological order, while during backpropagation the reverse order is used.

Another architecture suggested in the same publication is the multi-directional version of MDRNN, where a generalization of the Bidirectional Neural Network is used such that all possible 2^D directions are covered, where D is number of dimensions.

For video data we can consider having a 3D sequence and a new temporal dependency in addition to the two spatial dependencies corresponding to the two dimensions. This dependency might provide useful information, exploiting the fact that the same object is in approximately the same place in the subsequent frames.

The network might also benefit from learning the movement of the camera, enabling the ability of tracking the objects and of learning their features despite their apparent motion.

3.3. Preprocessing

Data preprocessing is an important step in the learning process since it can significantly improve the quality of the final results, but also the speed of training by passing only the relevant extracted information to the learning model. Several methods will be presented here, all being used in the work of Höft, 2014, some of them have been adapted to our different input specifications. The input consists of: the result of applying ZCA-Whitening to the image, channels of Histogram of Oriented Gradients (HOG) and Histogram of Oriented Depth (HOD) when depth is available. In order to improve generalization, we augment the input by applying several geometric transformation.

One requirement of our network is that the inputs are square sized. When the input images do not fulfill this they will be extended by a reflective border.

3.3.1. Histogram of Oriented Gradients

As its name suggests, this method uses the gradients of the input image. The gradient computed on one image pixel is a 2D vector that points in the direction in which the intensity increases the most, with the magnitude reflecting the intensity change. A histogram of oriented gradients collects the gradients from each patch of the image, in order to produce a direction of the whole patch. A final step is to quantize the gradients in the bins that cover the whole angular range.

We use a simplified version of HOG, simply calculating the gradients for every pixel and quantize them in bins. For each bin we build a separate channel representing the gradients of the corresponding bin by their magnitude. Algorithm 1 is an implementation from Höft, 2014 presenting all the steps needed for obtaining our simplified HOG representation.

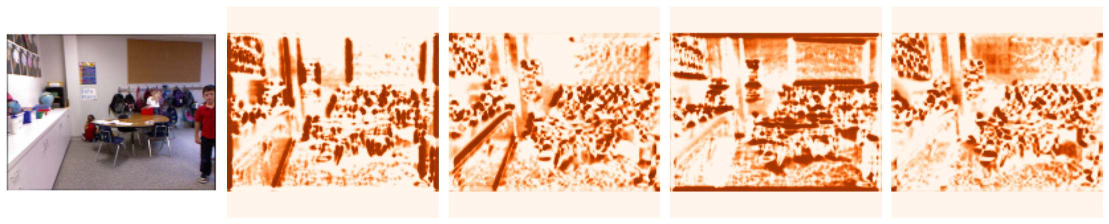


Figure 3.6: The HOG channels corresponding to four directions. The color intensity represents the magnitude of the gradient, white means zero intensity.

Algorithm 1: Simple Histogram of Oriented Gradients

```
Input: $I$ Grayscale Image, $r$ Number of HOG channels
Output: $hog$ Histogram of Gradients
$f_x = [-1, 0, 1], f_y = [-1, 0, 1]^T$ ;
$g_x = \text{convolve}(I, fx), g_y = \text{convolve}(I, fy)$ ;
$m = g_x^2 + g_y^2$ ;
$a = \text{atan2}(g_x, g_y)$ ;
foreach Pixel $x$ do
    $m_x$ and $a_x \rightarrow$ magnitude and angle of the gradient in pixel x;
    $b = \lfloor (a_x/\pi \cdot r) \rfloor$ ;
    $hog_x[b] \rightarrow$ position $x$ in the HOG channel corresponing to bin $b$;
    $a_1 = \pi/r \cdot b$ ;
    $a_2 = \pi/r \cdot (b + 1)$ ;
    $hog_x[b] += m_x \cdot (a_2 - a_x)/(a_2 - a_1)$ ;
    $hog_x[(b + 1)\%r] += m_x \cdot (a_x - a_1)/(a_2 - a_1)$ ;
$hog = \text{Gaussianfilter}(hog)$ ;
$hog = \text{normalize}(hog)$;
$hog = \min(hog, 0.2)$ ;
$hog = \text{normalize}(hog)$;
```

3.3.2. ZCA Whitening

As suggested in Krizhevsky, 2009, while models are able to learn the local correlations that exist in images, it might be a good idea to focus on higher-order correlations, which have the potential of making the model "more likely to discover interesting regularities".

We do this by an approximative decorrelation of the dataset of images, with the goal of forcing the mean to zero, and the covariance to the unit matrix. We carry the whitening process in two steps, first computing the decorrelation matrix and the mean of the samples, then subtracting the mean and applying the decorrelation

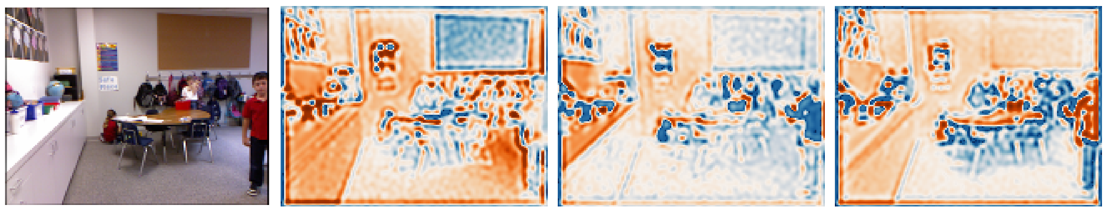


Figure 3.7: Result of ZCA whitening on each of the 3 channels. Orange represents positive values, blue negative ones

```
Algorithm 2: Computation of the decorrelation matrix $Q$
Input: $S$ Image patches from the image set
Output: $Q$ Decorrelation matrix
$\bar{S} = \mathrm{Mean}(S)$;
$C = \frac{1}{|S|}\left(\left(S - \bar{S}\right)\left(S - \bar{S}\right)^T\right)$ ;
$u, s, v = \mathrm{SVD}(C)$ ;
$s^{-1} = \sqrt{s}^{-1}$ ;
if $s_i^{-1} < 0.0000001$ then
    $s_i^{-1} = 0$
$Q = u \cdot \mathrm{diag}(s^{-1}) \cdot v^T$ ;
```

matrix. The decorrelation matrix can be seen as a collection of filters, each row in this matrix being one of the filters derived in the first step. Applying all those filters is equivalent to a matrix multiplication of the image to the decorrelation matrix. Algorithm 3 represents an implementation from Höft, 2014 for generating the decorrelation matrix.

3.3.3. Preprocessing the depth

We will cover the steps for preprocessing a raw depth image in the Section 5.2.3. Once we have followed those steps, we have an usable depth consistent to its corresponding RGB frame. We however do not present the depth map to the network as it is, because similar to the RGB case, we wish to make it easier for the network to extract the relevant information from the depth. We thus use the same HOG approach, but applied only on one channel of depth. Although the technique is the same, when we visually compare the preprocessed RGB and Depth (Figure 3.6 and Figure 3.8) we notice that preprocessing the depth produces a totally different set of features, which is exactly what we need.

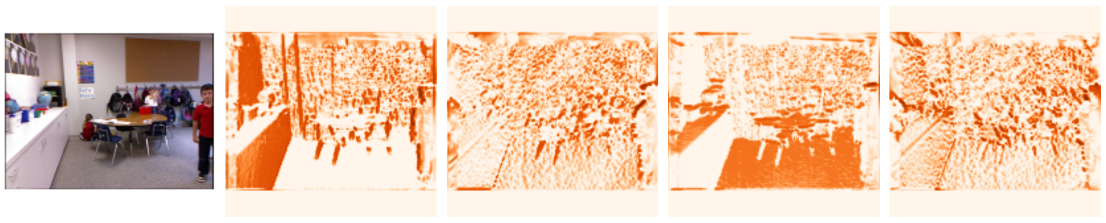


Figure 3.8: The HOD channels corresponding to four directions. The color intensity represents the magnitude of the gradient, white means zero intensity.

3.3.4. Geometric Transformations

Early experiments showed that the network tends to overfit quickly when working with real-life video sequences. One factor responsible for this is the complexity of the network. One way to counteract this effect to some extent is to build several variants of the same training set and use them in the training process.

We use the dataset augmentation implemented in Höft, 2014, adapted to our needs, which consists in a translation, a rotation and a 50% chance to flip horizontally. In our case, we need to make sure that we apply the same augmentation to the whole sequence, otherwise the apparent movement of the camera will be perturbed, the model no longer being able to properly exploit the smooth and continuous movement of the camera.

3.3.5. Generating Intermediate Ground Truths

Due to the large amount of frames present in video sequences, manually labeling all of them would require too much effort to make it a practical option. NYUD Dataset V2 for example has a total of 407,024 frames, out of which only 1449 are labeled. We wish to have ground truths for all the frames that we feed to the network, so that we can encourage the intermediate representations of the network to be closer to the structure of the expected output.

We adopt a simple solution of propagating the labels based on the optical flow. For our purpose, we do not require the intermediate ground truth to be exact, since intermediate evaluations weight less that the evaluation of the final network output.

Algorithm 3: Computation of the intermediate ground truths

```
Input: $R_{prev}$ previous RGB frame ,$R_{crt}$ current RGB frame, $GT_{prev}$
       Ground Truth for previous frame
Output: $GT_{crt}$ Ground Truth for the current frame
$FLOW$ = OpticalFlowFarneback($R_{prev}$);
$MIN_DIST = \infty$ ;
$GT_{crt}$ =ignore ;
foreach Pixel P do
    $\Delta = FLOW(P)$;
    if $|\Delta| > 0$ then
        if $MIN_DIST(P_x + \Delta_x, P_y + \Delta_y) > |\Delta|$ then
            $MIN_DIST(P_x + \Delta_x, P_y + \Delta_y) = |\Delta|$ ;
            $GT_{crt}(P) = GT_{prev}(P + \Delta)$ ;
```

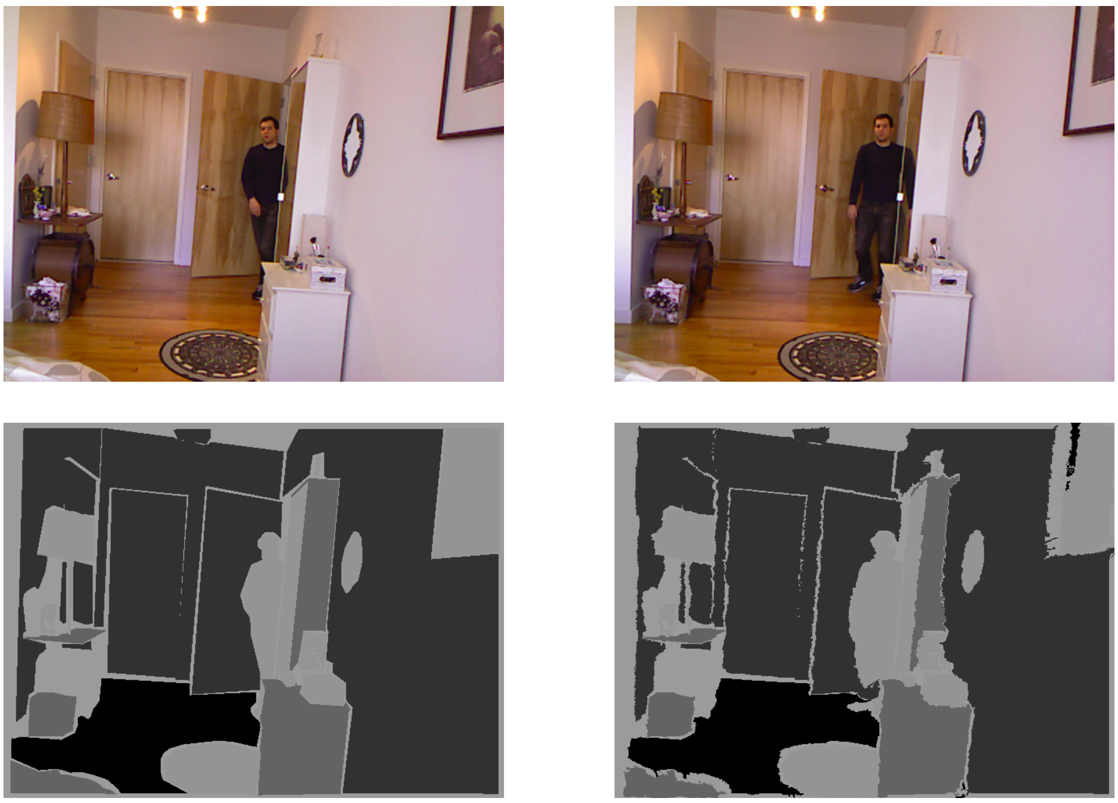


Figure 3.9: The two top images represent RGB frames taken with a small temporal interval in between. The two bottom images show the corresponding Ground Truth of the first frame, and the inferred ground truth of the second one respectively. Notice the right foot of the person causes an error, since it was obstructed in the first frame.

We also considered taking into account the depth when we compute the intermediate ground truth, but we visually noticed that the results did not improve. Two of the reasons are the error of depth measurements and the temporal delay between an RGB and a Depth measurement.

3.3.6. Sliding window

A common technique used in Computer Vision tasks is the use of sliding windows. This consists of running classifiers on image patches at different sizes. This allows the classifiers to learn concepts independent of the scale.

Inspired by the results obtained in Schulz et al., 2015 using such a technique, we integrate it in our learning models hoping this will enhance the class accuracy, especially on small objects. Another effect that we desire is to lower the overfit since this technique also augments in some sense our dataset, producing several samples out of a single one.

Schulz et al., 2015 suggest using the depth information to decide the size of the window, chosen it from a distribution that encourages a larger window size for regions closer to the camera and smaller windows for those that are far away. One reason behind this is the fact that the far away regions of input contain less detail, especially in the depth channels, thus there is no sense to process such regions densely. Also, instead of using a binary ignore mask, they use it as a set of weights that decrease radially (function r) starting from the center of the window as follows:

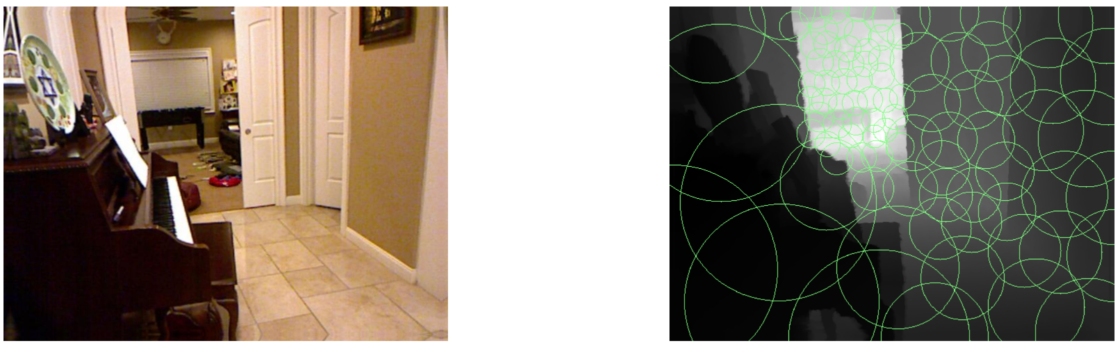


Figure 3.10: Original image (Left) and depth with sliding windows (Right). (Schulz et al., 2015)

$$w(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \text{ is not annotated} \\ 0 & \text{if } \mathbf{x} \text{ is outside the original image} \\ r(\|\mathbf{x} - \mathbf{x}_c\|)/p(c(\mathbf{x})) & \text{else} \end{cases} \tag{3.7}$$

where p(c(x)) is the prior probability of the class x is annotated with.

During the training phase we pick a small number of randomly chosen sliding windows from each image. During the test phase, we have to produce labels for the entire image, thus we cover the whole image with windows allowing them to overlap, predict on each window and combine the predictions together in the end.

3.4. Training

In this section, we discuss the factors involved in training and additional techniques that have the potential of improving the quality and speed of training are presented. In particular we describe different gradients, the loss function and several regularization methods used.

3.4.1. Gradient

Backpropagation together with the Stochastic Gradient Descent are widely used for updating the parameters in neural network models with a small number of layers. Equation (3.8) shows how a parameter θ is updated based on the partial derivative of the error function $\mathcal{E}$ and a fixed learning rate. However, in the context of deep learning and especially of recurrent neural networks, different other gradient types are more appropriate in the attempt to tackle the vanishing/exploding gradient behaviors.

$$\theta_{t+1} = \theta_t + \epsilon \cdot \frac{\partial \mathcal{E}_t}{\partial \theta_t} \tag{3.8}$$

Rmsprop

Rmsprop is the mini-batch version of the Resilient Propagation (Rprop) (Riedmiller and Braun, 1993). In Rprop, the weight updates no longer depend on the magnitude of the gradient, only on its sign, and the learning rate is adapted for each parameter individually based on the sign changes of the gradient. If the sign remains the same after one update, the update value of the parameter is slightly increased to speed up the convergence. If, on the other hand, the sign differs from the previous one, this means that the minimum was missed and the previous update is reverted, decreasing the update value at the same time.

As Rprop was designed to be used with full-batch learning, in order for it to work with mini-batch learning an adaptation is needed. The reason why regular Rprop does not work when mini-batches are used, is that there are frequent sign changes due to the variety of directions the mini-batches might point to. According to the Rprop algorithm previously described, such a situation causes the update to progressively shrink, leading to a state in which the network is unable of learning further.

To cope with this issue, Rmsprop keeps a moving average of squared gradient, as shown in Equation (3.9), using the value of the gradient divided by the square root of this average as update-value. This leads to treating the mini-batches differently, based on the gradient, but at the same time keeping to some extent the proprieties of Rprop by decreasing the influence the magnitude of the gradient has over the updates.

$$MS(\theta, t) = 0.9 \cdot MS(\theta, t-1) + 0.1 \cdot \frac{\partial \mathcal{E}_t}{\partial \theta} \tag{3.9}$$

Gradient Clipping

Gradient clipping is a technique counteracting the exploding gradient issue, by clipping the norm of the gradient when it exceeds a certain threshold. While this would allow us to train large networks using the plain gradient, we can apply gradient clipping in association with other gradient methods as well.

The effects of this technique on the learning process are analyzed in Pascanu et al., 2013. They describe the explosion of the gradients as the encounter, on the error surface, of a steep wall perpendicular to the gradient, forcing a large step in the opposite direction. This is not desirable especially if we are close to a minimum. Clipping of a such large step causes a move near the wall, to a smoother region of the error surface having the chance to explore the other directions, that were outweighed in the previous step.

3.4.2. Loss Function

In the task of object-class segmentation, our goal is to classify each pixel in one of the available m classes. This makes an one-out-of-m coding desirable for our learning model, with the output consisting of a vector v of size m and only the entry corresponding to the class of the input (in our case of the pixel) is set to 1, keeping the rest of the entries 0. Given the m outputs (o) returned by the model, the softmax function can be used to transform these outputs in a likelihood:

$$P(y = j|o) = \frac{e^{o_j}}{\sum_{i=1}^{m} e^{o_i}} \tag{3.10}$$

for which, given m teacher values ($\hat{o}$), we want to reduce the probability of a wrong assignment to zero and to increase the probability of a correct assignment to one. This is done using the multinomial logistic loss:

$$\mathcal{E}(o, \hat{o}) = \sum_{k=1}^{|o|} o_k \cdot \hat{o}_k - ln \sum_{k=1}^{|o|} e^{o_k} \tag{3.11}$$

which is averaged over all the inputs to produce the global loss function. In our particular case, the images exhibit regions for which we do not have a labeling and thus we are interested in ignoring those. We do this by applying an ignore mask (Figure 3.11) to both the teacher and the output before computing the loss.

3.4.3. Regularization

In the process of model selection, regularization has the role of controlling the complexity of the model, penalizing too complex models in order to prevent over-fitting.

Early stopping

Early stopping is such a regularization method, imposing a termination criterion based on the progression of the error on the validation set. The fact that although the error on the training set is decreasing, the error on the validation has ended its decreasing trend, or even worse, starts increasing, is one of the signs that the

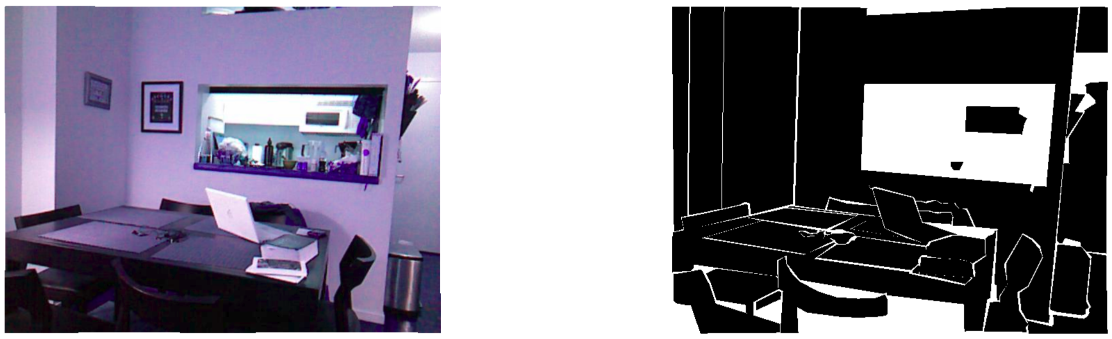


Figure 3.11: Original image (Left) and corresponding Ignore mask (Right). White shows the regions to ignore

model starts over-fitting. The early stopping terminates the training process once the validation error is no longer decreasing. To make sure that the termination occurs at the right moment, one can monitor instead a moving average of the validation error and use it to decide when the decreasing tendency is really over.

Weight decay

Weight decay, also called L2 regularization, is a technique used to penalize large weights, resulting in a simplification of the model. It consists in adding a fraction λ of the 2-norm of the weight vector w to the loss function $\mathcal{E}$. The following equation shows how the new loss function $\mathcal{E}^*$ is computed:

$$\mathcal{E}^* = \mathcal{E} + \sqrt{\sum_{k=1}^{|w|} |w_k|^2} \tag{3.12}$$

Dropout

Another way of counteracting overfitting is to combine together several different models and average their prediction. While the direct implementation of this concept for large neural networks is expensive, a simple way to simulate this is the technique of dropout, introduced in Hinton et al., 2012.

When dropout is used, a certain proportion p of the activations, chosen randomly each time, are set to zero (dropped). This is equivalent to dropping the corresponding neural units and thus to having a different network in every traversal. Moreover, the total number of possible generated networks is exponential. For example, when dropout is applied to N neurons, using a ratio $p = 0.5$, in each traversal we are equally likely to have one of the 2^N possible networks.

As suggested in Pham et al., 2014, one should be careful when using dropout on RNNs, not to affect the recurrent connections. Doing this might perturb the ability of the network to learn long-range dependencies.

4. Recurrent Convolutional Networks

In this chapter we describe the main contribution included in this work. Three different recurrent neural networks are implemented based on an already existent convolutional model. We further discuss implementation details and show how our modules interact with other modules from the two libraries that we use: CUV and CUVNET.

4.1. Architectures

In this section we describe the network architectures used in this work, all inspired from the Neural Abstraction Pyramid described in the Section 2.0.1. We first present the common proprieties in the proposed architectures.

Between the layers of consecutive copies, three types of connections exist: forward, backward and lateral.

One layer is connected to the next one of the next temporal copy through a forward connection. These connections allow the vertical flow of activation from the bottom of the network to the top of it and thus to the construction of high level features based on the low level ones. Backward connections on the other hand, connect a layer to the previous one, allowing the construction of the features by taking into account the higher level ones. Lateral connections allow the horizontal flow of activations, connecting layers residing on the same level of abstraction. The benefit from using such horizontal connections is that the features learn about the features from their neighborhood, which is especially useful in Computer Vision tasks due to the local spatial dependencies. In order to implement the horizontal connections efficiently, we use a neural operator for upscaling the output of the convolution and we feed the upscaled version as input for the same layer of the next time step.

The output of the network is always obtained in the lowest layer of the network, after we make sure that the activations were able to reach the highest layer and return back. Since the number of filters of the first layer should not be restricted

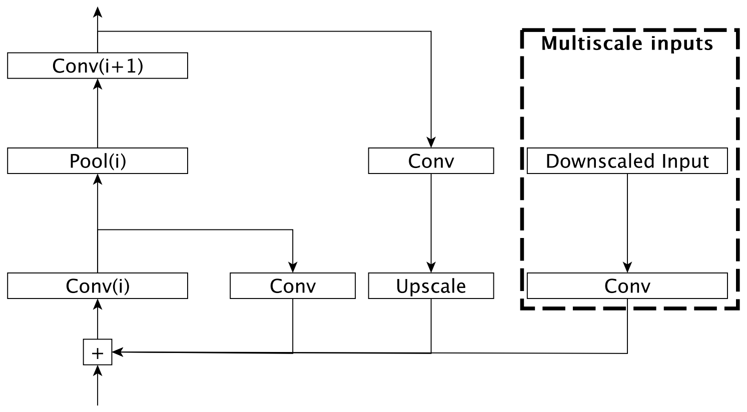


Figure 4.1: Connecting several layers together

to the number of classes C, we extract the first C channels of the convolution and we consider each of them responsible for one class.

Connecting together different layers requires aligning their filter sizes and their number of filters. The upscale operator solves the problem of aligning the filter sizes, while another convolution operation aligns the number of filters. A final addition combines the filters together. In case we use multiple scales for the inputs, we also bundle inside the summation a scaled down version of the input. An overview is showed in Figure 4.1.

A difference between our design and the Neural Abstraction Pyramid is that we keep the number of filters constant for all layer of the network, while in the pyramid the number of filters doubles as the resolution of the representation is halved. Initial experiments on our dataset have shown that doubling the number of filters leads to a too complex model, taking longer to be trained and overfitting early.

4.1.1. Simplified RNN

In the first architecture we concatenate all inputs in one map and feed it to the network. This is a simplified version of a recurrent network, since the capability of exploiting the temporal dependencies is limited. On the other hand, spatial dependencies can still be exploited since we keep the pyramid structure of the network. Thus, comparing the results obtained by this network against the other ones can help us understand what is the improvement brought by accumulating information over several time-steps.

As the first temporal copy we use a simple feed forward one, making sure that we

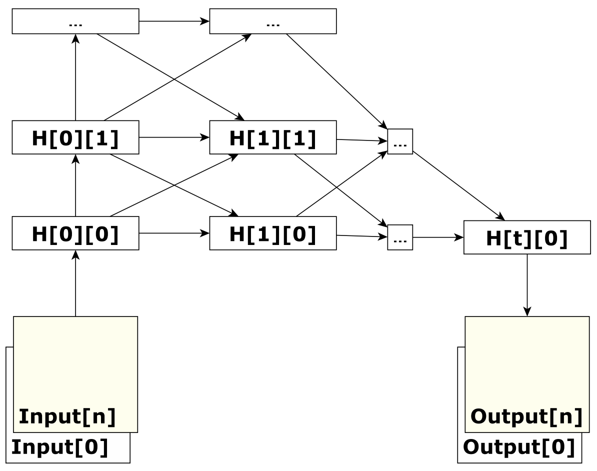


Figure 4.2: Architecture of the Simplified RNN

have at each layer the activations needed for the next step. Thus we can consider that in the first time step the activations are already propagated up to the highest level of the network. For a depth of N hidden layers, we need $N-1$ additional time steps for the signal to propagate downwards until the lowest level. As Figure 4.2 shows, the last temporal steps do not need all the hidden layers, since their output would no longer propagate anyway.

4.1.2. Unidirectional RNN

In the second architecture, we feed the images as a sequence, one in each time step. The state of the network represents the information derived from the past context that, together with the result of the convolutional model, produces an output and a new state.

Since the last output benefits from learning from the whole sequence, it is natural to place the frame that we want to evaluate at the end.

The first temporal copy of the network is a regular feed forward one. We organize it differently than the others since we want to have activations in each layer such that all connection types are used in the transition between the first time step and the second.

As in the previous network architecture, from the moment we feed one input, it is necessary to wait several time steps to allow its propagation in the entire network. This means $N-1$ time steps for reaching the top level of the network and $N-1$ time steps for propagating back at the bottom layer where the output is computed.

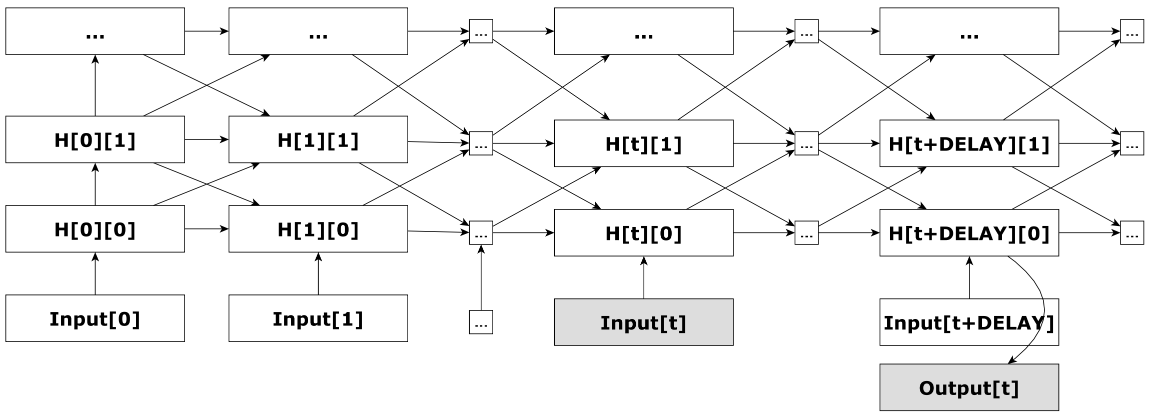


Figure 4.3: Architecture of the Unidirectional RNN

4.1.3. Bidirectional RNN

The third architecture is inspired by the bidirectional recurrent network. Here, connections exist in both directions of the temporal dimension. This network is able to exploit not only the past context, but also the future context. In real-time, of course, this introduces an amount of overhead due to waiting for all the future frames to become available in order to produce the result of the current one. Our implementation consists of two unidirectional networks whose outputs we combine in order to obtain the final outputs of the Bidirectional RNN. In this architecture, it is optimal to place the frame which we want to evaluate in the middle, unlike the unidirectional architecture where we place it at the end.

As an optimization, we do not have to use two full replicas of the unidirectional

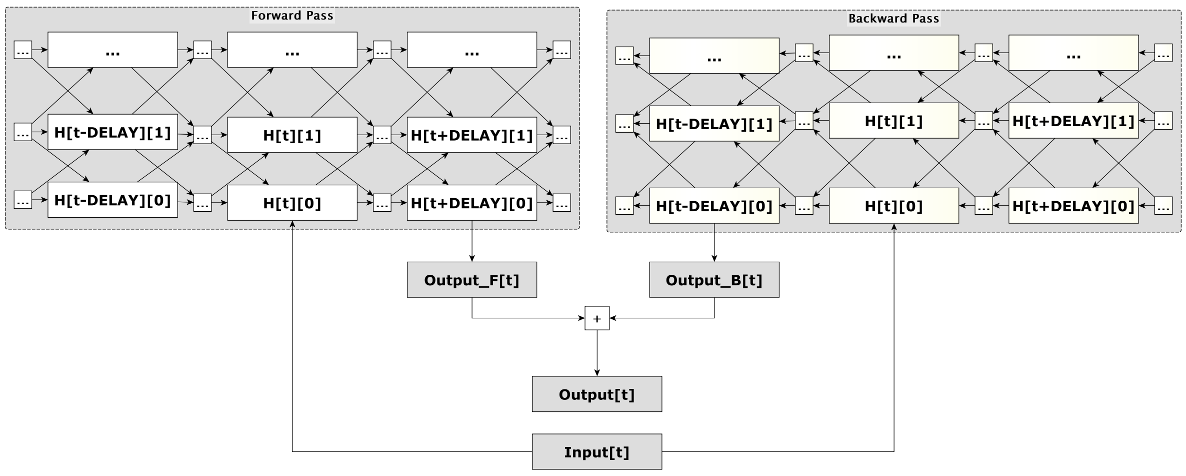


Figure 4.4: Architecture of the Bidirectional RNN

network, only about half of it, until we have the output of the middle time step. This significantly reduces the amount of GPU memory involved.

4.1.4. Extra network features

In addition to the presented architectures we also investigate some extensions, that could improve the performance of our networks.

Unshared biases

While in a RNN we normally share both the biases and the weight set, it might prove helpful for the network to be able to treat time steps differently. We do this by not sharing the biases between time steps, allowing the network to store in them information associated to the time step. This also has the effect of increasing the number of parameters of the network and thus its complexity.

Encoding the output

As mentioned before and as suggested in the network diagrams, the first layer will provide the output of the network. Since the number of maps of the first layer might be different than the number of classes N, one approach is to encode the output in the first N maps.

Another possibility is to encode the output in the entire first layer and to use an additional convolution to decode the N classes. This adds complexity to the network but at the same allows more freedom in representing the output.

Intermediate ground truths

While the network is able to learn temporal and spatial correlations from only one ground truth corresponding to the last time step, in order to reinforce the signal we can also evaluate ground truths at intermediate time steps. This approach is also used in the GoogLeNet (Szegedy et al., 2014), where 9 inception layers are used and the ground truth is evaluated three times along the network.

We do not wish to force an exact representation of the output at an early stage, since this would limit the representation power of the network. Thus, in the loss function, the intermediate evaluations will weight less.

If we choose to have a final convolution that decodes the output from the first layer, it might be useful to keep the parameters of these convolutions not shared, such that we encourage different representations at different steps of times.

4.2. Implementation of RNN on CUDA

In this section we describe the implementation of our learning models in the CUDA framework. We also briefly describe the two library used in this work: CUV and CUVNET.

4.2.1. CUDA framework

Neural networks, especially deep or recurrent ones, are an expensive model in terms of computing power and memory consumption. Thus, we are interested in using scalable algorithms that can be run in parallel on several processing units.

Central Processing Units (CPU) have a limited number of cores, being optimized for sequential processing. Graphical Processing Units (GPU), on the other hand, posses thousands of small cores, being optimized for running a large number of tasks in parallel. We use a mix of both GPU and CPU, running the intensive computations on images, activations and gradients as matrix operations on the GPU and the sequential tasks on the CPU.

CUDA is a general purpose parallel computing architecture from NVIDIA aiming to simplify the development of scalable tools that exploit the parallel architecture of NVIDIA graphic cards. In this framework, as a convention, CPU is named "host" and GPU is named "device". The main CUDA entities are: blocks, wraps, thread. The code to be run in parallel is defined in routines named kernels.

Block

A block contains several threads and is executed by one multiprocessing unit. Blocks can be represented in one, two or three dimensional grids, depending on the input dimension. For example, in images, a two dimensional grid leads to a direct mapping between the patch of the image being processed and the position in the grid of the block responsible for the patch.

Thread

The threads are running the same code, but can follow different branches. The threads inside a block can coordinate through shared memory.

Warps

One block is splitted into several units called warps. All threads in one warp execute the same instruction at a given moment. Due to this, branch divergence of threads in a warp can cause performance degradation.

Kernels

A kernel is called by the host and is ran by the threads of the device. Each kernel instance receives the block and thread ID on which is run. Based on them it determines what part of the input should it process.

4.2.2. CUV library

CUV is a C++ and Python library which, among other functionalities, handles operations on tensors and matrices in a distributed way, transparent to the developer.

In addition, CUV includes functionality for some of the CUVNET operators. The convolution and pooling are examples of such operators which have a corresponding kernel in CUV, allowing them to achieve parallel efficiency.

4.2.3. CUVNET library

The CUVNET library is built on top of CUV. It defines the neural operators and learning algorithms. We extend CUVNET by adding two new operators: upscale and gradient clip operator. Also we adapt the convolutional model to allow weight sharing between convolutional layers of the same level of abstraction but part of different temporal copies.

Upscale operator

We extend CUVNET and CUV by adding another operation and respectively a new kernel needed by the network, the upscaling. We use upscaling to counteract the resolution decrease of the input representation while it traverses the network, so that we can reuse it in the lower levels. This operation does simply replicates an activation on a $FACTOR \times FACTOR$ patch of the upscaled representation.

Although, normally, the implementation of such an operation does not represent a challenge, our task is to implement it efficiently, as a CUDA kernel, exploiting the fact that it can be split in smaller, independent sub-tasks.

Also, a neural operator, will generally be involved in propagating the gradients backwards during the back propagation phase as well. Exceptions to this rule are the operators from which the multinomial logistic loss operator is not reachable, thus they have noting to propagate (e.g. classification loss). Other exceptions are the operators that either have no inputs, either none of its inputs requires a derivative (e.g. input layer). An existent implementation of upscaling from

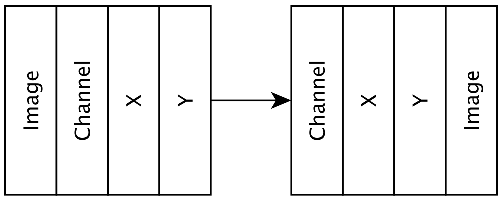


Figure 4.5: reorder_for_conv routine

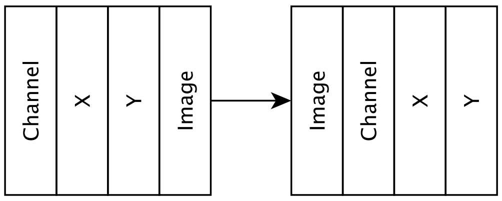


Figure 4.6: reorder_from_conv routine

cuda-convnet uses bilinear filtering to resize the image, but implements no back-propagation functionality, restricting the user to incorporate such operator in the data layer.

In our network, the upscaling operation is applied between time-steps and thus the backward propagation functionality is mandatory. Although we need only the operation of upscaling by a factor of two, we implement a more general upscaling operation that can upscale by an integer factor. Moreover, the operator can easily be adapted to support even different factors for the two dimensions. While we work with rectangular input, the operator was implemented and tested to work on non-rectangular input as well.

One technical detail which helps the implementation of such an operator efficiently is the shape of the input tensor. The first dimension stores the channels of the image, the second is the horizontal axis of the image, the third is the vertical one and the fourth is storing the image number. This, although not a natural representation of a batch of images, allows us to have access to the pixel of the same location, from the same channel, of different images on a contiguous memory space, and since the control flow does not depend on the value of the pixel but only on its location, all thread of a warp will execute the same instruction at each point of time, avoiding branch divergence. Other kernels such as convolution or pooling exploit this structural property as well, thus it is useful to keep the same

Operation	Time (s)	Speedup
Forward Pass, CPU	0.0468	
Forward Pass, GPU	0.0015	29.7
Backward Pass, CPU	0.0635	
Backward Pass, GPU	0.0019	33.4

Table 4.1: Time measurement of the upscale operation: GPU vs. CPU (single thread). As input we used 16 samples of size 80x80, each having 16 channels. The measurements are averaged over 100 runs.

tensor shape in the whole network. Since the natural shape of the data is: image number, channel, X coordinate, Y coordinate we use two routines implemented in CUV for this purpose: reorder_for_conv and reorder_from_conv which handle switching between those two orderings

During the backpropagation step, we do the reverse operation of the upscaling: all gradients inside the patch are summed and passed back as one gradient.

We test our upscale operator in three ways. We check the correctness of the CUDA implementation by implementing the forward and backward pass in CPU as well, testing several situations and corner cases on both architectures. We expect a small difference between the two outputs caused by the two architectures performing float operations and rounding differently, but we constrain this error to be small enough, not to influence the results.

This test, however assumes that the gradient calculation is correct. For this we need an additional derivative tester that computes the Jacobian using the finite difference approximation and compares it to the Jacobian we obtained using our operator. This test along with the first one confirmed that the upscale operator works correctly.

A final objective is the parallel efficiency. We compute the speedup of the GPU implementation over the CPU one. Table 4.1 shows the time measurements for handling input tensors whose shape is one that frequently occurs in our network. Of course, running GPU code involves an overhead, so processing small tensors, such as 16x3x8x8 will lead to a slower processing on GPU than on CPU.

Gradient clipping operator

In addition to the theoretical description of this operation, provided in Section 3.4.1, we will further describe also the challenges faced when implementing it.

Gradient clipping is one of the operations active only in the backward pass, thus in the forward pass the input is propagated unchanged. In the backward pass however we require all gradients to be available in order to be able to compute the norm, which is further required to clip, where needed, the gradients. Thus, our operator will receive as input all the parameters of the network and have one output for each. All of the neural operators in CUVNET are executed in topological order.

Since we share weights between temporal copies, the gradients are accumulated over all copies. Once all gradients are available, the operator computes the norm and clips the gradients larger than the threshold.

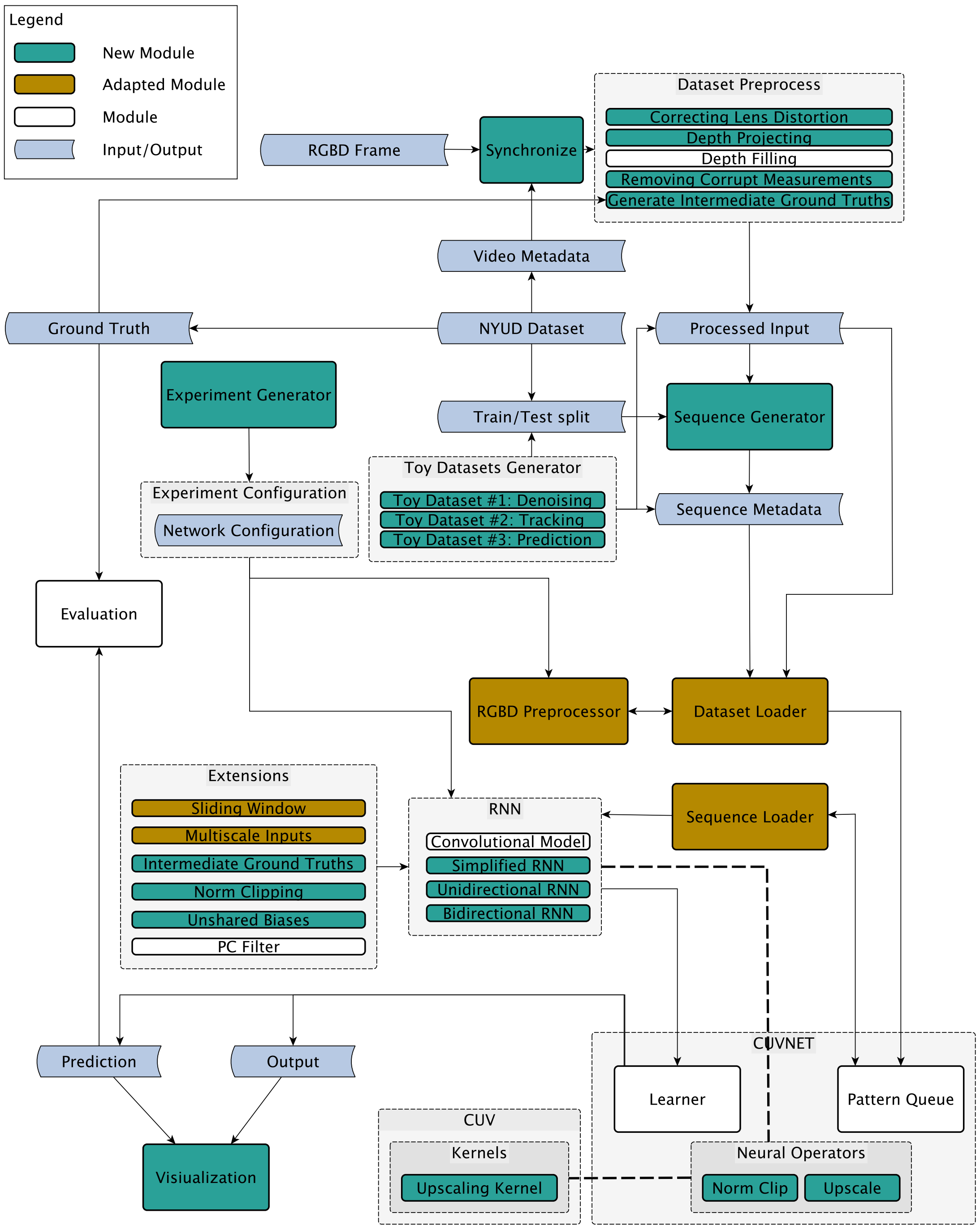


Figure 4.7: An overview of the whole system, showing the modules involved in different stages of the process.

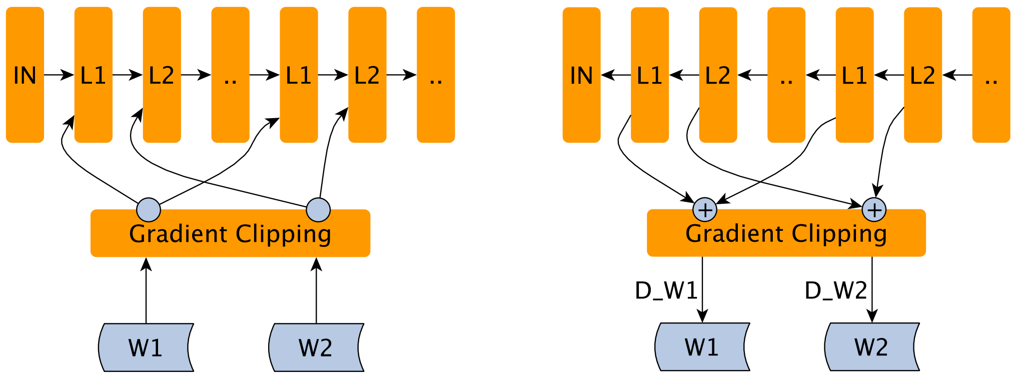


Figure 4.8: Gradient Clipping operator: Forward pass **(Left)**, Backward pass **(Right)**

4.2.4. Integration with CUV and CUVNET

We now discuss the high level design and how the networks and the developed modules interact with each other and with other modules. Figure 4.7 Depicts this interaction. The Dataset Loader module reads the sequence dataset from a meta-data file. Each image of the dataset is processed asynchronously by a worker of the thread pool and once the image is processed, it is added to a queue as part of the sequence.

Once a sequence is fully processed, it can be extracted by the Sequence Loader which further inserts every input in the right place of the chosen network architecture. A Learner determines the topological order in which the operators should be executed, and runs gradient descent based on this ordering.

5. Experiments

In this chapter we first show that our network architectures work correctly and are able to learn temporal dependencies. To do this, we run several experiments on toy tasks that would require the learning model to accumulate information over time for correctly solving them.

We then train our models on a real life dataset and present the network features that improved the learning process. We also compare the results with those of a simple convolutional model, and with other models as well.

5.1. Toy Experiments

In this section we present three toy experiments that prove the capabilities of our network to learn from sequences, accumulating information over time.

5.1.1. Denoising a static object

In this experiment we simply feed to the network different noised versions of the same binary image. We use binary noise, namely salt and pepper noise, uniformly distributed over the whole image. We also draw random black or white stripes, to make the task more difficult.

To make sure that the network is able to generalize instead of learning an object by heart, we use different objects for the train, validation and test sets.

The task consists in obtaining the original, denoised image. One way the network could solve this task would be to learn to average the images over time. Of course this can be combined with the power of the convolutional model, which might also learn denoising filters.

Setup

Since the task has a reduced complexity, we opt for a simple convolutional model of only one hidden layer with 32 maps. Since we do not want the network to rely entirely on the filters learned by the convolutional model, we keep those at a size

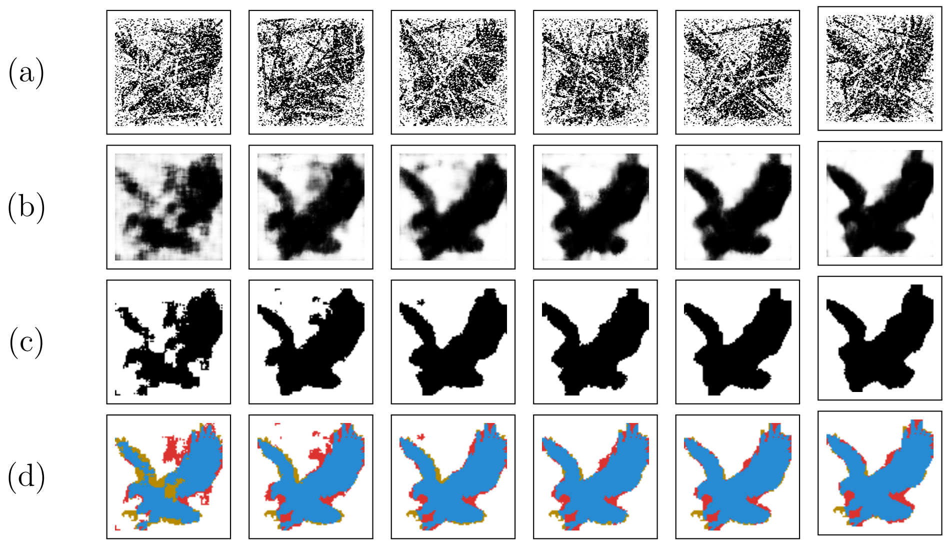


Figure 5.1: Outputs for Toy Experiment #1. Row (a) shows the input of the network for each time-step. Row (b) shows the output of the softmax layer. Row (c) shows the final outputs of the network. Row (d) shows the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives). The last output represents the final output of the network and we use it for evaluation.

of 5x5 pixels. Since there is no specific order in such a sequence of noised images, we only test one of our architectures on this task, the unidirectional.

Two classes will be learned: the object and the environment. This way we discourage both the false positives and the false negatives.

We use six temporal copies, and thus feed six different noised versions of the image. We evaluate each step and do a weighted sum of all losses. Since the last output is the output of the network, we emphasize on it by setting the weight of

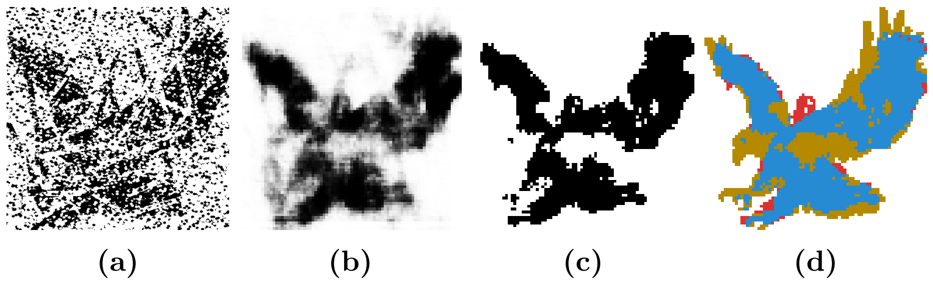


Figure 5.2: Outputs for first toy experiment, non recurrent model. Figure (a) shows the input of the network. Figure (b) shows the output of the softmax layer. Figure (c) shows the final output of the network. Figure (d) shows the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives).

Model	Pixel-Wise Accuracy (%)	Class Accuracy
Simple CNN (1 layer)	84.4	84.7
Recurrent CNN (1 layer, 6 time steps)	93.0	93.2

Table 5.1: Results obtained on Toy Experiment #1

the corresponding loss higher by a factor of ten.

We also test the simple convolutional model, with no recurrent connections such that we can investigate whether the recurrent structure of the network makes actually a difference. We train both models for 600 epochs.

Results

Figures 5.1 and 5.2 show the results we obtain using a recurrent network and a non-recurrent network respectively on the test set. We can see that the best that the convolutional model can do on its own is to reduce the noise in areas where despite the noise, the structure of the object is still distinguishable. On areas such as the center of the image, where the object presents a more complex structure with more details, the noise has a substantial negative effect, difficult to be managed by such a simple model.

The recurrent model, on the other hand, is able to improve its prediction step by step, accumulating over time information even from the areas which are more sensitive to noise. After only two steps the network is able to solve most of the false positives and to assemble together almost all features of the object.

Seeing that the convolutional model on its own is unable to reproduce the results of the recurrent model, a natural question is whether the network can solve this task without learning any features at all, simply by averaging the inputs as discussed previously? Figure 5.3 shows that even after averaging together all six inputs, we still end up with a noisy image. The structure becomes , however, more clear and the noise is significantly reduced.

We showed how this task can be solved using a simple convolutional model for extracting features together with recurrent connections that propagate those

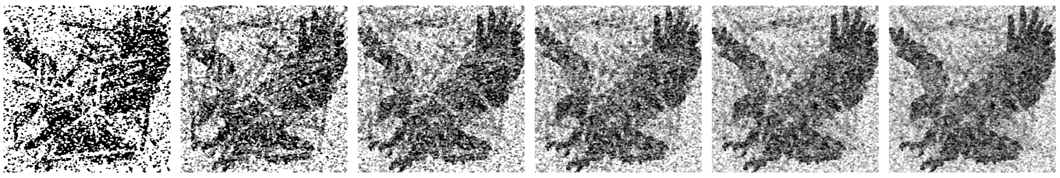


Figure 5.3: Moving average on the inputs

features to the next time step. Further we will investigate two tasks of higher difficulty, where the RNN will be even more useful.

5.1.2. Learning a trajectory

In this experiment we test the capabilities of the network to track an object while moving. We move the object while the whole image is noised. One could argue in this case that the object position might be detected by the network, considering the density of each blob, since we have one object on a large canvas, and the noise blobs are significantly smaller than the object. Not wanting our network to rely on such details, we add two additional dummy objects, identical to the one that we want to track but at random positions in each frame.

To prevent the network from overfitting on only one type of movement and only one speed, we generate several subsequences, each moving the object from a random position to another, slightly variating the speed.

Task

The goal is similar to the one of the previous task, the only change is that we want to denoise a moving object, and only the one having a continuous trajectory, ignoring the other identical objects which move randomly from one frame to another. The network could also solve this task by learning to average over time but, in addition to this, the network would also have to learn a transposition.

Setup

In this experiment we test all three network architectures. There is no point to test the convolutional model alone here, since there is not enough information in one frame to distinguish between the aliases and the moving object.

Results

Figure 5.4 shows the results obtained on this task using an unidirectional network. As we expect, in the first time step, the network can not decide which object is moving continuously. Already in time step two the network detects a slight positional change from one of the objects, while the others are further away from their initial position. The softmax layer output shows that the certainty of the hypothesis increases step by step. Also, one can notice that more features are added, literally on-the-fly, to the representation. Some false positives still exist when the new random position of an alias object is close to its former position.

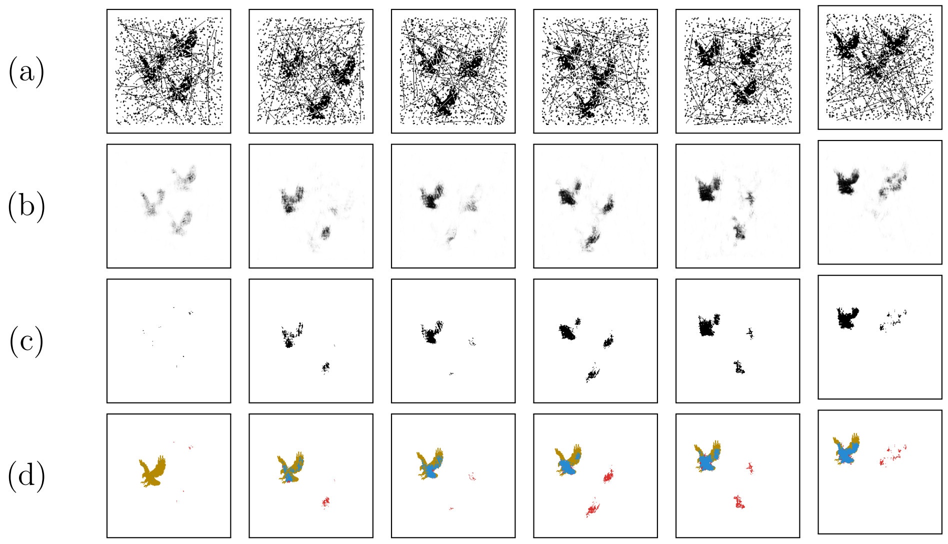


Figure 5.4: Outputs for Toy Experiments #2. Row (a) shows the input of the network for each time-step. Row (b) shows the output of the softmax layer. Row (c) shows the final outputs of the network. Row (d) shows the evaluation (🔵 True Positives ○ True Negatives 🔴 False Positives 🟡 False Negatives). The last output represents the final output of the network and we use it for evaluation.

Table 5.2 shows the results we obtained on the test set of the toy example #2. Since we have a large class, covering more than 90% of the image, we can expect a large pixel-wise accuracy for any classifier. Thus, we should focus on the class-accuracy, which weights the two classes the same.

The simplified network obtains the worst results among the three tested models. As previously stated, since the whole sequence is placed in one map, the network has limited ability in detecting long range dependencies. In current scenario, this dependency is very strong and it is important to learn it for the optimally solving the task. The best thing such a network can do is to compute some statistics based on several input frames, but it is not able to keep and update a state information.

The other two more complex models obtain significantly better results, improving the class accuracy by about 15%. Although the result obtained using a

Model	Pixel-Wise Accuracy (%)	Class Accuracy (%)
Simplified RNN	96.3	72.4
Unidirectional RNN	97.7	87.3
Bidirectional RNN	98.1	89.2

Table 5.2: Results obtained on Toy Experiment #2

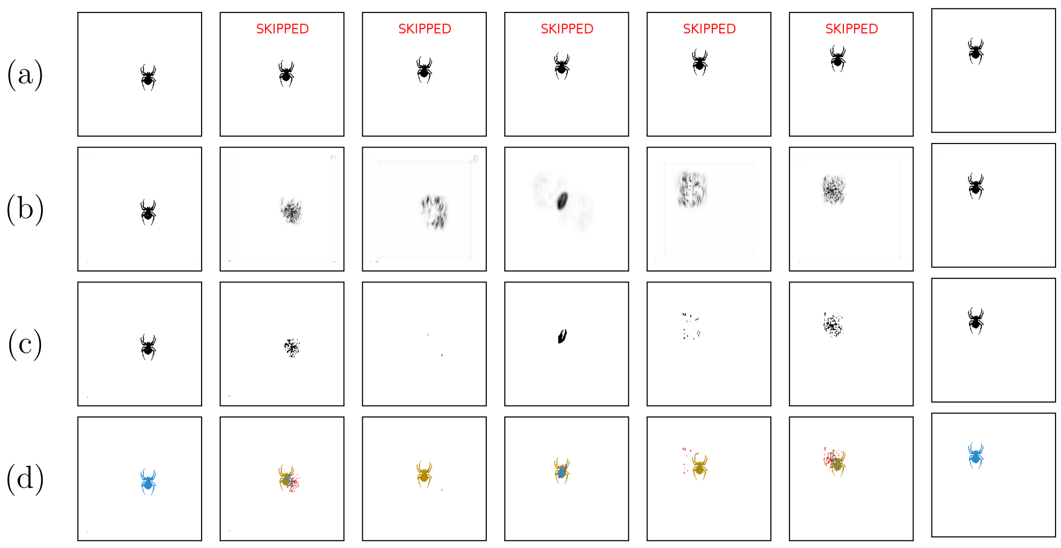


Figure 5.5: Outputs for Toy Experiment #3. Row (a) shows the input of the network for each time-step. Row (b) shows the output of the softmax layer. Row (c) shows the final outputs of the network. Row (d) shows the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives). The middle output represents the final output of the network and we use it for evaluation.

bidirectional network is only 2% better than in case of the unidirectional one, this shows that keeping the same context, but processing the input in two directions is, in this scenario, beneficial.

5.1.3. Trajectory Inference

If in the previous experiment we convinced ourselves that our recurrent network is able to track a moving object. We are now interested if it is actually capable of inferring such a regular movement based on partial information.

To check this we will no longer provide the last inputs to the network, but we will expect the network to output the moving object at the right position, in the evaluated time-step. There is no longer need to apply noise or add any aliases to the inputs, since the network can not rely only on the information existent in current frame. If it would do so, it would not be able to generate a prediction for time steps where input is missing.

Task

In the first time steps the task of the network is simple: It only has to reproduce the corresponding input. However, since the last few inputs are missing, the network should predict the position of the object based on the information gained in previous time-steps. Since averaging over time is not an option here, the network might learn the transposition and simply propagate the first inputs further, since no noise is present anyway.

Another challenge is for the network to learn to ignore the non-existing input and thus to not decrease the intensity of the signal. This is problematic since the network does not know its current position along the temporal axis.

Since the task is no longer to denoise an object, and we are rather interested in its movement, we use a less complex moving object for this task.

Setup

For solving this toy task, we will use both unidirectional and bidirectional networks, which we expect to have predicting capabilities. In our experiments we investigate what happens when two or three out of our seven inputs are not presented to the unidirectional network. In case of the bidirectional network, in order to have equivalence to the unidirectional network experiments, we skip three and five inputs. The reasoning behind this is that the sub-network that manages one of the directions receives half of the inputs (plus the middle one which is present in both directions). For example, skipping three inputs means that both sub-networks will skip two inputs.

Result

As in the last task, the class accuracy metric gives a better overview since the two classes are unbalanced.

Table 5.3 presents the results obtained on the test set. We notice that even in the least performant configuration, where we have a bidirectional architecture and we only feed the first and the last input, the network is still able to track the object, without being able to reproduce all its features. Figure 5.5 depicts this scenario, showing how the network actually learn to infer the position in the

Model	Nr. Skipped Frames	Pixel-Wise Accuracy (%)	Class Accuracy (%)
Unidirectional RNN	2	97.0	80.1
Unidirectional RNN	3	96.6	75.4
Bidirectional RNN	3	97.2	81.4
Bidirectional RNN	5	96.6	74.9

Table 5.3: Results obtained on Toy Experiment #3

middle time step, where we expect the final output of the network.

5.2. Experiments on NYUD

The NYUD dataset is comprised of video sequences taken from 464 indoor scenes, adnotating a total of 894 classes. We reduce them to four high level abstractions:

Prop Small objects that can be easily carried

Furniture Large objects that cannot be easily carried

Structure Non-floor parts of the room: walls, ceiling, columns.

Ground The floor of the room.

Everything that falls out of these categories will be ignored using the ignore mask.

5.2.1. Memory limitation

One technical issue when working with recurrent or deep neural networks is the large amount of memory involved in the training phase. The reason is that some of the neural operators require remembering the old value of the activation in order to compute the gradient. In our network such operators are: convolution, max-pooling, hyperbolic tangent (tanh). This implies that we have to keep the activations in memory for all such operators, at the same time, until the final result of the network is computed. Once backpropagation reaches an operator, we

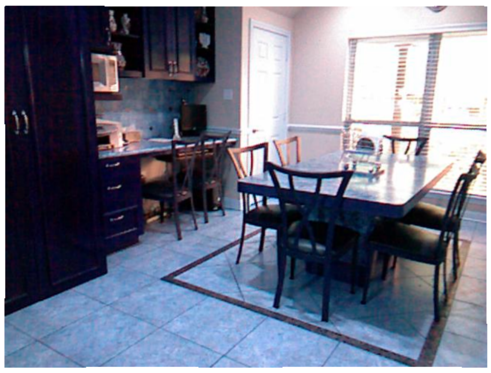


Figure 5.6: Original image

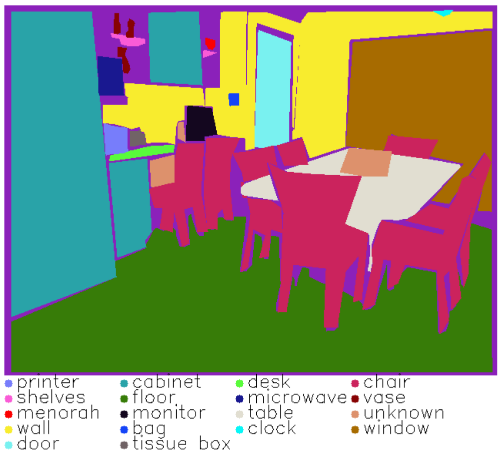


Figure 5.7: Labeled frame of NYUD

no longer need the previous activation and we can discard it. Thus, an analysis of the needed memory will not only tell us which configurations are feasible, but also give us an insight on how large is the effect of each structural property of the network on the memory consumption.

We show a formula that estimates the memory consumption in bytes

$$4b(xys + \sum_{i=1,n} \frac{m_i xyf}{2^{2i}}(s + 2(n-1) - i))$$

, where b represents the batch size, n the number of layers, x and y the dimension of the input, s the sequence size, m the number of filters for convolution i and f represents a number of operators per layer for which have to store the activation during forward propagation. f takes values between 3 and 5 depending on the position of the layer inside the network (layer number and time step).

To give a brief explanation for the formula, $m_i xy$ represents the memory we need to store for one operator. Both x and y are halved after each level of abstraction, which is reflected by the denominator. Each level of abstraction of the network has a total of s temporal copies plus an additional $2(n-1)$ to allow the signal to reach the highest level of abstraction and return to the lowest, where the output is constructed.

We present the estimations for several possible network architectures, such that the reader can get a sense on the restrictions such architectures face at the moment.

One must however keep in mind that this memory amount is needed only to store the activations, and additional memory is involved in the computation done by the operators. A bidirectional network, as well as extensions such as: final convolution, multiscale inputs require more memory.

We chose our configurations such that the memory requirement is fulfilled. We

Image Size	Number of Maps	Sequence Size	Batch Size	GPU Mem.(MB)
160	32-32-32	8	16	3581
160	32-32-32-32	8	16	4242
160	32-64-128	8	16	4875
160	32-32-32	16	16	6106
160	32-32-32	8	32	7162
600	32-32-32	8	16	50361

Table 5.4: GPU Memory Consumption estimation for several potential configurations

use NVIDIA GeForce GTX Titan graphic cards, having an internal memory of 6144 MB. We could relax this limitation by caching the activation, saving them in RAM after they are computed, and loading them back to GPU when we back-propagate. However this would result in a large amount of memory being transferred twice between GPU memory and RAM, during every traversal of the neural network, slowing down the whole training process.

5.2.2. Kernel size limitation

When first running the experiments for image sizes of 160x160, we noticed that the simplified architecture was failing, while other were running fine. We localized the issue in the convolution kernel. Due to the large number of channels, resulted by concatenating all inputs in one map, the convolution operation was calling kernels of a size larger than the maximum manageable by our GPUs. This limit is in our case of 1024 threads per block.

In order to comply to this restriction we could simply run the experiments that use the first architecture on a down-scaled input of size 80x80. This however would cause an important disadvantage compared to the other two architectures where such a situation does not occur.

The solution we implement is first to detect such situations and to split the convolution in two, each responsible for half of the channels. We then concatenate the results. The overhead introduced by this solution is significant, consisting of a convolution and a concatenate operator. On the other hand, we only have to do

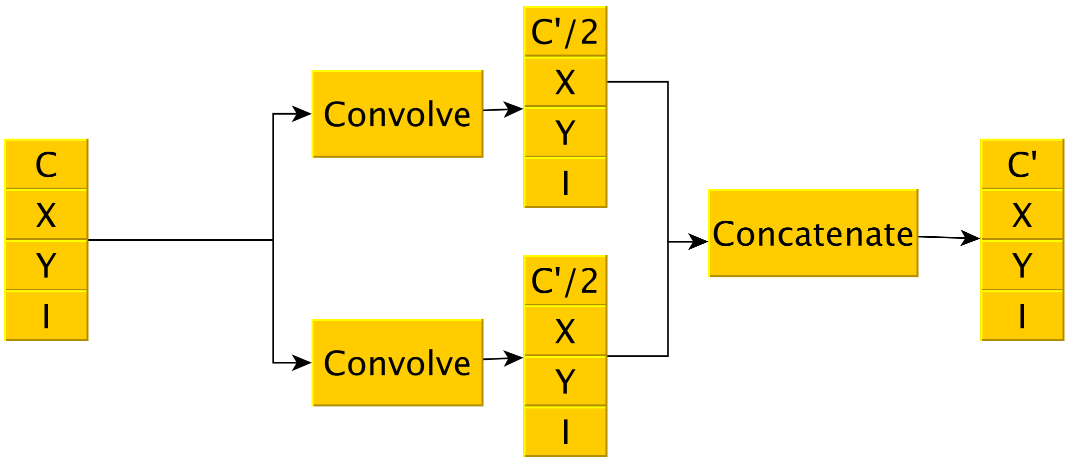


Figure 5.8: Splitting convolution operation in order to prevent exceeding the maximum threads per block limit

this for the first layer, since due to the resolution decrease, the kernel size will also decrease under the maximum limit.

5.2.3. Creating sequences

Although NYUD was recorded as a video sequence, the actual dataset consists of a subset of 1449 frames which were preprocessed and manually labeled. The rest of 407,024 frames are the raw output of the RGB and Depth cameras from the Microsoft Kinect.

In order to be able to transform the dataset into a image sequence dataset, but at the same time use the labeled frames for evaluation, we have to extract the past and future context of each labeled frame and preprocess it. The network, depending on the architecture, will receive half of the context or the whole context and we will compare the output corresponding to the labeled frame with the ground truth.

We need to keep the temporal distance between frames short enough, to ensure that the translation stays within the 7x7 filter size, such that each abstraction level of the network can track the changes between two frames. Although the speed of the camera movement can vary, a fixed interval of 0.1 seconds between frames allows us to have a smooth transition, but also to cover a significant time-span. During extracting the context it is necessary to synchronize the RGB and Depth frames, not captured at the same by the Microsoft Kinect since the RGB and the Depth sensors work independently of each other.

The next step is the preprocessing of the RGB and Depth images. Preprocessing involves: lens correction, projecting the depth onto the RGB sensor perspective and filling the depth.

Lens correction attempts to fix the barrel distortion typical to wide angle cameras and is done using the "Camera Calibration and 3D Reconstruction" package of OpenCV and the camera parameters provided by the creators of the dataset.

Also, the fact that the RGB camera and the Depth camera have slightly different positions should not be neglected. The depth must be projected from the depth camera viewport to the RGB camera viewport. This is done by a rotation and a translation using the matrices provided by the creators of the dataset.

The final step is to fill the depth. The Kinect cameras produce depth images where often significant parts are missing. The producers suggest that this phenomenon is "due to certain materials or scene structures which do not reflect infra-red (IR) light, very thin structures or surfaces at glancing incidence angles". (Newcombe et al., 2011).

While we could simply ignore the depth holes, the network would not benefit from the depth information in those area, or even worse, might misinterpret them. Thus, we fill the depth using the colorization algorithm(Levin et al., 2004). We use the already existing port of the algorithm in Python and C++ from **waldvogel13** and Schwarz, 2014 respectively.

5.2.4. Dataset Issues

Among the unprocessed frames, both RGB and Depth, of the NYUD, corrupt or partially corrupt measurements exist. If the number of such problematic samples in the training set is small, it would not influence learning significantly since the gradient is computed over the whole mini-batch. However, if such samples occur in the test set, they will affect the final results, since the accuracy for classifying one sample has a weight of 0.16% in the average accuracy that we use for comparing models. While ignoring those frames when building the dataset is an easy task, detecting them is more involved since we can not inspect manually every frame of the dataset. Instead, we use an heuristic to detect some of those corrupted frames.

Another problem is caused by one of the sensor being inactive for several seconds. This is a major issue for us since we expect one frame every 0.1 seconds, due to the reasons explained in the previous subsection. There are also situations when the context simply does not exist, for example if one of the labeled frames is at the beginning or at the end of the video sequence. In such situations we are not able to produce a prediction. When we use a unidirectional network and only the past context is partially unavailable we can process the frames in the reverse order, this workaround allowing us to produce a prediction. On the other hand, for bidirectional networks this is not an option. In practical situations,when such cases occur, we should posses and use an additional non-recurrent CNN to provide predictions until the RNN recovers from the lack of context.

5.2.5. Methodology

Unlike the network structure used in the toy examples, we now use a three layer neural network based on the implementation from Höft, 2014, unfolded in time. Due to the large depth of our network, the model requires a large amount of training time, which prevents us from running many experiments. However, we have numerous parameters and features to be tested.

We can not afford to test all possible combinations of parameters, nor to use a hyperopt client which would as well require a large number of experiments to figure the right combination of parameters. We instead do experiments varying a set of parameters that, based on the results on smaller experiments seemed to

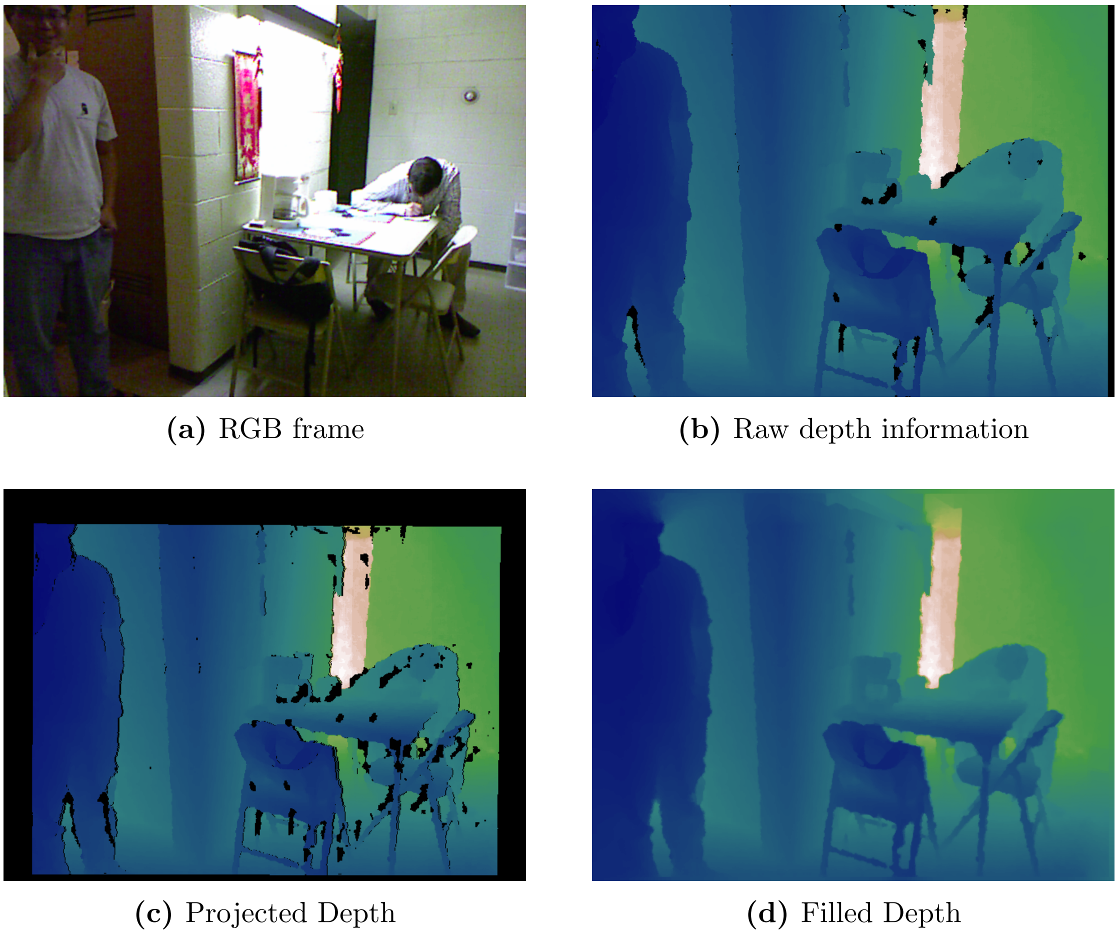

(a) RGB frame **(b)** Raw depth information **(c)** Projected Depth **(d)** Filled Depth

Figure 5.9: Steps done in preprocessing the depth. Notice that the raw RGB frame and the raw depth frame are not aligned. This is caused by different perspectives of the RGB and depth sensors. Projecting the depth aligns the two of them, but produces new holes in the depth map as well. After filling the depth missing values, we have a usable RGBD frame.

have a stronger impact:

- Architecture type
- Intermediate Ground Truth
- Weight Decay
- Dropout

The reason why we included two regularizers in this list is the fact that we noticed a tendency to overfit.

In the second batch of experiments, we chose one of the most promising configurations and we add gradually the rest of the extra featured, investigating which of them actually improve the results. We will, of course, not be able to derive all correlations between different configuration parameters, but such an extensive analysis is beyond our scope and our current possibilities.

5.2.6. Results, first stage of experiments

During this stage we obtained numerous results and we will highlight here those that we consider are enough to provide an insight on the effect of each parameter.

Simplified Network

Table 5.5 presents the results obtained using different configurations involving a simplified network. We notice that there is a tendency to overfit since the configurations that include regularizers obtain, in general, better results.

Configuration	Class Accuracy (%)	Pixel-wise Accuracy (%)
Base	59.2	59.3
Base + WD	58.1	59.9
Base + FC	58.6	59.6
Base + FC + DO	59.2	**60.0**
Base + IGT	58.2	59.8
Base + FC + WD + DO	59.3	59.9
Base + FC + DO + IGT	**59.4**	59.9

Table 5.5: Results obtained on the test set of NYUD using the simplified architecture. WD = Weight decay, DO = Dropout, IGT = intermediate ground truths

Configuration	Class Accuracy (%)	Pixel-wise Accuracy (%)
Base	61.8	61.8
Base + WD	62.0	61.8
Base + IGT	61.9	61.9
Base + FC + DO	60.7	60.9
Base + FC	62.2	62.3
Base + FC + IGT	61.5	61.3
Base + FC + DO + IGT	**62.3**	**62.7**
Base + FC + WD + DO + IGT	61.2	62.0

Table 5.6: Results obtained on the test set of NYUD using the unidirectional architecture. FC = final convolution, WD = Weight decay, DO = Dropout, IGT = intermediate ground truths

The best result that we obtain with respect to the pixel-wise accuracy is when using dropout. The pixel-wise accuracy boost is of about 0.8%. On the other hand, when using a combination of dropout and intermediate ground truths we obtain improvements of both the class and pixel-wise accuracy.

Another thing to notice is that using both weight decay and dropout simultaneously did not further improve the results significantly.

Unidirectional Network

A similar overview, for the unidirectional architecture, is presented in Table 5.6. In this case, as well, we notice a significant improvement when using intermediate ground truths, but only when combined with dropout. The pixel-wise accuracy boost is also about 0.8%. The same configuration obtains the best class accuracy as well, clearly outperforming the other configurations.

Weight decay was unable to affect the results in a positive way, neither on its own nor in combination with dropout or intermediate ground truths. Also an interesting fact is that dropout on its own is ineffective, unless intermediate ground truths are used.

Bidirectional Network

Finally, the results for the last tested architecture are presented in Table 5.8. The accuracy obtained by the network without any add-ons is already more than 62%, larger than in case of the unidirectional network. We were able to improve the pixel-wise accuracy by only 0.2% by adding regularizers and intermediate ground

Configuration	Class Accuracy (%)	Pixel-wise Accuracy (%)
Base	61.9	62.3
Base + WD	61.1	61.8
Base + FC + DO	61.9	61.9
Base + IGT	60.1	61.5
Base + WD + DO	60.6	61.9
Base + FC +DO + IGT	**62.6**	62.2
Base + FC + WD + DO + IGT	62.2	**62.5**

Table 5.7: Results obtained on the test set of NYUD using the bidirectional architecture. WD = Weight decay, DO = Dropout, IGT = intermediate ground truths

truths, but surprisingly we were able to obtain an improvement of 0.7% in the class accuracy, surpassing the results of the other architectures with respect to this metric.

We noticed that sometimes, during the training of the bidirectional network, an unwanted effect occurs. This effect is pictured in Figure 5.10. Whenever we encounter an unfortunate initialization that favors one direction in learning faster, there is a risk the network will learn to ignore the other direction. This means that the weight set corresponding to one direction becomes zero and will not be able to recover from this state. Since this situation occurs rarely, one solution to solve this would be to restart the experiment, re-initializing the network and thus starting from another position on the error surface.

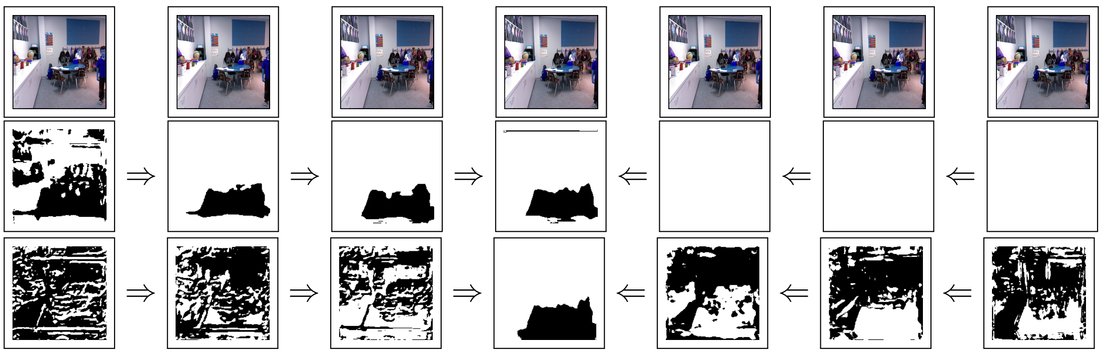


Figure 5.10: First row shows the RGB sequence given as input. The second and third row show the activation corresponding to the class floor, when the described effect occurs and respectively when not. The arrows show the propagation direction, and the middle output is the final output of the network.

Another solution would be to have an unique parameter set for both directions. This would however mean that both directions learn the same features, which would simplify the network but limit its flexibility at the same time.

Analysis

Comparing the results of the three experiments, we notice that the best result with respect to pixel-wise accuracy was obtained using an unidirectional architecture. However, on this metric, the difference between the unidirectional and bidirectional network was of less then 0.2%. On the other hand, if we consider the class-accuracy we notice that the best result is obtained by a bidirectional architecture. However, since the networks are optimized for the pixel-wise accuracy, a such large class accuracy, given the unbalanced classes, might suggest that the network was still in an early stage of learning.

Although the bidirectional network should make better use of the input data, in this set of experiments we did not get the improvement we hoped for compared to the unidirectional. This might also suggest that the context is too small, since we had to halve it to keep the same number of inputs.

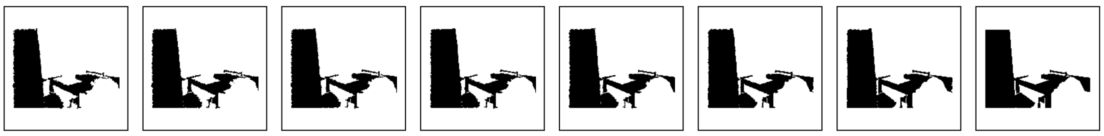
Another observation is that, in all three architectures, the best configurations used dropout and intermediate ground truths. Indeed, if we have a look at Figure 5.11b, we notice that when using IGT, a more structured representation takes shape even for the time-steps where we do not have ground truth. When using ground truth only for the last frame (Figure 5.11c), only the last two time steps show structure, while the others develop a chaotic representation that only changes slightly. At the same time, we do not want to constraint the network to learn the exact representation in an early time-step, since that would limit the network flexibility by also expecting an early convergence. For this reason, the intermediate ground truths are weighted less than the final ground truth.

Using the simplified network we obtained significantly worse results. This is consistent to the results obtained in the toy example and a sign that we should not hope having a winning configuration among those relying on such an architecture. On the other hand this proves learning temporal dependencies to be beneficial.

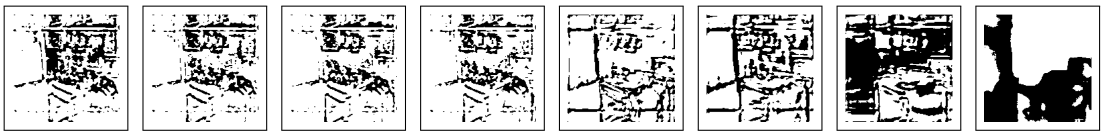
5.2.7. Results, second stage of experiments

In the second set of experiments we attempt to fine-tune the results by applying several extensions to configurations that obtained promising results in the first set of experiments. We will present each of these extensions and briefly discuss the results.

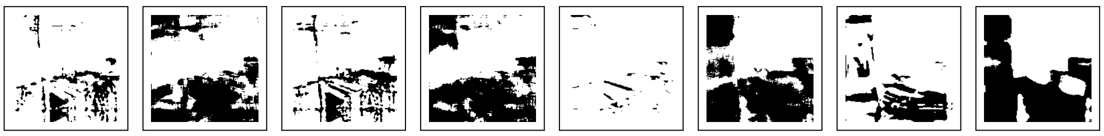
(a) Original RGB frame, NYUD dataset



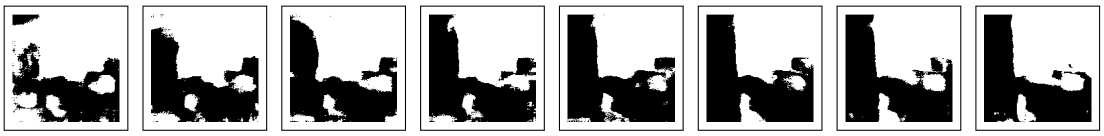
(b) Ground truth for class “furniture”



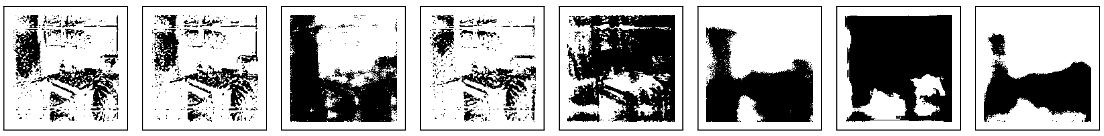
(c) Activations: No intermediate ground truth



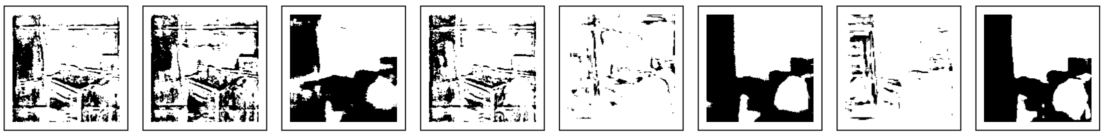
(d) Activations: IGT in time-steps two and five



(e) Activations: IGT in all time-steps



(f) Activations: Plain Gradient



(g) Activations: Multiscale inputs

Figure 5.11: Activations for different network configurations

Configuration	*Class Accuracy (%)*	*Pixel-wise Accuracy (%)*
Base	62.3	62.7
Base + SS1	61.9	61.9
Base + PS176	62.2	62.6
Base + FS5	62.0	60.9
Base + IGTALL	61.8	61.8
Base + 16-32-64	61.8	60.4
Base + NSB	62.6	62.0
Base + PLAIN	49.7	53.6
Base + PC	59.9	60.6
Base + MS	62.4	63.1
Base + SW	**69.5**	**66.8**

Table 5.8: Results obtained on the test set of NYUD using the unidirectional architecture + extensions. All extensions together with their abbreviations are described in this section.

Single time step (SS1)

One might argue that the results that we obtain are due to using a strong convolutional model, on which the network is relying, ignoring the information received from the other time steps.

This experiment shows this is not the case. Using only the input corresponding to the current time step leads to a lower performance then when a context of eight time steps is used.

Pattern size of 176 pixels (PS176)

Up to this point we always scaled the input to 160x160, using reflective border to switch from a 1,33:1 aspect ratio to 1:1. Our network produces an output of size 160x160 which we upscale to 640x640 and compare to the original, ignoring the extended border.

We have chosen 160 as pattern size since it is a divisor of 640, the network being responsible of representing in one pixel a patch of 4x4 pixels. On the other hand, Höft et al., 2014 used by default 176 as pattern size.

The result of this experiment shows that a pattern size of 160 is more appropriate.

Smaller filters (FS5)

Simplifying the model is another way one could cope with the over-fitting. Reducing the filter size also reduces the amount of weights and thus simplifies the model. However, as the experiment shows, this did not proved to be beneficial in our scenario. Moreover, by inspecting the loss on training and validation data, we noticed that the over-fitting has not decreased. This could suggest that the over-fitting is mainly caused by the redundancy present in the input data rather than due a too complex model.

Plain Gradient + Norm Clip (PLAIN)

The worst results were obtained using the plain gradient instead of RmsProp. Using norm clipping we were able to avoid gradient explosion, an evidence of this is the fact that the network did not saturate after 600 epoch. The problem with this approach is that norm clipping only prevents gradient explosion but not gradient vanishing. Indeed, we notice from Figure 5.11f that some of the features are forgot after about 4 frames, the network relaying on a very limited context. We are aware that different gradients may have different ranges for the optimal learning rates, and thus we have done this experiments with three different values chosen from the logarithmic scale and kept the best result. The progression of the loss function on training and test set showed a decreasing tendency and a small overfit, which suggests that we would have to wait longer for the convergence to local minima. Due to the large amount of time an epoch lasts, this is not a viable solution.

Ground Truth in all time steps (IGTALL)

Even though counter-intuitive, telling the network how the output should look like at every time-step leads to worse performances. This might be explained by the lack of freedom the network has to develop its own internal representation. One can notice this aspect in Figure 5.11e. Already after the first two time-steps, the network is already close to a local minima to which it converges early.

Number of maps doubles every layer (16-32-64)

Doubling the number of maps after each level of abstraction might be a good solution in some scenarios since this means having more features as the resolution of the representation decreases. In our case, however, such an architecture did not improve the results.

After testing different such configurations, using 16,32,64 maps for our three layers seemed to be the most promising one. However, the results are significantly lower then those obtained with 32 maps in all layers. The optimal number of features for each layer is, however, difficult to be determined analytically as it depends on the application and on the dataset.

Not sharing biases (NSB)

As previously mentioned, not sharing biases could lead to managing time steps differently, which is desirable in some situations.

We notice an increase of more than 0.2% in the class accuracy, which is unexpectable due to the significantly lower pixel-wise accuracy. The individual accuracies per class showed that the accuracy for the least frequent classes: “Ground” and “Prop” increased by 3%. On the other hand, the accuracies for “Structure” and “Furniture”, decreased by about 0.2%, which explains the decrease of the pixel-wise accuracy. This result, along with the fact that the loss function used minimizes the pixel-wise accuracy suggest that the network was still far from the local minima at the end of the 600 epochs.

Pairwise class location filter (PC)

We investigated the weights learned by the PC filter, to determine the reason the network performs poorly when this extension was used. Although the weights shown structure, it looked like for training the PC, more epochs were necessary. This is due to the large filter that the PC filter has, and thus larger weight set, compared to the other convolutions of the network. Since the network itself produces already strong predictions after 600 epochs, and an epoch is expensive, one approach would be to train the network in two stages, one for training the network and one for training the PC filter separately.

Multi-scale input (MS)

A significant improvement was obtained after using inputs at different scales. The improvement in terms of pixel-wise accuracy is of almost 0.5%. Also the class-accuracy is slightly improved. As we can notice from Figure 5.11g, the internal representation of the input has a more defined structure than in the case where we use only one scale for the input. This leads to a more accurate prediction.

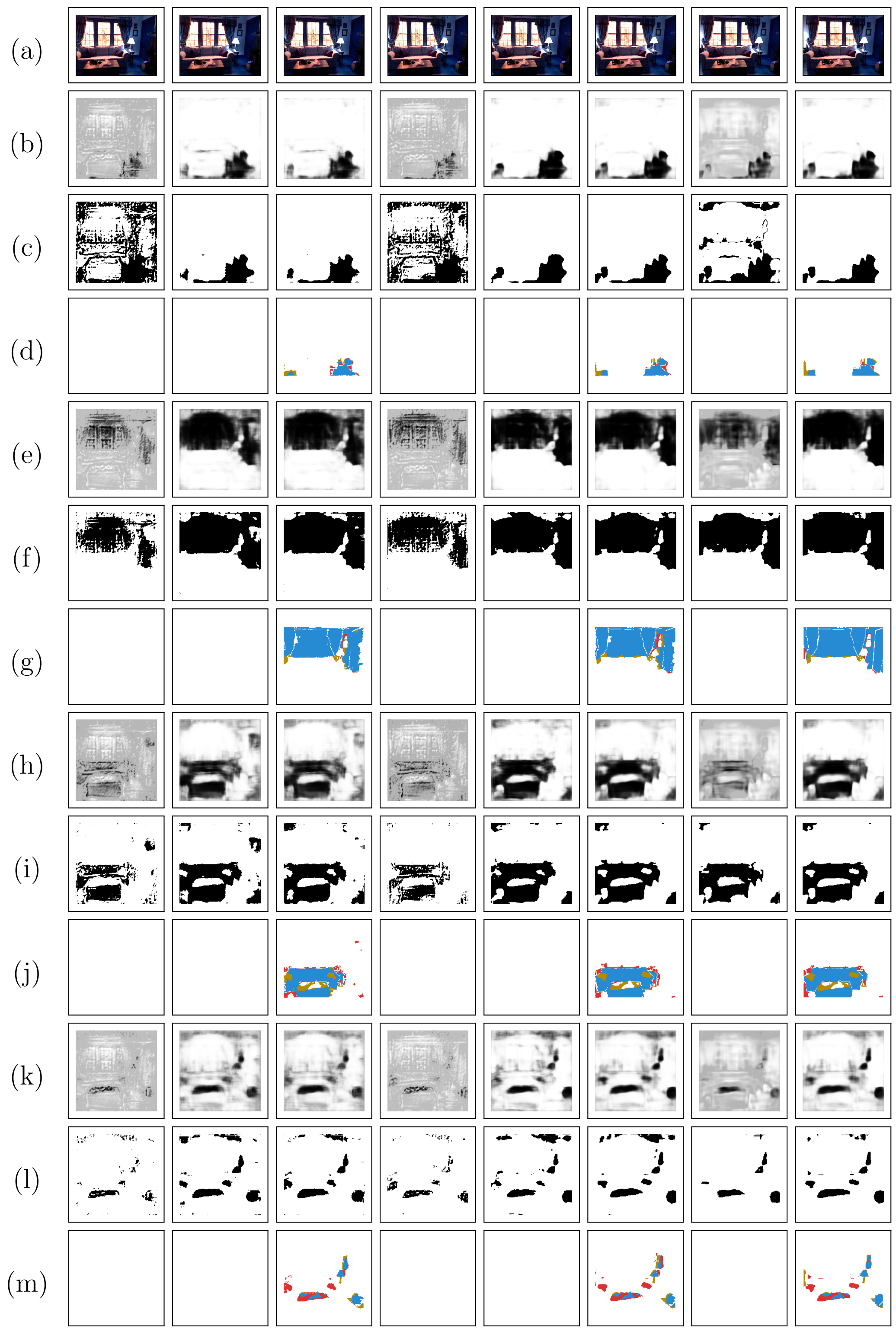


Figure 5.12: Prediction for one sample of NYUD test dataset. Row (a) is the RGB input. Rows (b), (e), (h), (k) represent the softmax layer output; rows (c), (f), (i), (l) represent the output of the network; rows (d), (g), (j), (m) represent the evaluation (🔵 True Positives ○ True Negatives 🔴 False Positives 🟤 False Negatives) for the classes ground, structure, furniture and prop respectively .

Sliding Window (SW)

The use of sliding windows enhanced both the class and pixel-wise accuracy. An explanation for this result could be the fact that the network has to focus now on the dynamics that take place within a smaller spatial context and thus, is able to track easier smaller objects. This is supported also by the increase in the accuracy of classifying "Ground" and "Prop" objects, whose classes are less frequent than the "Furniture" and the "Structure" classes.

Also, the network has a more broader set of inputs since the choice of the window that determines the input of the network is non-deterministic. This has the potential of reducing over-fitting.

Finally, since the network processes small patches, we also reduce the size of the input the network receives from 160 to 80. This leads to a simpler model, faster to be trained. One drawback is the time required for prediction, since the network has to process patches that fully cover the input and combine their prediction afterwards.

5.2.8. Overview on the results

We further discuss our results on the NYUD dataset in overview, comparing them with other results obtained on this dataset.

Ability to exploit temporal dependencies

It is clear from the experiments that the network is able to learn temporal dependencies, since the results obtained using an architecture that accumulates information over several time steps performs better than the equivalent architecture processing only one time step. However, we wish to provide an additional argument to this statement, by showing the learned filters of our network. As previously described our network uses three types of connections: Forward (Figure 5.13), lateral (Figure 5.14) and backward (Figure 5.15). All weight sets show structure, which means that both the horizontal and the vertical flow of activations are active.

Overfitting

Overfitting was one of the effects that limited right from the first versions of our network the ability to learn and generalize well. Using dataset augmentation was constantly helpful up in the small experiments we have carried, thus we decided to use it by default in all of our experiments.

Regularizers helped only partially, dropout being part of the configurations which obtained the best results but only when we also evaluated the network

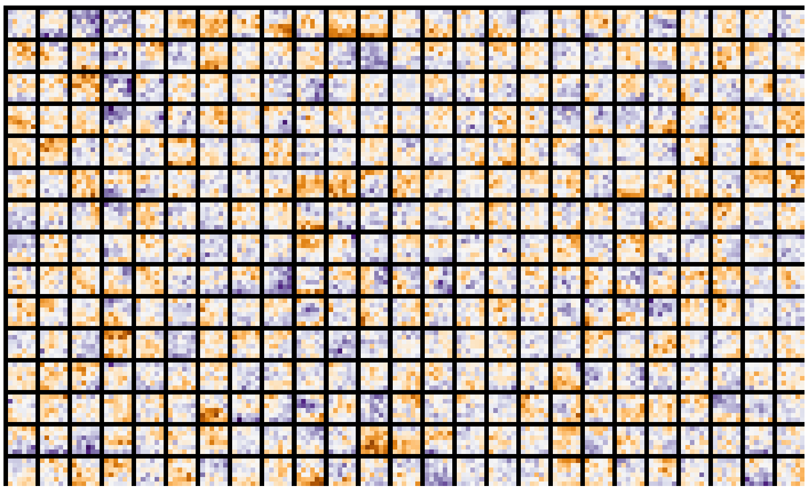


Figure 5.13: Forward connections between Layer 2 and Layer 3

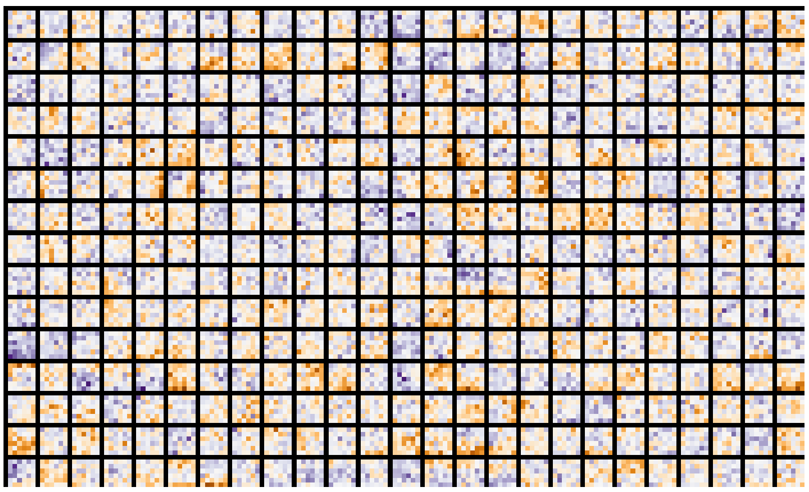


Figure 5.14: Lateral connections between Layer 2 and Layer 2

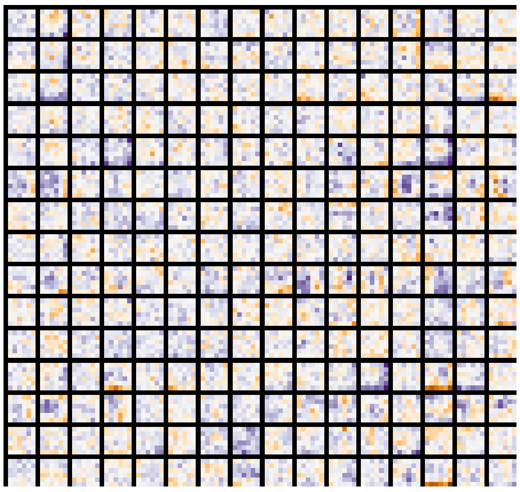


Figure 5.15: Backward connections between Layer 2 and Layer 1

in intermediate time-steps. In other words, only when we changed the structure of the network. Generally weight decay did not help, even though we tried different values, from the logarithmic scale, for the decay factor. This could suggest that most of the overfitting is due to the redundancy of the data rather than due to the complexity of the model. The success of dropout and dataset augmentation also support this claim.

5.2.9. Comparison with other results

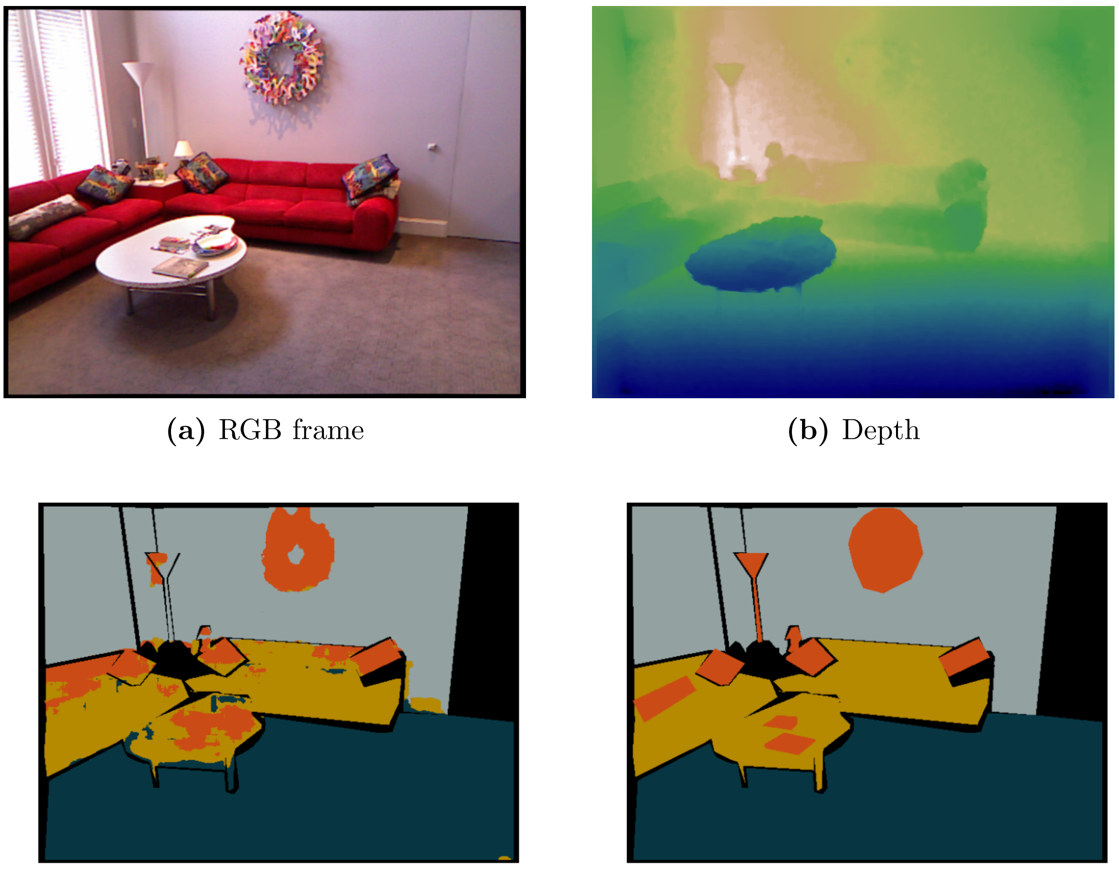
We compare the results of our network with the results obtained by the baseline models. We use two baseline models for two different situations: when sliding window is used and when not.

Our recurrent neural networks uses the convolutional model from Höft et al. (2014), with a few changes. We use it as baseline for our implementation without sliding window. We used a simplified version of their convolutional model, not training the network stage-wise, since that would emphasize on the convolutional connections more than on the recurrent ones, which we wish to avoid. Also, during training we did not notice the need of using such a technique since all three layers showed similar relative weight updates. Another change is that we do not use the pair wise localization filter (PC filter). We use a final convolution to extract the output from the filters of the bottom layer, but with the same filter size as the other convolutions. While a PC filter might be useful against spurious observations, we noticed that it would require more training then the rest of the model, and it was not an option to further slow experiments down.

We chose the model which obtained the best results in our experiments, namely the unidirectional network using two intermediate ground truths, a final non-recurrent convolution onto which we apply dropout and finally, we provide the network with inputs at different scales depending on the level of abstraction. The

Method	ground	struct	furnit	prop	Class Avg.	Pixel Acc.
Baseline	77.9	65.4	55.9	49.9	62.0	61.1
UNI + IGT + DO + MS	73.4	66.8	60.3	49.2	62.4	63.1
Baseline (SW)	87.7	70.8	57.0	53.6	67.3	65.5
UNI + IGT + DO + SW	90.2	74.5	52.2	61.0	69.5	66.8

Table 5.9: Comparison with the baseline. UNI = Unidirectional recurrent neural network SW = Sliding Window, DO = Dropout, IGT = intermediate ground truths, MS = multi-scale input



(a) RGB frame **(b)** Depth

(c) Prediction **(d)** Ground Truth

Figure 5.16: Prediction for one of the NYUD dataset frames. Images (a) and (b) represent together the RGBD frame after being preprocessed. Image (c) and (d) represent the "floor" (●), "prop" (●), "furniture" (●), "structure" (●) and "ignore".(●) Not only were we able to detect almost all object, but we also detected objects that were not labeled (e.g the third object on the table).

results show an improvement of 2% in the pixel accuracy and of about 0.4% in the class accuracy.

The sliding window approach we use is an adapted version of the one implemented in Schulz et al., 2015. Although they used a simple feed-forward network as a model, due to the sliding window approach with depth-normalized patch sizes they were able to outperform the state of the art results. Another piece of their design, having an important impact on the results is the computation and use of the actual height map. Having access to the height information can help detect elements that have specific vertical locations in the room (such as the floor) more accurate and avoid many false positives. Computing the height maps for all images in our sequence dataset was not an option for us due to the limited time, so we chose to only use the sliding window and depth-normalized patch sizes approaches. Thus, we also compare our result with the result they obtained without the height feature. This time the improvement of the class accuracy is of more than 2% and of more than 1% in pixel-wise accuracy.

Table 5.10 shows our result together with the results of different models. Here we also show the result of Schulz et al., 2015 with the previously discussed model, with the height map included. Müller and Behnke (2014) use Conditional Random Fields over a Random Forest prediction aggregated in super pixels. They use the height feature as well.

Our method is still behind the state of the art method, but shows promising results. We have high accuracy in detecting classes that occur less frequent such as “Ground” and “Prop”, while the accuracies for the frequent classes “Structure” and “Furniture” remain competitive. Using the height feature might further improve our result, especially the accuracy for the “Ground” class.

Method	ground	struct	furnit	prop	clsav	pixav
UNI + IGT + DO + SW	90.2	74.5	52.2	**61.0**	69.5	66.8
Müller and Behnke (2014)	**94.9**	78.9	**79.7**	55.1	**71.9**	**72.3**
Schulz et al., 2015	93.7	72.5	61.7	55.5	70.9	70.5
Stückler et al. (2013)	90.8	81.6	67.9	19.9	65.0	68.3
Couprie et al. (2013)	87.3	**86.1**	45.3	35.5	63.5	64.5
Höft et al. (2014)	77.9	65.4	55.9	49.9	61.1	62.0
Silberman et al. (2012)	68	59	70	42	59.6	58.6

Table 5.10: Comparison with other results from the literature . UNI = Unidirectional recurrent neural network SW = Sliding Window, DO = Dropout, IGT = intermediate ground truths, MS = Multi-scale input

We must admit, however, that a direct comparison of our result with the result of others is not fair since we accumulate information over a larger context, while other works listed here work only with the current frame. Also, due to the problems described in Section 5.2.4, seven out of 654 images could not be evaluated. Still, this does not have a significant impact on our results, since even if we would consider obtaining the chance accuracy for the seven images, which is already unlikely, we would still have competitive results.

On the other hand, this comparison shows the potential one has when exploiting such a larger context.

5.3. Time measurements

In this section we initially planned to present the time measurements based on the experiments that were already carried out. However, due to the long period an experiment takes, the high variance in the CPU usage over time, the fact that we used eight GPUs of two different machines, were all factors that influenced the time measurements and led to confusing results.

We thus perform a separate small set of experiments, training only for a few epochs during a short period of time. Also, we run these experiments sequentially, on the same GPU.

Table 5.11 shows that the most expensive architecture in term of time consumption is, surprisingly, the simplified one. Although the depth of such a network is about quarter the depth of the unidirectional network, the number of channels increases proportionally to the number of temporal steps covered. Also, several concatenate and subtensor operators constitute an important overhead. On the other hand, as expected, the bidirectional network is slower than the unidirectional network that covers the same temporal context. It is however not slower by a factor of two, even though we include two unidirectional networks, due to the optimization we used, described in Section 4.1.3 .

The extensions might as well increase the training time. Using intermediate ground truths involves a small overhead, since the intermediate evaluations are included in the loss, which involves a traversal of a slightly larger network. This has a limited effect during prediction since we only need the output of one temporal step.

A final convolution causes a more substantial overhead since, compared to a subtensor operator, more calculations are involved. During prediction, we only do the forward propagation step, which leads to close timings between the two approaches.

Configuration	Training time / epoch (s)	Prediction time / image (s)
SIMP	26.8	0.8
SIMP + IGT	26.9	0.8
SIMP + FC	27.4	0.8
SIMP + MS	28,5	0.8
SIMP + SW	8.1	2.3
UNI	15.1	0.5
UNI + IGT	15.2	0.5
UNI + FC	15.6	0.5
UNI + MS	15.8	0.6
UNI + SW	5.9	0.9
BIDI	18.6	0.6
BIDI + IGT	18.8	0.6
BIDI + FC	19.5	0.6
BIDI + MS	19.6	0.6
BIDI + SW	10.8	1.6

Table 5.11: Time measurements. 16 images per batch used, 20 batches per epoch, input size is 160x160. SIMP = Simplified RNN, UNI = Unidirectional RNN, BIDI = Bidirectional RNN, IGT = intermediate ground truths, MS = multi-scale input. These measurements include the loading and the preprocessing of the images.

Using multiscale inputs affects the training time significantly. Inserting new inputs in the network is not trivial and involves several operators being added to the network (Figure 4.1).

In the case of sliding window, due to working with patches of the input, and thus processing them at 80x80 pixels instead of 160x160 pixels, training is significantly faster. Another positive effect is visible in the simplified architecture, where we no longer have to split the convolutions, as described in Section 5.2.2.

Among these results, the fastest approach requires about six seconds per epoch during training. For the prediction step, the best result is of half of second per image. This is a pessimistic perspective, since models such as the random forest, implemented in CUDA, and described in Waldvogel, 2013, can compute one prediction in 13 ms, and achieves a class accuracy only 4% smaller than ours. Since performance is an important factor in today's applications, the practical applicability of our model is questionable, at least at the moment. Along with the technological developments in graphic cards, which could make our model usable in several years, a neural network once trained can be implemented in hardware, as suggested in Farabet et al., 2010, leading to an important performance boost.

5.4. Difficulties

One of the difficulties of this work was the amount of time spent in transforming the NYUD dataset in a sequence dataset. We had to work with the raw RGBD frames and to redo all preprocessing steps done in Silberman et al., 2012 to obtain the image dataset, but for more frames. Although they provided Matlab scripts for preprocessing the dataset we had to convert the code into a more accessible and fast implementation, except for the depth filling which was already ported to Python (Waldvogel, 2013) and C++ (Schwarz, 2014).

The size of the raw dataset and the way it was organized was another issue that consumed time. The authors provide the dataset in one huge archive of 428 GB and alternatively in archives of about 20GB each. The split is done based on the sequence as well, but since the authors did not provide a meta-data file to specify in which archive a certain file exists, we found it more fast to actually search in the large archive, and to cache the search results. While the authors provided MD5 sums for each of the small archives, It was, however, important to be able to obtain some results on this dataset since we already had results obtained using the simple convolutional network, hence we could perform a comparison. Unfortunately we did not have the chance to test the network on another dataset, except for the toy experiment datasets, although having such results would have been interesting as well.

In this context, the fact that we had access to workstations with a large processing power, numerous CPUs, and, most important, powerful graphic cards played an key role in this work. For example, since the dataset preprocessing could be easily divided in sub-tasks and ran in parallel, we waited for about one week for its completion, while normally the task would have required months on a regular PC.

Apart from the handling the dataset, we also faced a set of limitations, some of them described in Section 5.2.1 and Section 5.2.2. We had to perform estimations in order to find the appropriate ranges for the parameters. Particularly the slowness of our experiments and the fact that we could run only a small number of them in parallel led to a late discovery of issues in the design, or problems in the dataset, since some of the problems required a large number of epochs to be detected. This was partially solved sometimes by running smaller and faster experiments, but we were not able to replicate some problems on small datasets.

The size of the networks that we implemented and the complex connectivity also made finding problems in the design much harder (Figure A.1, Figure A.2, Figure A.3). We built simplified versions of the function graph to detect high level issues, but for fine-tuning we had to follow edges in a large graph. Visualizations of the filters, the outputs of different time steps, helped a lot as well to detect inconsistencies.

Training such large networks also introduced the challenge of avoiding saturation of the non-linearity units. Since we use ReLU, we had to avoid situations in which maps resulted after a convolution are fully negative since those become zero maps after the non-linearity is applied and leads the network in a state from which it is unable to recover. Also, since ReLU is one-sided, not saturating for large activations, we also have to avoid exploding activations.

One way to avoid such situations is to have a careful initialization of the weights. Ideally, at the beginning, we would want to keep the activations from any point of our network in the same ranges. Once the training phase starts, provided that the learning rate is not too large, the network should adjust those ranges optimally. Keeping the same ranges is however difficult to control, especially in such large networks, and we found easier to adjust the parameters such that the values of the activations are on average positive, and they slightly decrease with every time step, making sure that every set of activations remain positive on average as well. We found that this simple strategy is enough to provide a good starting point for training the network.

In addition to the problems presented here that arised during the development of this work, we also have reasons to be self-critical. We were skeptical from the start that RMSProp together with ReLU will work well in our recurrent network, despite the fact that they were successful in training deep convolutional neural networks. Thus, we invested additional time in implementing and testing the gradient clipping, which was not trivial due to design constraints, before even trying to use RMSProp on NYUD dataset.

6. Conclusion

In this work we successfully implemented a recurrent neural network, able to learn not only spatial dependencies as a convolutional network does, but also to learn temporal ones from a sequence of video frames presented as input. We started with a series of toy examples that helped us detect initial problems in the design of our networks and at the same time showed that the recurrent structure allows the network to learn temporal dependencies needed for solving tasks such as denoising. Also, our toy examples showed that the networks can track and infer motion, both features being important pieces later on, when working on the real-life video sequences.

The next part was a process of adapting the NYUD dataset to our needs, providing an additional temporal context to each frame. Experiments on this dataset highlighted new problems and limitations that we had to handle. Large networks exceeded the amount of memory we had at our disposal, reason why we made an estimation for the GPU memory consumption based on the network configuration. This allowed us to know how each structural network influences the memory consumption of the network and which configurations are feasible. Another important issue was the large duration of our experiments. We have chosen the experiment set carefully and ran it on eight GPUs for about six weeks.

In addition to testing the three architectures, we also proposed and tested a set of extensions to the design, most of them inspired from other works on deep neural networks. Although few of those extensions were successful in improving the results we already had, we were able, using multiscale inputs and the sliding window approach, to shorten the gap between our results and the results obtained by the state-of-the-art method. We showed that our recurrent model is outperforming the convolutional model in relies on, which suggests that our network is able to make use of the temporal context.

When comparing the three architectures, we could conclude that the simplified RNN is weaker then the other two architectures which process one input at a time. The comparison between the unidirectional network and the bidirectional one showed that the bidirectional performs worse than the unidirectional most likely due to having a shorter temporal context to exploit, in the two compound sub-networks.

We performed time measurements of our models. As we expected, such a large neural network requires a long period of time for training and prediction. Even when powerful graphic cards are used, such a model needs about half of second to produce a prediction. This means that we are far from being capable of producing predictions in real-time. However, our work, among many others, reinforces the idea that such recurrent models, together with convolutional connections, are promising especially for processing video sequences.

6.1. Future work

Our design can be improved in many ways. Also, the method for generating the dataset could be optimized. For example, the algorithm for generating the intermediate ground truths is a very basic one, simply propagating the label according to the optical flow calculated based on the two consecutive frames. The algorithm could include heuristics for determining regions where we are not certain about the labeling (e.g where the optical flow is zero), and represent such regions in the ignore mask. Another improvement would be to decide the ground truth of a certain time step based on several previous time steps, not only on the last one.

Currently, when building a sequence, we consider the temporal distance between frames and try to keep this distance constant. However, this does not take into account the speed of the camera. A better solution would be to keep the spatial distance between the camera perspectives constant. This would be possible, since we have access to the accelerometer data.

Also, it would be interesting to see results of such a model when a wider context is used. This is not applicable on the NYUD dataset due to frequent gaps where we have no RGB or Depth information.

One important addition to our model would be to consider the height as suggested in the Section 5.2.9. The height feature provides the network with important information, especially useful when sliding window approach is used, since we have no indication at all where are we located inside the image. By observing the predictions produced by our model we noticed that, many times, the ceiling for example is confused with the ground. Having the height as input would help avoiding such situations. Computing the height for a frame, based on its depth, is not trivial and has to be performed for all frames of the sequence, a slow process that, together with the limited time we had at our disposal, led us to the decision of not include this in our work.

Another interesting improvement would be a post-processing of the predictions, removing spurious observations. This can be implemented as an additional convolution, of a larger filter size, which learns making the predictions more smooth. Such a convolution, however, would require more epochs for being trained, and thus, for this situation, training the network in stages could be an idea to integrate such a concept.

Finally, additional effort could be invested in determining the optimal parameters of the network. However, exploring a large number of configurations also requires a large number of experiments, making potential speed optimizations very important.

Bibliography

Behnke, S. (2003). *Hierarchical Neural Networks for Image Interpretation.* Vol. 2766. Lecture Notes in Computer Science. Springer-Verlag (cit. on p. 4).

Bengio, Y., P. Simard, and P. Frasconi (1994). “Learning Long-Term Dependencies with Gradient Descent is Difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (cit. on p. 13).

Ciresan, D., U. Meier, and J. Schmidhuber (2012). “Multi-column deep neural networks for image classification”. In: *Computer Vision and Pattern Recognition* (cit. on p. 3).

Couprie, C., C. Farabet, L. Najman, and Y. LeCun (2013). “Indoor Semantic Segmentation using depth information”. In: *The Computing Resource Repository* (cit. on p. 65).

Elman, J. L. (1990). “Finding structure in time”. In: *Cognitive Science* 14.2, pp. 179–211 (cit. on p. 12).

Farabet, C., B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello (2010). “Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems”. In: *International Symposium on Circuits and Systems* (cit. on p. 67).

Farabet, C., C. Couprie, L. Najman, and Y. LeCun (2013). “Learning Hierarchical Features for Scene Labeling”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (cit. on p. 3).

Fukushima, K. (1980). “Neocognitron: A self-organizing neural network model for a mechanish of pattern recognition unaffected by shifts in position”. In: *Biological Cybernetics* 36 (cit. on pp. 1, 9).

Glorot, X., A. Bordes, and Y. Bengio. In: *Journal of Machine Learning Research* (cit. on p. 11).

Grangier, D., L. Bottou, and R. Collobert (2009). *Deep Convolutional Networks for Scene Parsing* (cit. on p. 3).

Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks.* Vol. 385. Studies in Computational Intelligence. Springer, pp. 1–131. ISBN: 978-3-642-24796-5 (cit. on pp. 5, 6, 16).

Graves, A., A. rahman Mohamed, and G. E. Hinton (2013). “Speech Recognition with Deep Recurrent Neural Networks”. In: *International Conference on Acoustics, Speech and Signal Processing* (cit. on pp. 5, 15).

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2012). “Improving neural networks by preventing co-adaptation of feature detectors”. In: *Computing Research Repository* (cit. on p. 26).

Hochreiter, S. and J. Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8 (cit. on p. 14).

Höft, N. (2014). “Bildsegmentation in Objekt-Klassen mit Konvolutionalen Neuronalen Netzen” (cit. on pp. 17, 19, 20, 50).

Höft, N., H. Schulz, and S. Behnke (2014). “Fast Semantic Segmentation of RGB-D Scenes with GPU-Accelerated Deep Neural Networks”. In: *German Conference on Artificial Intelligence* (cit. on pp. 57, 63, 65).

Hopfield, J. J. (1982). “Neural Networks and Physical Systems with Emergent Collective Computational Abilities”. In: *Proceedings of the National Academy of Sciences* 79 (cit. on p. 12).

Jesorsky, O., K. J. Kirchberg, and R. W. Frischholz (2001). “Robust Face Detection Using the Hausdorff Distance”. In: *Audio and Video-Based Person Authentication.* Lecture Notes in Computer Science. Springer (cit. on p. 5).

Jordan, M. I. (1986). *Serial Order: A Parallel Distributed Processing Approach.* Tech. rep. Institute for Cognitive Science, University of California, San Diego (cit. on p. 12).

Krizhevsky, A. (2009). “Learning multiple layers of features from tiny images” (cit. on p. 18).

Le, Q., M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng (2012). “Building high-level features using large scale unsupervised learning”. In: *International Conference in Machine Learning* (cit. on p. 3).

LeCun, Y., K. Kavukvuoglu, and C. Farabet (2010). “Convolutional Networks and Applications in Vision”. In: *Proc. International Symposium on Circuits and Systems* (cit. on p. 9).

Levin, A., D. Lischinski, and Y. Weiss (2004). “Colorization Using Optimization”. In: *Special Interest Group on Graphics and Interactive Techniques* (cit. on p. 50).

M. Jung J. Hwang, J. T. (2014). “Multiple Spatio-Temporal Scales Neural Network for Contextual Visual Recognition of Human Actions”. In: *International Conference on Development and Learning and on Epigenetic Robotics* (cit. on p. 3).

Martens, J. (2010). “Deep learning via Hessian-free optimization.” In: *International Conference on Machine Learning* (cit. on p. 14).

Müller, A. C. and S. Behnke (2014). “Learning Depth-Sensitive Conditional Random Fields for Semantic Segmentation of RGB-D Images”. In: *International Conference on Robotics and Automation.* Hong Kong (cit. on p. 65).

Newcombe, R. A., S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon (2011). “KinectFusion: Real-time Dense Surface Mapping and Tracking”. In: *International Symposium on Mixed and Augmented Reality* (cit. on p. 49).

Pascanu, R., T. Mikolov, and Y. Bengio (2013). “On the difficulty of training recurrent neural networks.” In: *Journal of Machine Learning Research.* Vol. 28 (cit. on pp. 13, 24).

Pham, V., T. Bluche, C. Kermorvant, and J. Louradour (2014). “Dropout improves recurrent neural networks for handwriting recognition”. In: *International Conference on Frontiers in Handwriting Recognition* (cit. on p. 26).

Pinheiro, P. H. O. and R. Collobert (2013). *Recurrent Convolutional Neural Networks for Scene Parsing.* Tech. rep. Idiap Research Institute (cit. on pp. 6, 12).

Räsñen, O. (2013). “Studies on unsupervised and weakly supervised methods in computational modeling of early language acquisition” (cit. on p. 15).

Riedmiller, M. and H. Braun (1993). “A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm”. In: *International Conference on Neural Networks* (cit. on p. 23).

Schulz, H. and S. Behnke (2012). “Learning object-class segmentation with convolutional neural networks”. In: *European Symposium on Artificial Neural Networks* (cit. on p. 3).

Schulz, H., N. Höft, and S. Behnke (2015). “Depth and Height Aware Semantic RGB-D Perception with Convolutional Neural Networks”. In: *European Symposium on Artificial Neural Networks (submitted)* (cit. on pp. 22, 65).

Schwarz, M. (2014). “Objektklassifikation, Identifizierung und Posenschätzung mit Hilfe von vortrainierten Konvolutionsnetzen” (cit. on pp. 50, 68).

Silberman, N., D. Hoiem, P. Kohli, and R. Fergus (2012). “Indoor segmentation and support inference from RGBD images”. In: *European Conference on Computer Vision* (cit. on pp. 2, 65, 68).

Socher, R., C. C. Lin, A. Y. Ng, and C. D. Manning (2011). “Parsing Natural Scenes and Natural Language with Recursive Neural Networks”. In: *International Conference on Machine Learning* (cit. on p. 3).

Stückler, J., B. Waldvogel, H. Schulz, and S. Behnke (2013). “Dense real-time mapping of object-class semantics from RGB-D video”. In: *Journal of Real-Time Image Processing* (cit. on p. 65).

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2014). “Going Deeper with Convolutions”. In: *Large Scale Visual Recognition Challenge Workshop* (cit. on pp. 4, 31).

Waldvogel, B. (2013). “Accelerating Random Forests on CPUs and GPUs for Object-Class Image Segmentation”. MA thesis. Rheinische Friedrich-Wilhelms-Universität Bonn (cit. on pp. 67, 68).

Yao, K., G. Zweig, M.-Y. Hwang, Y. Shi, and D. Yu (2013). “Recurrent neural networks for language understanding.” In: *Interspeech* (cit. on p. 16).

A. Appendix

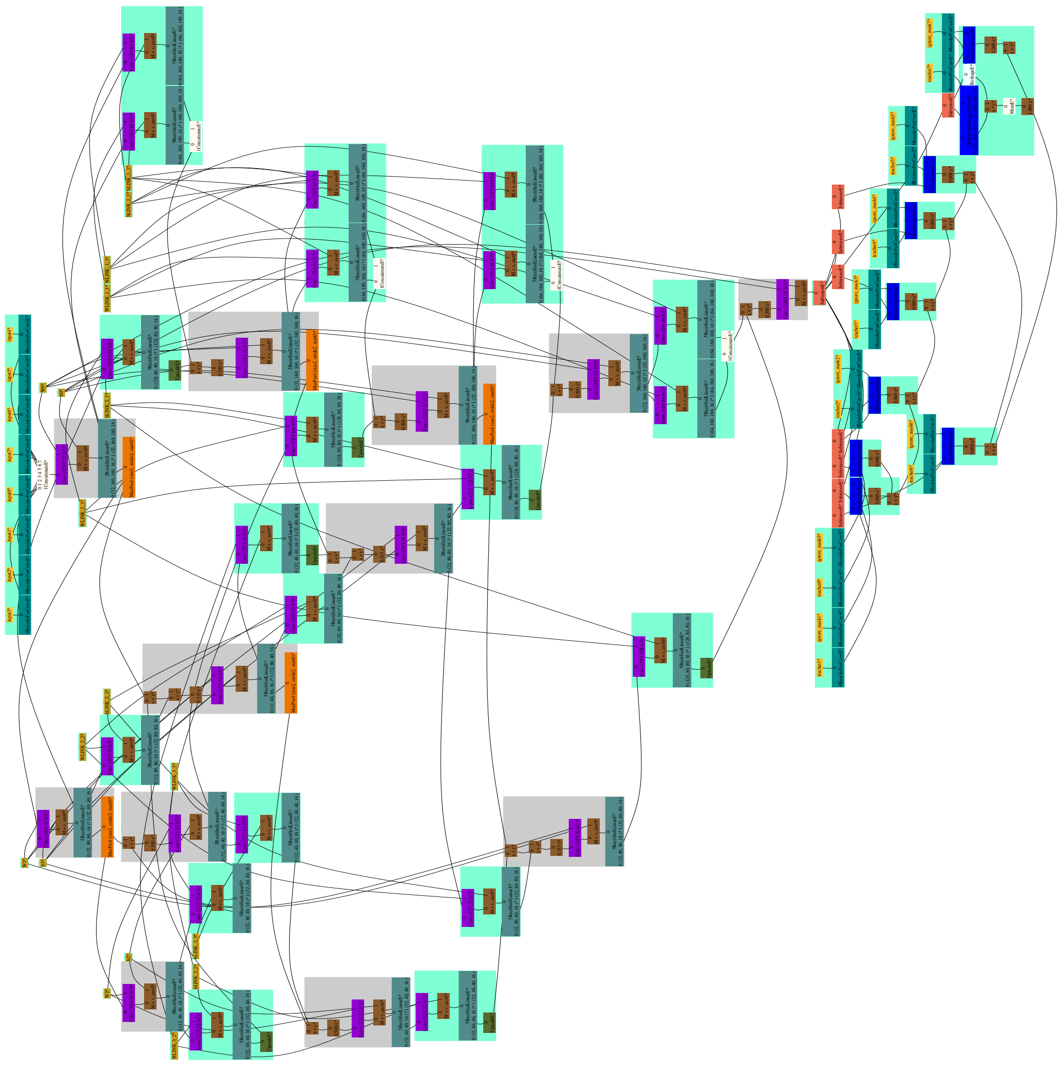


Figure A.1: Simplified architecture. Components: Inputs ●, Pooling ●, Convolution ●, Upscale ●, Subtensor ●, ReLU ●, Arithmetic Unit ●, Evaluation ●, Connector Layer ●, Hidden Layer ●

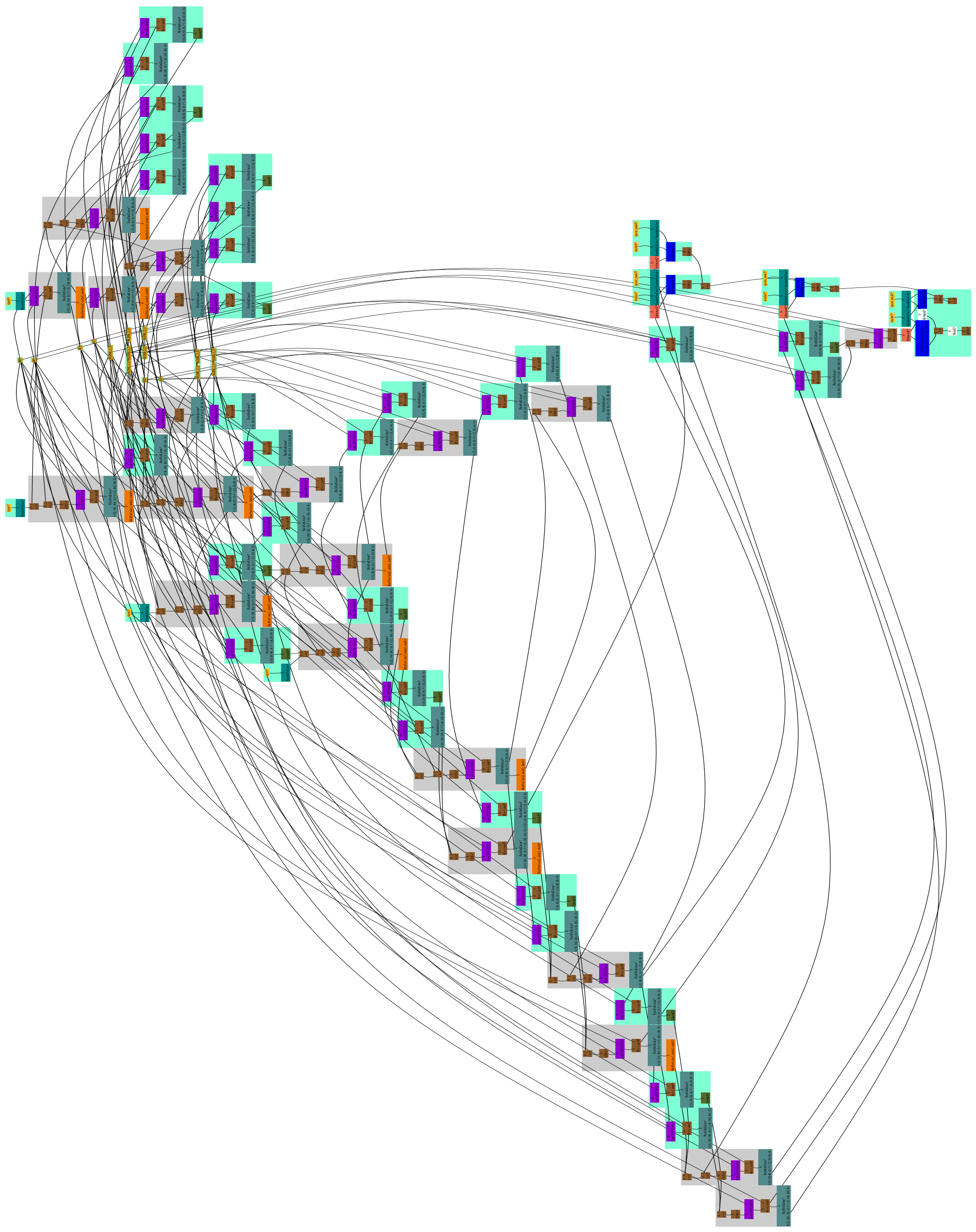


Figure A.2: Unidirectional architecture. Components: Inputs ●, Pooling ●, Convolution ●, Upscale ●, ReLU ●, Arithmetic Unit ●, Evaluation ●, Connector Layer ●, Hidden Layer ●

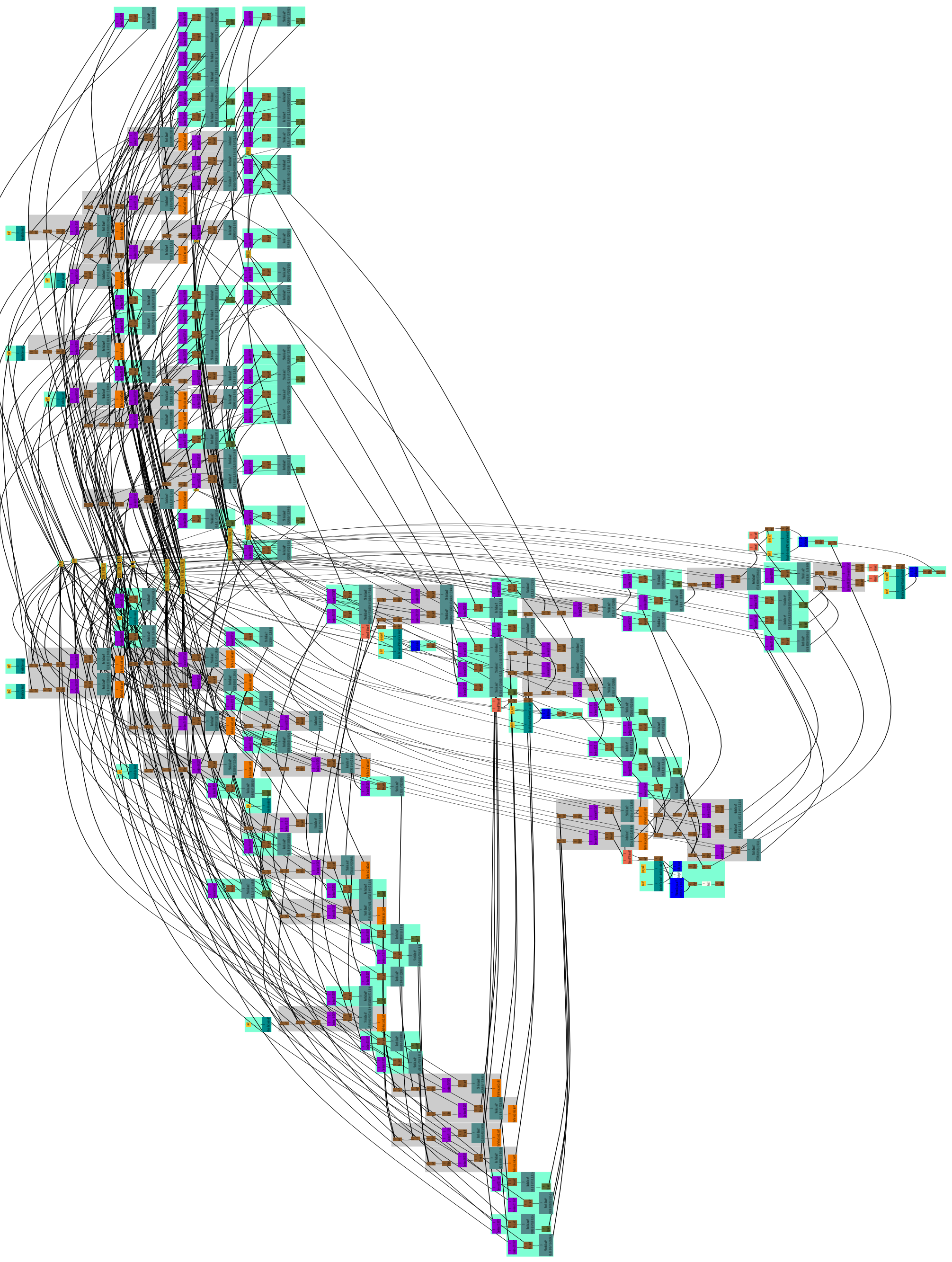


Figure A.3: Bidirectional architecture. Components: Inputs ●, Pooling ●, Convolution ●, Upscale ●, ReLU ●, Arithmetic Unit ●, Evaluation ●, Connector Layer ●, Hidden Layer ●