POLITECNICO DI MILANO
DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING
DOCTORAL PROGRAMME IN COMPUTER SCIENCE

DEEP RECURRENT NEURAL NETWORKS FOR VISUAL SCENE UNDERSTANDING

Doctoral Dissertation of:
Francesco Visin

Supervisor:
Prof. Matteo Matteucci

Co-Supervisor:
Prof. Aaron Courville

Tutor:
Prof. Andrea Bonarini

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2016 – XXVIII

All models are wrong, but some are useful.

GEORGE E. P. BOX

Abstract

ABSTRACT goes here.

Summary

SUMMARY goes here.

Contents

CHAPTER 1

Introduction

I am convinced that machines can and will think. I don't mean that machines will behave like men. I don't think for a very long time we are going to have a difficult problem distinguishing a man from a robot. And I don't think my daughter will ever marry a computer. But I think that computer will be doing the things that men do when we say they are thinking. I am convinced that machines can and will think in our lifetime.

– The Thinking Machine (Artificial Intelligence in the 1960s),
O. SELFRIDGE (LINCOLN LABS, MIT)

The dream of machines that can think and substitute humans in doing their jobs dates back to the '60s, if not before. We are still not at that point, although in the last decade the field experienced outstanding advancements and has been object of increasing interest. Machine Learning (ML) settled the state of art in many fields, such as e.g., image classification Krizhevsky *et al.* (2012a); Szegedy *et al.* (2016); Visin *et al.* (2015), semantic segmentation Chen *et al.* (2015); Visin *et al.* (2016), video understanding (Srivastava *et al.*, 2015; Xu *et al.*, 2015), natural language processing and machine translation (Bahdanau *et al.*, 2014). Much of this research is already in commercial products we use every day, such as e.g., speech recognition and speech synthesis in phones, face detection in cameras and socials, or traffic signs enhancement in cars. Even more impressively, a ML algorithm recently won several games of go (Silver and Hassabis, 2016) – a game known for being extremely challenging – against one of the best human players.

Despite its many successes, machine learning is no lamp genie that can tackle any problem by simply providing it with enough data. To get results in machine learning

requires a meticulous analysis of the characteristics of the problem, clever architecture modelling, smart engineering, as well as careful inspection of complex and extremely nonlinear compositions of transformations. Most of all, it requires good organization, intuition and patience, since many of the experiments can last days if not weeks – even on big clusters of GPUs.

My research is focused on visual scene understanding. My claim is that understanding a visual scene – be it an image or a video – requires to capture its semantic, and that this has to be done by building an incremental representation of the context while processing the elements of the observed environment scene. For this reason I decided to focus on Recurrent Neural Networks (RNNs), a family of neural networks with memory – or state – that can decide autonomously when to store, retrieve or delete information from their memory.

As a first step to address the problem of image understanding, I focused on object classification, i.e., the problem of selecting the class an object in a scene belongs to. Historically, this problem was addressed by hand-engineering global and local descriptors as characteristic as possible, so that their presence or absence could be used as a proxy for the presence or absence of a specific class of objects. From 2012 onwards, handcrafted methods were abandoned in favour of convolutional neural networks (CNNs), after the CNN-based model presented in Krizhevsky *et al.* (2012a) improved the state of the art by 10%. Since then CNNs-based models dominated the object classification panorama.

In Visin *et al.* (2015) my co-authors and I presented ReNet, an alternative to the ubiquitous CNNs for object classification. Our model is based on 4 RNNs that scan the image in 4 directions. RNNs have the potential to store in their memory any information that is relevant to retain the context of the part of the image they have seen up to that moment. The first two RNNs scan each line of the image reading one pixel (or patch, depending on the configuration) at the time from left to right and from right to left, respectively. The two resulting feature maps (i.e., output of each of these RNNs) are concatenated in each position over the channel axis, yielding a composite feature map where each position has information on the context of the full row, as in each position it is a concatenation of an RNN reaching the position from the right and of an RNN reaching the same position from the left. The second two RNNs sweep over the composite feature map vertically, top-down and bottom-up respectively. By reading the composite feature map, each RNN has access in each position to a "summary" of the corresponding row. Once again, the two feature maps are concatenated, resulting in a final feature map where each position is specific to a pixel (or patch) of the image but has information on the full image. The ReNet architecture allow us to capture the full context of the image with just one layer (to be fair, two sublayers), as opposed to CNN based architectures that would need many layers to span the entire image. As usual, it is still possible to stack multiple ReNet layers to increase the capacity of the network. ReNet obtained comparable results to the CNN state of the art on three widely used datasets.

Encouraged by the results of ReNet and the positive feedback from the scientific community, I worked on a second model based on ReNet, to perform fine-grained Object Segmentation (i.e., to classify each pixel of the image as belonging to a specific class). Being able to classify objects without losing information on their position in the image can be exploited to allow very precise pixel-level Object Localization, which is essential to many applications and to a proper understanding of the image.

ReSeg (Visin *et al.*, 2016) takes advantage of the inner structure of the ReNet layers that, in contrast to classical convolutional models, allow to propagate the information through several layers of computation retaining the topological structure of the input. To speed up training the image is first preprocessed with a CNN pretrained on big datasets for object classification, to extract meaningful features and exploit the extra training data. Those rich features are then processed by several ReNet layers. This results though in an intermediate feature map that has a smaller resolution than the image. To be able to classify each pixel, the original resolution has to be recovered. To this aim, one or more transposed convolutional layers (Dumoulin and Visin, 2016) upsample the feature map to the desired size. This model obtained state of the art results on three datasets and won the best paper award at the DeepVision Workshop at CVPR 2016.

The natural next step in the direction of visual scene understanding is the processing of videos, to exploit the temporal correlation between frames and improve the performance of the algorithm. It is not trivial to work with videos in the domain of semantic segmentation: big enough dataset are still lacking due to the very high cost of labelling each pixel of each frame of a video; in many cases labels are imprecise and noisy, or missing a well defined semantic (e.g. "porous" or "vertical mix"), which makes learning harder. Still, it is a challenging but important problem to tackle and there seems to be room for improvement w.r.t. the current state of the art. The proposed model combines the benefits of CNNs – namely the exploitation of the topological structure in the images and the processing speed – and the ability to retain temporal and context information of RNNs. The paper builds on Xingjian *et al.* (2015) that introduced an RNN whose internal state is convolutional. This idea is improved by stacking several convolutions inside the RNN state (as opposed to only one) and by introducing a *deconvolutional RNN*, whose state is a stack of multiple transposed convolutions. This model achieved so far state of the art results on two datasets and encouraging results on a last one.

The rest of this manuscript is organized as follows: Chapter 2 introduces the most important models and concepts needed to understand the work done; Chapter 3 introduces the problem of object classification and describes in detail the ReNet model and its results; Chapter 4 defines what is referred to as semantic segmentation and how ReSeg tackles that problem. Finally, Chapter 5 moves to video understanding and specifically video semantic segmentation and highlights the advantages of convolutional-deconvolutional RNNs in this context. In Chapter 6 summarizes the main contribution of this research and proposes some of the many possible future directions of research that can build on top of this work.

CHAPTER 2

Background

Artificial intelligence (AI) is a broad field that aims to develop intelligent software that can e.g., acquire knowledge from its interaction with the world, find optimized strategies for problem solving, automate tasks, detect patterns in audio, video and textual data, play games, drive cars and much more.

Machine learning is a complex subfield of AI that witnessed a very quick expansion in the recent years. It aims to enable computers to *learn* how to tackle problems by detecting patterns and regularities in the training data and trying to generalize this extracted knowledge to new, unseen data.

Among its many powerful tools, artificial neural networks are models that take inspiration from what is known about the human brain, by mimicking its connectivity patterns, learning rules and signals propagation, under the constraints imposed by out limited knowledge of the brain and a less powerful hardware.

While it is beyond the scope of this document to give a formal and in-depth introduction to every concept needed to fully comprehend Machine Learning, the following sections will introduce Artificial neural networks, with a specific focus on two of the most used kinds of neural networks. The interested reader can find a more detailed overview of Machine Learning in Bishop (2006); Bengio and Courville (2016)

2.1 Artificial neural networks

Brains are composed by a large amount of simple elements, called neurons, that are highly interconnected. The number of neurons in the human brain is estimated to be around 10^{11},

each one connected to a little less than 10^4 other neurons, for a total between 10^{14} and 10^{15} synapses Drachman (2005). The activity of each of these either excites or inhibits the surrounding neurons it is connected to, generating a complex network of interactions.

Artificial Neural Networks (ANNs) take inspiration from this understanding of the human brain, building networks composed of many artificial neurons, small elements that perform very simple operations on their inputs.

2.1.1 Brief history of neural networks

In 1943 McCulloch and Pitts defined a mathematical model of how a biological neuron works (McCulloch and Pitts, 1943). The artificial neuron they proposed was able to solve simple binary problems, but did not learn. In 1949 Hebb (1949) suggested that human learn by enhancing the neural pathways between neurons that collaborate, and weakening the others. Only decades later this learning rule inspired the Perceptron (see Figure 2.1), the first ANN that was able to vary its own weights, i.e. to find the setting that allowed it to exhibit the desired behavior (Rosenblatt, 1957).

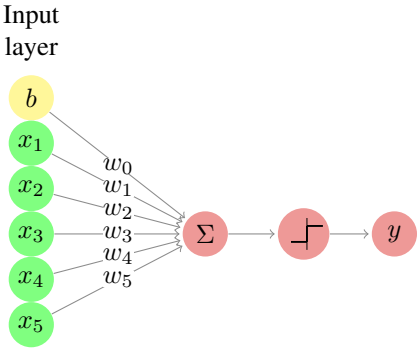


Figure 2.1: *A representation of the Perceptron.*

The activation rule of the perceptron is very simple and is at the base of many modern neural networks. Given an n-dimensional input $\mathbf{x}$, the weighted sum of each dimension of the input x_i and its corresponding weight w_i – often referred to as *preactivation* – is computed as

$$z = \sum_{i=0}^{n} (w_i \cdot x_i)$$

where to simplify the notation the bias term b has been replaced by an equivalent input term $x_0 = b$. Note that this is simply the dot product of the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$. The result of this first affine transformation is then passed through a step function of form

$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

that determines the binary output of the Perceptron.

This can be used, e.g., to classify whether the input belongs to a specific class or not. Note that the model can be easily extended to handle multiple classes, by simply adding more dimensions to y and assigning each of them to a different class.

The behavior of the Perceptron is indeed remarkable, but the biggest innovation of Rosenblatt (1957) is most probably the update algorithm that allowed to modify the weights of the model in a Hebbian rule fashion. The weights and the bias – also called the *parameters* of the network – are *learned* according to the following rule:

$$w_i^{\text{new}} = w_i^{\text{old}} + \eta \cdot (\hat{y} - y) \cdot x_i \tag{2.1}$$

where $\hat{y}$ is the target (i.e. desired) value, y is the output of the Perceptron, x_i^t and w_i^t are respectively the i-th input and weight at time t and η is a scaling factor that allows to adjust the magnitude by which the weights are modified.

The introduction of a model that could learn from the data was welcomed with excitement as the beginning of a new era and research in ANNs became very active for approximately a decade, until in 1969 Minsky and Papert published a detailed mathematical analysis of the Perceptron, demonstrating that a single layered Perceptron could not model basic operations like the XOR logic operation (Minsky and Papert, 1969). The limit of Perceptrons is that they can only solve linearly separable problems, and fail at tackling nonlinearly separable problems like the XOR (see Figure 2.2). This is not the case for MultiLayer Perceptrons (MLP, depicted in Figure 2.3), an evolution of Perceptrons that introduces one or more intermediate layers (called *hidden layers*) between the input and the output. Since each of these layers is followed by a nonlinearity, the result is a nonlinear transformation that projects the input into a linearly separable space.

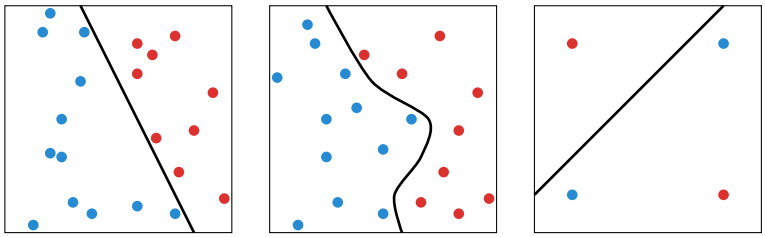


Figure 2.2: *a) A linearly separable problem. b) A nonlinearly separable problem. c) The XOR problem with a tentative solution that fails at separating the points of the space.*

The excessive enthusiasm for the early successes of ANNs turned into strong disappointment: even if Minsky and Papert (1969) showed that an MLP could model the XOR bitwise operation, it also pointed out that Rosenblatt's learning algorithm was limited to single layered Perceptrons and could not autonomously learn how to solve the problem. The expectation of an artificial intelligence that could learn by itself to solve problems and interact with humans appeared suddenly unrealistic and most of the research community lost interest in ANNs. The field experienced a severe slow down and most of the fundings were cut.

After a decade known as the AI Winter, in 1982 John Hopfield presented a model of the human memory that did not only give insights on how the brain works, but was also

useful in practical applications and had a sound and detail mathematical grounding. At the same time at the US-Japan Joint Conference on Cooperative/Competitive Neural Networks Japan announced a renewed effort in building Neural Networks and the fear that the US might be left behind renewed their effort on this topic.

The breakthrough that completely restored the interest in the field came in 1986, when Rumelhart *et al.* (1986a) rediscovered the backpropagation algorithm (Linnainmaa, 1970; Werbos, 1974) that allowed to train ANNs composed by multiple layers by performing gradient descent (see Section 2.1.4). Since then ANNs have been constantly focus of study and innovation and established the state of the art in several domains.

2.1.2 MultiLayer Perceptron

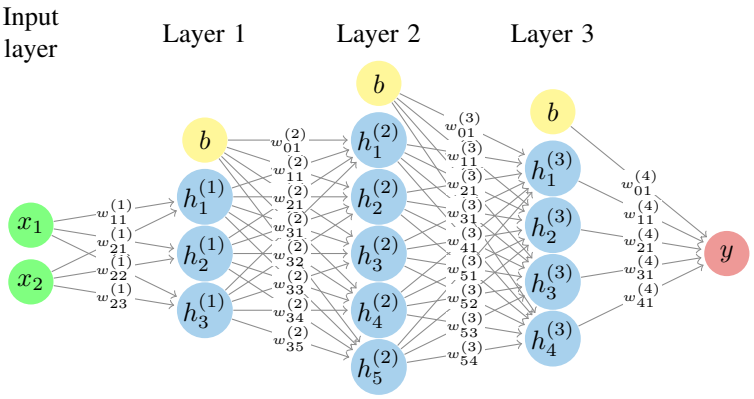


Figure 2.3: *A MultiLayer Perceptron. The sum and the nonlinearity nodes have been omitted for the sake of clarity.*

Consider the network in Figure 2.3. As opposed to the Perceptron, the MLP has multiple hidden layers, where each neuron of one hidden layer is connected to all the neurons of the previous and next layer. Each connection from the i-th neuron of layer $l-1$ to the j-th neuron of layer (l) is associated to a weight $w_{ij}^{(l)}$, and all the weights are stored in a matrix $\mathbf{W}$.

Similarly to the Perceptron case, each layer computes an affine transformation

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)} \cdot \mathbf{a}^{(l-1)} \tag{2.2}$$

followed by a nonlinearity – usually more complex than the one used in the Perceptron – called *activation function*

$$\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) \tag{2.3}$$

One hidden layer can thus be seen as a function $h^{(l)}$ that depends on some *parameters* – namely the vector of weights $\mathbf{w}^{(l)}$ and the bias of the layer $b^{(l)}$ – as well as some *hyperparameters* such as, e.g., the number of neurons and the choice of activation function. It is common to refer to the set of trainable parameters that fully characterize a layer as *sufficient statistics*, usually denoted as $\boldsymbol{\theta}$.

The processing performed by the hierarchy of layers of the MLP results in the output vector $\mathbf{y}$ and is equivalent to a composition of multiple functions

$$\mathbf{y} = h_{\boldsymbol{\theta}}^{(L)} \circ h_{\boldsymbol{\theta}}^{(L-1)} \circ \ldots \circ h_{\boldsymbol{\theta}}^{(1)} \tag{2.4}$$

Although very common and widely used, the MLP is not the only architecture for ANNs. In Section 2.2 and Section 2.3 two of the most used alternatives will be described. Generally, each layer of an ANN computes some activation based on its input and a non-linear activation function. The choice of which activation function to use in each layer can have a big impact on the performance of the model and is sometimes constrained by the semantic assigned to the output of some units. The most important activation functions will be introduced in the following section.

2.1.3 Activation functions

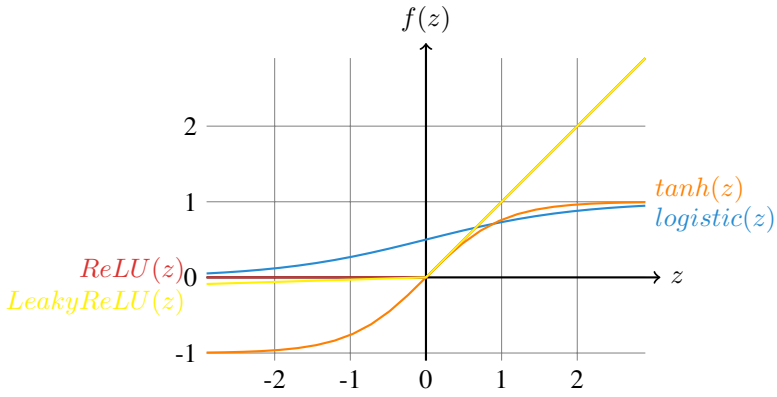


Figure 2.4: *Some of the most common activation functions: sigmoid, tanh, ReLU and Leaky ReLU*

The activation function is one of the most important component of an ANN. As explained in the previous sections, to tackle nonlinearly separable problems it is imperative to map the input into a space that is linearly separable. The activation function does this by performing an *element-wise nonlinear transformation* of the pre-activation that comes from the affine transformation.

The affine transformation and the nonlinearity work together in a very tight interaction: the latter is fixed and does not evolve during training, but is a powerful transformation; the former instead, is determined by the weights that are learned during training and exploits the latter to map the incoming activation into a new space where they are easier to separate.

In addition to this it is interesting to point out that if there was no activation function, since the composition of multiple affine transformation is an affine transformation, the layers of the MLP could be replaced by a single equivalent layer, and the MLP would become a Perceptron.

Many activation functions have been proposed in the years and even if our understanding of this component has improved, which one to use with the different architectures and

tasks is still object of active debate and sometimes a matter of personal preference.

Logistic

The logistic, often called *sigmoid*, is a differentiable monotonically increasing function that takes any real-valued number and maps it to $[0, 1]$. As evident from its representation in Figure 2.4, for large negative numbers it asymptotes to 0 while for large positive numbers it asymptotes to 1. It is defined as

$$logistic(\mathbf{z}) = \frac{1}{1 + \exp(-\mathbf{z})} \tag{2.5}$$

The logistic function has probably been the most used nonlinearity historically due to its possible interpretation as the firing rate of a neuron given its potential (i.e. the level of excitement provoked by its incoming spikes): when the potential is low the neuron fires less often whereas when the potential is high the frequency of the spikes increases.

Another very important property of the logistic function is that it is very fast to compute its derivative, once solved analytically:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{z}} logistic(\mathbf{z}) &= \frac{\exp(-\mathbf{z})}{\left(1 + \exp(-\mathbf{z})\right)^2} \\
&= \frac{1}{1 + \exp(-\mathbf{z})} \cdot \frac{\exp(-\mathbf{z})}{1 + \exp(-\mathbf{z})} \\
&= logistic(\mathbf{z}) \cdot \frac{\exp(-\mathbf{z})}{1 + \exp(-\mathbf{z})} \\
&= logistic(\mathbf{z}) \cdot \frac{1 + \exp(-\mathbf{z}) - 1}{1 + \exp(-\mathbf{z})} \\
&= logistic(\mathbf{z}) \cdot \left(1 - \frac{1}{1 + \exp(-\mathbf{z})}\right) \\
&= logistic(\mathbf{z}) \cdot (1 - logistic(\mathbf{z}))
\end{aligned} \tag{2.6}$$

Its use is not as widespread as it used to be though, due to two major drawbacks:

- *Saturation kills the gradient:* backpropagation Section 2.1.4 relies on the gradient of the error to determine the parameter update. The logistic function saturates at both ends, resulting in a very small or zero gradient. This problem – often referred to as *vanishing gradient* – makes training very slow or prevents it in some cases. This also makes the logistic units very sensitive to the initialization of the weights of the network, that ideally should be such that the initial input of the logistic function is close to zero.

- *The output is not zero-centered:* the dynamics of ANNs are usually complex and difficult to inspect, but it is widely believed that normalizing (i.e. zero-centered with unit variance) the intermediate activations of the network helps training (Ioffe and Szegedy (2015); Laurent *et al.* (2015); Arpit *et al.* (2016); Cooijmans *et al.* (2016)). The output of the logistic function is always positive, which causes the mean activation to be non-zero. This could introduce undesirable dynamics that could slow down or prevent training.

Hyperbolic tangent (tanh)

The hyperbolic tangent, typically shortened as *tanh*, is a differentiable monotonically increasing function that maps any real-valued number to $[-1, 1]$. This nonlinear function suffers from the same vanishing problem as the logistic, but its mean is centered in zero. Furthermore, the tanh is often chosen where it is desirable to be able to increase or decrease some quantity by small amounts, thanks to its codomain. It is defined as

$$tanh(\mathbf{z}) = \frac{1 - exp(-2\mathbf{z})}{1 + exp(-2\mathbf{z})} \tag{2.7}$$

Rectified Linear Unit (ReLU)

Since its introduction, the Rectified Linear Unit (ReLu) Jarrett *et al.* (2009); Nair and Hinton (2010) has become the nonlinearity of choice in many applications Krizhevsky *et al.* (2012a); LeCun *et al.* (2015); Glorot *et al.* (2011). It is defined as

$$relu(\mathbf{z}) = max(0, \mathbf{z}) \tag{2.8}$$

Although very simple, it has some very interesting properties and a few drawbacks:

- *No positive saturation:* the ReLU is zero for non-positive inputs, but does not saturate otherwise. This ensures a flow of gradient whenever the input is positive that was found to significantly speed up the convergence of training.
- *Cheap to compute:* as opposed to many other activation functions that require expensive operations, such as e.g. exponentials, ReLU's implementation simply amounts to thresholding at zero. Another important characteristic is that the gradient is trivial to compute:

$$\nabla(relu(\mathbf{z}^{(l)})) = \begin{cases} \mathbf{a}^{(l-1)}, & \text{if } \mathbf{z}^{(l)} > 0 \\ 0, & \text{if } \mathbf{z}^{(l)} < 0 \\ undefined, & \text{if } \mathbf{z}^{(l)} = 0 \end{cases} \tag{2.9}$$

- *Induce sparsity:* ReLU units induce sparsity, since whenever the input preactivation is negative their activation is zero. Sparsity is usually a desired property: as opposed to dense encoding, sparsity will produce representations where only a few entries will change upon small variations of the input, i.e. will produce a representation that is more consistent and robust to perturbations. Furthermore, sparsity allows for a more compact encoding, which is desirable in many contexts such as, e.g., data compression and efficient data transfer. Finally, it is also usually easier to linearly separate sparse representations (Glorot *et al.*, 2011).
- *ReLU units can die:* when a large gradient flows through a ReLU unit it can change its weights in such a way that will prevent it from ever being active again. In this case every input will put the ReLU unit on the flat zero side. This will prevent any gradient flow and the unit will never leave this state becoming *de facto* "dead". This can be alleviated using a lower learning rate or choosing some modification of ReLU less sensitive to this problem.

Leaky Rectified Linear Unit (Leaky ReLU)

Leaky ReLUs are one of the most adopted alternatives to ReLUs. They have been proposed as a way to alleviate the dying units problem of ReLUs, by preventing the unit from saturating allowing a small gradient to always flow through the unit, potentially recovering extreme values of the weights. Leaky ReLUs are defined as

$$leaky\_relu(\mathbf{z}) = max(\beta * \mathbf{z}, \mathbf{z}) \tag{2.10}$$

Softmax

One peculiar nonlinearity that deserves a particular mention is the softmax. This function differs from the previously described activations in that it does not only depend on the value of one dimension (or neuron), but rather on that of all the dimensions (or neurons) in the layer. The softmax is a *squashing function* that maps its input to a categorical distribution. It is defined as

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_{k=0}^{K} \exp(z_k)} \tag{2.11}$$

where K is the number of classes, i.e. of dimensions (or neurons).

Note that it is possible to add a temperature parameter T to the softmax that allows to control its steepness (see Figure 2.5), i.e. to control the randomness of predictions. When T is high the distribution over the classes will be flatter - with the extreme case of $T = \inf$, where all the classes have equal probability. When T is low, the probability curve is pickier on the class with higher probability and has a light tail on the other classes.

$$softmax(z_i) = \frac{\exp(z_i/T)}{\sum_{k=0}^{K} \exp(z_k/T)} \tag{2.12}$$

2.1.4 Backpropagation

The learning rule introduced with Perceptrons does not allow to train models with multiple layers (i.e. with *hidden* layers), such as the one depicted in Figure 2.3.

This is not possible due to the fact that to compute the variation of the weights of a layer with Equation 2.1 it is necessary to know the correct value of its output, which is given only for the last layer.

To address this apparently insurmountable obstacle it suffices to notice that the computation performed by the activation function is a nonlinear but differentiable function of the inputs. This allows for computing the partial derivatives of the error (e.g. the expected value of the quadratic loss) , w.r.t. the weights of the network. In other words, it is possible to use calculus to determine the amount by which each neuron of the last layer contributed to causing the error, and then further split the responsibility of each of them among the ones of the preceding layer. This way the error can be *backpropagated* through the layers of the network, assigning to each weight its amount of blame. This information can be used by an *optimization algorithm* to iteratively change the weights to minimize the error.

The backpropagation algorithm has some resemblance with the learning rule of the Perceptron (Equation 2.1). The main idea in that case was to modify each weight of the

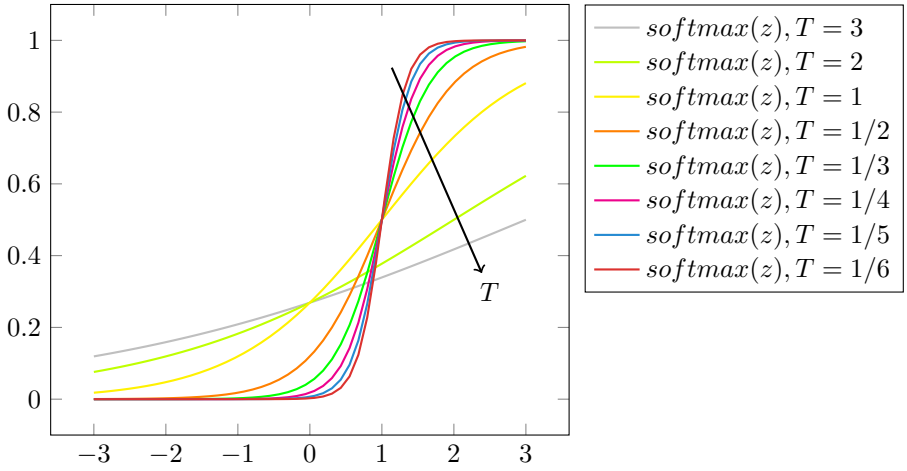


Figure 2.5: *The behaviour of softmax as temperature T grows. The plots have been obtained considering a bidimensional input setting where the preactivation associated to the second class z_1 is always 1. As T increases, the function becomes steeper.*

network by a factor proportional to the error ($E = \hat{y} - y$, in the Perceptron) and to the input. Even if in MLPs it is usually common to consider other kinds of errors, the same concept applies: the learning procedure tries to modify the weights in order to minimize some error

$$\mathbf{W} = \mathbf{W} + \eta \frac{\partial E}{\partial \mathbf{W}} \tag{2.13}$$

where η is a scaling factor typically referred to as *learning rate*, that determines the size of the gradient descent steps.

It is easy to understand backpropagation with a practical example. Consider once again Figure 2.3: the network processes a bidimensional input $\mathbf{x}$ and, after three layers of affine transformations followed by a nonlinearity, returns a unidimensional value $\mathbf{y}$. The correct output $\hat{\mathbf{y}}$ is given and is used to compute the error, or *cost*, with some *differentiable* metric. For this example consider e.g. the mean squared error (MSE)

$$E_{mse} = \frac{1}{M} \sum_{\mathcal{D}} \frac{1}{2} (\hat{\mathbf{y}} - \mathbf{y})^2 \tag{2.14}$$

the summation is done over a dataset $\mathcal{D}$ of M samples, each composed by an input $\mathbf{x}$ and its associated desired output $\hat{\mathbf{y}}$. $\mathbf{y}$ is used as a compact notation for $\mathbf{y}(\mathbf{x})$ and represents the output of the network for one input sample $\mathbf{x}$.

It is possible to compute the fraction of the error that is imputable to each neuron of the network by taking the derivative of the error w.r.t. to its weights

$$\begin{aligned}\frac{\partial E_{mse}}{\partial w_{ij}^{(l)}} &= \frac{\partial}{\partial w_{ij}^{(l)}}\left(\frac{1}{m}\sum_{\mathcal{D}}\left[\frac{1}{2}(\hat{\mathbf{y}}-\mathbf{y})^2\right]\right) \\ &= \frac{1}{m}\sum_{\mathcal{D}}\left[\frac{1}{2}\frac{\partial}{\partial w_{ij}^{(l)}}\left(\hat{\mathbf{y}}-\mathbf{y}\right)^2\right] \\ &= \frac{1}{m}\sum_{\mathcal{D}}\left[(\hat{\mathbf{y}}-\mathbf{y})\cdot\frac{\partial}{\partial w_{ij}^{(l)}}(-\mathbf{y})\right]\end{aligned} \tag{2.15}$$

Backpropagation allows to compute the partial derivative $\frac{\partial}{\partial w_{ij}^{(l)}}(-\mathbf{y})$ exploiting the chain rule of derivation. Consider e.g. the weights of the second layer $\mathbf{W}^{(2)}$

$$\begin{aligned}-\frac{\partial \mathbf{y}}{\partial \mathbf{W}^{(2)}} &= -\frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(4)}}\cdot\frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{W}^{(2)}} \\ &= -\frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(4)}}\cdot\frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{a}^{(3)}}\cdot\frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{W}^{(2)}} \\ &= -\frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(4)}}\cdot\frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{a}^{(3)}}\cdot\frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{z}^{(3)}}\cdot\frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{W}^{(2)}} \\ &= -\frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(4)}}\cdot\frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{a}^{(3)}}\cdot\frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{z}^{(3)}}\cdot\frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{a}^{(2)}}\cdot\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{W}^{(2)}} \\ &= -\frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(4)}}\cdot\frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{a}^{(3)}}\cdot\frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{z}^{(3)}}\cdot\frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{a}^{(2)}}\cdot\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}}\cdot\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}}\end{aligned} \tag{2.16}$$

From Equation 2.2 and Equation 2.3 follows

$$-\frac{\partial \mathbf{y}}{\partial \mathbf{W}^{(2)}} = -\sigma'(\mathbf{z}^{(4)})\cdot\mathbf{W}^{(4)}\cdot\sigma'(\mathbf{z}^{(3)})\cdot\mathbf{W}^{(3)}\cdot\sigma'(\mathbf{z}^{(2)})\cdot\mathbf{a}^{(1)} \tag{2.17}$$

where σ' can be computed analytically and depends on the activation function of choice (see e.g. Equation 2.6 and Equation 2.9)

2.2 Convolutional networks

Deep CNNs have been at the heart of spectacular advances in deep learning. Although CNNs have been used as early as the nineties to solve character recognition tasks (Le Cun *et al.*, 1997), their current widespread application is due to much more recent work, when a deep CNN was used to beat state-of-the-art in the ImageNet image classification challenge (Krizhevsky *et al.*, 2012b).

Section 2.1.2 introduced MLPs, a powerful and very common kind of ANN that computes an affine transformation of its inputs followed by an activation function. One property of MLPs is that they are dense, in the sense that they connect all the units of one layer to all the units of the next layer (often referred to as being *fully connected*. When the input has a structure, a dense connectivity pattern might be wasteful and it is usually preferable to be able to exploit the data structure. The reason for this is twofold: first, adapting

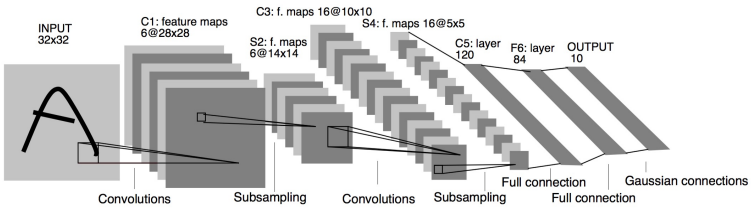


Figure 2.6: *The LeNet architecture for handwritten characters recognition.*

the connectivity pattern to the structure of the data reduces the number of operations performed by the network and, consequently, the computation time; second, constraining the connectivity pattern has the effect of forcing the network to focus on what is important, yielding faster training and better performance.

Convolutional neural networks (CNNs) are an example of model that exploits the structure of the data. The intuition behind CNNs is that many kinds of data – such as e.g. images or audio clips – have a *local* topological structure that does not depend on the specific location in the global reference system. In the case of images, for example, this means that a hand can appear in the center of the image as well as in one of the corners, and is not less of a hand in either cases. Similarly, the sound of the word *hand* should be detected the same way when pronounced with a high or low pitch.

CNNs exploit this understanding of the data by applying the same pattern detector on many locations of the image. This is formally done through a *convolution* (hence the name CNN), a signal processing operation that superimposes a pattern detector – usually called *filter*, *kernel* or *mask* – on different locations of the image and emits an activation in each position. In the previous example this would mean applying a "hand detector" on every location of the image to produce a matrix of activations, typically referred to as *feature map*. In any real-world application though, it would be common to apply multiple kernels at once with the same convolution, hence obtaining a tensor of feature maps.

The convolution operation can be seen as the repetition of two operations: the superimposition of one or more kernels, followed by a shift – or *stride* – of the kernel to allow the subsequent applications of the filter. This system would work successfully in most cases, but it would fail to detect hands that are not completely contained in the image. The typical solution adopted to overcome this limitation is to *pad* the image by adding a frame (usually of zeros) around it on every side. This ensures that on the borders of the image the convolution performs both a complete superimposition of the kernel on the image as well as a partial one.

The elements described so far fully define a convolution. The shape of the feature maps produced by a convolutional layer is affected by the shape of its input as well as the choice of kernel shape, zero padding and strides, and the relationship between these properties is not always trivial to infer. This contrasts with fully-connected layers, whose output size is independent of the input size.

Additionally, CNNs also usually feature *pooling* layers, adding yet another level of complexity with respect to fully-connected networks. Finally, so-called transposed convolutional layers (also known as fractionally strided convolutional layers) have been employed in more and more work as of late (Zeiler *et al.*, 2011b; Zeiler and Fergus, 2014a;

Long *et al.*, 2015a; Radford *et al.*, 2015; Visin *et al.*, 2016; Im *et al.*, 2016), and their relationship with convolutional layers has been explained with various degrees of clarity.

The convolution operation will be formally introduced in Section 2.2.1 followed by a description of pooling in Section 2.2.2. The rest of this section will focus on the arithmetic to compute the output shape of a convolution given its parameters. In particular Section 2.2.7 targets transposed convolutions, a smart application of convolutions to upsampling. For an in-depth treatment of the subject, see Chapter 9 of the Deep Learning textbook (Bengio and Courville, 2016).

2.2.1 Discrete convolutions

The bread and butter of neural networks is *affine transformations*: a vector is received as input and is multiplied with a matrix to produce an output (to which a bias vector is usually added before passing the result through a nonlinearity). This is applicable to any type of input, be it an image, a sound clip or an unordered collection of features: whatever their dimensionality, their representation can always be flattened into a vector before the transformation.

Images, sound clips and many other similar kinds of data have an intrinsic structure. More formally, they share these important properties:

- They are stored as multi-dimensional arrays.
- They feature one or more axes for which ordering matters (e.g., width and height axes for an image, time axis for a sound clip).
- One axis, called the channel axis, is used to access different views of the data (e.g., the red, green and blue channels of a color image, or the left and right channels of a stereo audio track).

These properties are not exploited when an affine transformation is applied; in fact, all the axes are treated in the same way and the topological information is not taken into account. Still, taking advantage of the implicit structure of the data may prove very handy in solving some tasks, like computer vision and speech recognition, and in these cases it would be best to preserve it. This is where discrete convolutions come into play.

A discrete convolution is a linear transformation that preserves this notion of ordering. It is sparse (only a few input units contribute to a given output unit) and reuses parameters (the same weights are applied to multiple locations in the input).

Figure 2.7 provides an example of a discrete convolution. The light blue grid is called the *input feature map*. To keep the drawing simple, a single input feature map is represented, but it is not uncommon to have multiple feature maps stacked one onto another.[1] A *kernel* (shaded area) of value

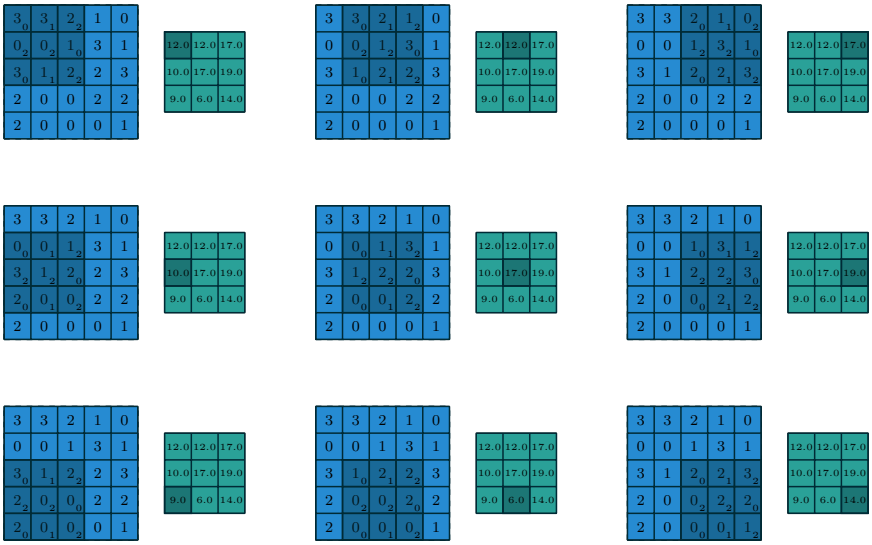


Figure 2.7: *Computing the output values of a discrete convolution.*

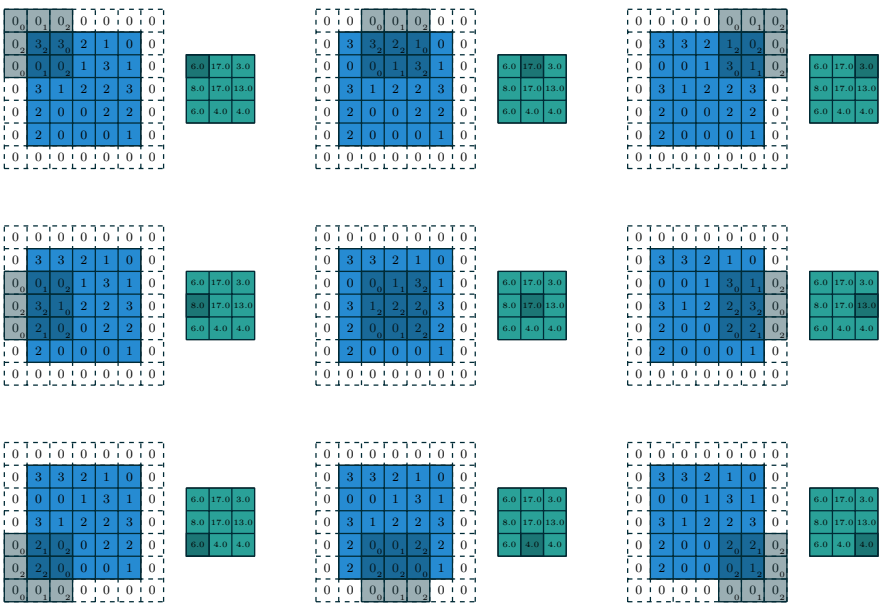


Figure 2.8: *Computing the output values of a discrete convolution for* $N = 2$*,* $i_1 = i_2 = 5$*,* $k_1 = k_2 = 3$*,* $s_1 = s_2 = 2$*, and* $p_1 = p_2 = 1$*.*

slides across the input feature map. At each location, the product between each element of the kernel and the input element it overlaps is computed and the results are summed up to obtain the output in the current location. The procedure can be repeated using different kernels to form as many output feature maps as desired (Figure 2.9). The final outputs of this procedure are called *output feature maps*.[2] If there are multiple input feature maps, the kernel will have to be 3-dimensional – or, equivalently each one of the feature maps will be convolved with a distinct kernel – and the resulting feature maps will be summed up elementwise to produce the output feature map.

The convolution depicted in Figure 2.7 is an instance of a 2-D convolution, but it can be generalized to N-D convolutions. For instance, in a 3-D convolution, the kernel would be a *cuboid* and would slide across the height, width and depth of the input feature map.

The collection of kernels defining a discrete convolution has a shape corresponding to some permutation of $(n, m, k_1, \ldots, k_N)$, where

$$
\begin{aligned}
n &\equiv \text{number of output feature maps,} \\
m &\equiv \text{number of input feature maps,} \\
k_j &\equiv \text{kernel size along axis } j.
\end{aligned}
$$

The following properties affect the output size o_j of a convolutional layer along axis j:

- i_j: input size along axis j,
- k_j: kernel size along axis j,
- s_j: stride (distance between two consecutive positions of the kernel) along axis j,
- p_j: zero padding (number of zeros concatenated at the beginning and at the end of an axis) along axis j.

For instance, Figure 2.8 shows a 3×3 kernel applied to a 5×5 input padded with a 1×1 border of zeros using 2×2 strides.

Note that strides constitute a form of *subsampling*. As an alternative to being interpreted as a measure of how much the kernel is translated, strides can also be viewed as how much of the output is retained. For instance, moving the kernel by hops of two is equivalent to moving the kernel by hops of one but retaining only odd output elements (Figure 2.10).

2.2.2 Pooling

In addition to discrete convolutions themselves, *pooling* operations make up another important building block in CNNs. Pooling operations reduce the size of feature maps by using some function to summarize subregions, such as taking the average or the maximum value.

[1] An example of this is what was referred to earlier as *channels* for images and sound clips.

[2] While there is a distinction between convolution and cross-correlation from a signal processing perspective, the two become interchangeable when the kernel is learned. For the sake of simplicity and to stay consistent with most of the machine learning literature, the term *convolution* will be used in this guide.

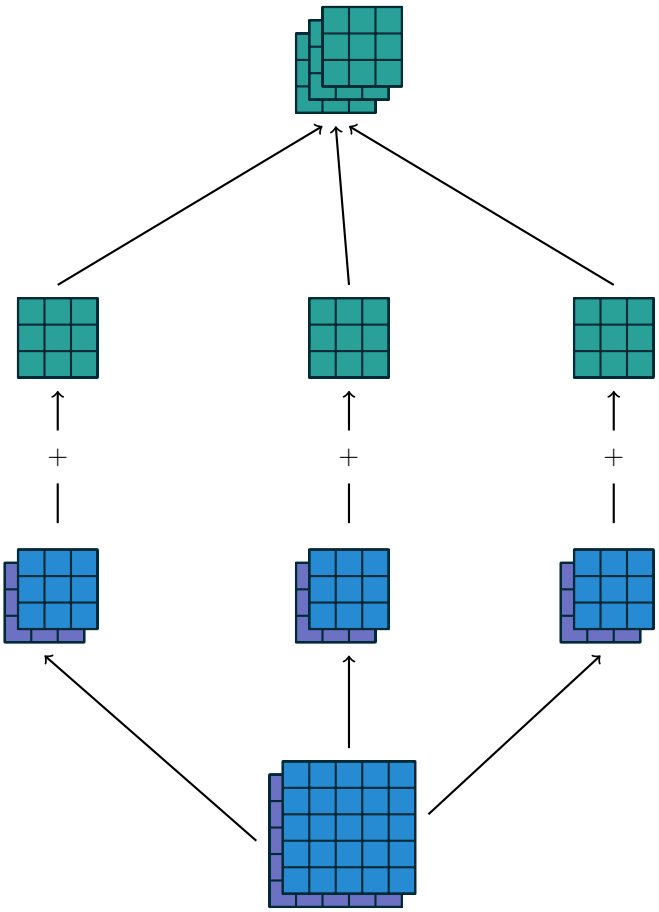


Figure 2.9: *A convolution mapping from two input feature maps to three output feature maps using a* $3 \times 2 \times 3 \times 3$ *collection of kernels* $\mathbf{w}$*. In the left pathway, input feature map 1 is convolved with kernel* $\mathbf{w}_{1,1}$ *and input feature map 2 is convolved with kernel* $\mathbf{w}_{1,2}$*, and the results are summed together elementwise to form the first output feature map. The same is repeated for the middle and right pathways to form the second and third feature maps, and all three output feature maps are grouped together to form the output.*

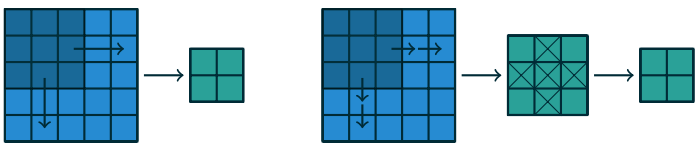


Figure 2.10: *An alternative way of viewing strides. Instead of translating the* 3×3 *kernel by increments of* $s = 2$ *(left), the kernel is translated by increments of* 1 *and only one in* $s = 2$ *output elements is retained (right).*

Pooling works by sliding a window across the input and feeding the content of the window to a *pooling function*. In some sense, pooling works very much like a discrete convolution, but replaces the linear combination described by the kernel with some other function. Figure 2.11 provides an example for average pooling, and Figure 2.12 does the same for max pooling.

The following properties affect the output size o_j of a pooling layer along axis j:

- i_j: input size along axis j,
- k_j: pooling window size along axis j,
- s_j: stride (distance between two consecutive positions of the pooling window) along axis j.

2.2.3 Convolution arithmetic

The analysis of the relationship between convolutional layer properties is eased by the fact that they don't interact across axes, i.e., the choice of kernel size, stride and zero padding along axis j only affects the output size of axis j. Because of that, this chapter will focus on the following simplified setting:

- 2-D discrete convolutions ($N = 2$),
- square inputs ($i_1 = i_2 = i$),
- square kernel size ($k_1 = k_2 = k$),
- same strides along both axes ($s_1 = s_2 = s$),
- same zero padding along both axes ($p_1 = p_2 = p$).

This facilitates the analysis and the visualization, but keep in mind that the results outlined here also generalize to the N-D and non-square cases.

No zero padding, unit strides

The simplest case to analyze is when the kernel just slides across every position of the input (i.e., $s = 1$ and $p = 0$). Figure 2.13 provides an example for $i = 4$ and $k = 3$.

One way of defining the output size in this case is by the number of possible placements of the kernel on the input. Let's consider the width axis: the kernel starts on the leftmost part of the input feature map and slides by steps of one until it touches the right side of the input. The size of the output will be equal to the number of steps made, plus one, accounting for the initial position of the kernel (Figure 2.20a). The same logic applies for the height axis.

More formally, the following relationship can be inferred:

Relationship 1. *For any i and k, and for $s = 1$ and $p = 0$,*

$$o = (i - k) + 1.$$

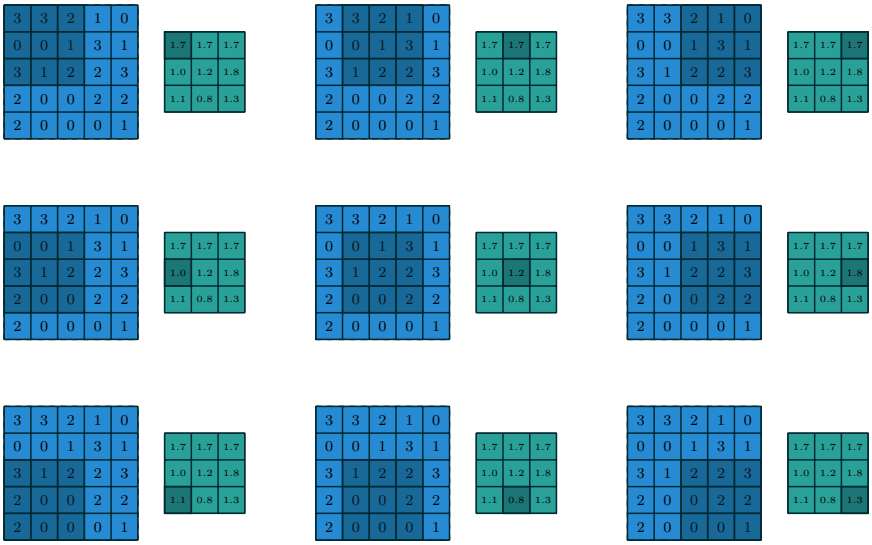


Figure 2.11: *Computing the output values of a* 3×3 *average pooling operation on a* 5×5 *input using* 1×1 *strides.*

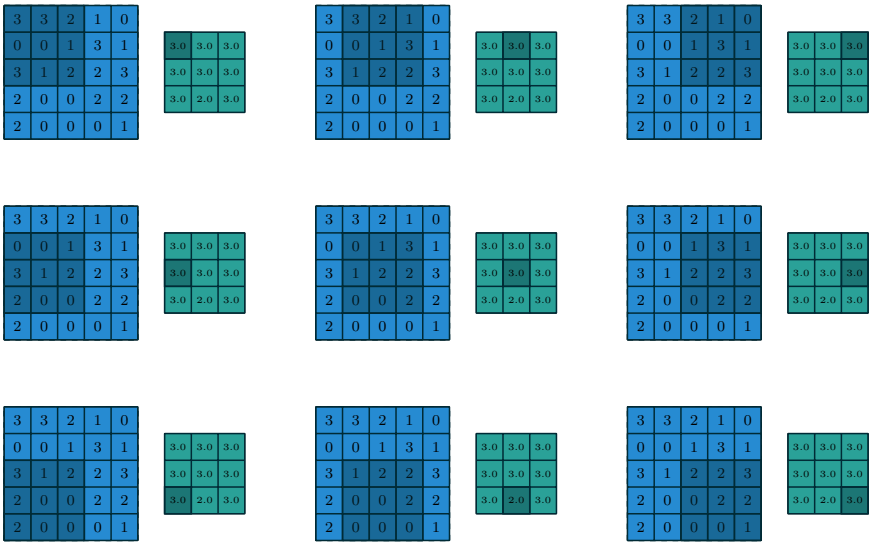


Figure 2.12: *Computing the output values of a* 3×3 *max pooling operation on a* 5×5 *input using* 1×1 *strides.*

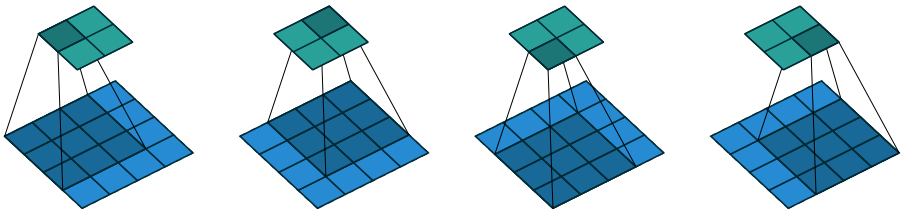


Figure 2.13: *(No padding, unit strides) Convolving a* 3×3 *kernel over a* 4×4 *input using unit strides (i.e.,* $i = 4$*,* $k = 3$*,* $s = 1$ *and* $p = 0$*).*

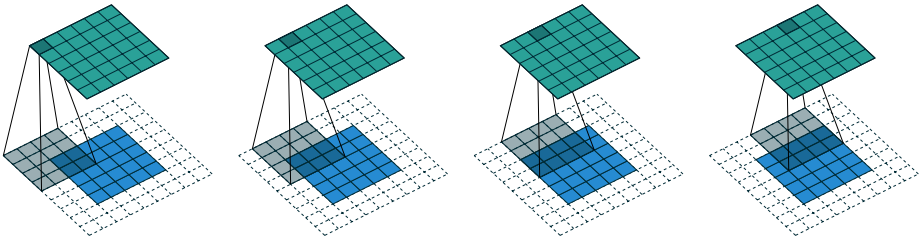


Figure 2.14: *(Arbitrary padding, unit strides) Convolving a* 4×4 *kernel over a* 5×5 *input padded with a* 2×2 *border of zeros using unit strides (i.e.,* $i = 5$*,* $k = 4$*,* $s = 1$ *and* $p = 2$*).*

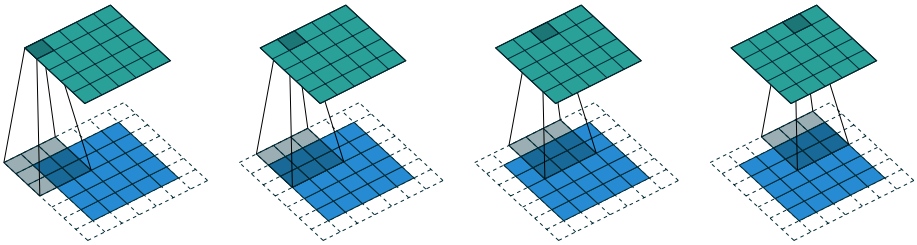


Figure 2.15: *(Half padding, unit strides) Convolving a* 3×3 *kernel over a* 5×5 *input using half padding and unit strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 1$ *and* $p = 1$*).*

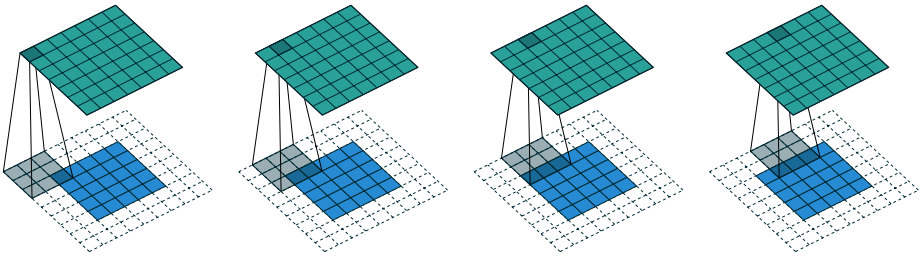


Figure 2.16: *(Full padding, unit strides) Convolving a* 3×3 *kernel over a* 5×5 *input using full padding and unit strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 1$ *and* $p = 2$*).*

Zero padding, unit strides

To factor in zero padding (i.e., only restricting to $s = 1$), let's consider its effect on the effective input size: padding with p zeros changes the effective input size from i to $i + 2p$. In the general case, Relationship 1 can then be used to infer the following relationship:

Relationship 2. *For any i, k and p, and for $s = 1$,*

$$o = (i - k) + 2p + 1.$$

Figure 2.14 provides an example for $i = 5$, $k = 4$ and $p = 2$.

In practice, two specific instances of zero padding are used quite extensively because of their respective properties. Let's discuss them in more detail.

Half (same) padding Having the output size be the same as the input size (i.e., $o = i$) can be a desirable property:

Relationship 3. *For any i and for k odd ($k = 2n + 1, \quad n \in \mathbb{N}$), $s = 1$ and $p = \lfloor k/2 \rfloor = n$,*

$$\begin{aligned} o &= i + 2\lfloor k/2 \rfloor - (k - 1) \\ &= i + 2n - 2n \\ &= i. \end{aligned}$$

This is sometimes referred to as *half* (or *same*) padding. Figure 2.15 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 1$.

Full padding While convolving a kernel generally *decreases* the output size with respect to the input size, sometimes the opposite is required. This can be achieved with proper zero padding:

Relationship 4. *For any i and k, and for $p = k - 1$ and $s = 1$,*

$$\begin{aligned} o &= i + 2(k - 1) - (k - 1) \\ &= i + (k - 1). \end{aligned}$$

This is sometimes referred to as *full* padding, because in this setting every possible partial or complete superimposition of the kernel on the input feature map is taken into account. Figure 2.16 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 2$.

2.2.4 No zero padding, non-unit strides

All relationships derived so far only apply for unit-strided convolutions. Incorporating non unitary strides requires another inference leap. To facilitate the analysis, let's momentarily ignore zero padding (i.e., $s > 1$ and $p = 0$). Figure 2.17 provides an example for $i = 5$, $k = 3$ and $s = 2$.

Once again, the output size can be defined in terms of the number of possible placements of the kernel on the input. Let's consider the width axis: the kernel starts as usual on the leftmost part of the input, but this time it slides by steps of size s until it touches the right side of the input. The size of the output is again equal to the number of steps made, plus one, accounting for the initial position of the kernel (Figure 2.20b). The same logic applies for the height axis.

From this, the following relationship can be inferred:

Relationship 5. *For any i, k and s, and for $p = 0$,*

$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1.$$

The floor function accounts for the fact that sometimes the last possible step does *not* coincide with the kernel reaching the end of the input, i.e., some input units are left out (see Figure 2.19 for an example of such a case).

2.2.5 Zero padding, non-unit strides

The most general case (convolving over a zero padded input using non-unit strides) can be derived by applying Relationship 5 on an effective input of size $i + 2p$, in analogy to what was done for Relationship 2:

Relationship 6. *For any i, k, p and s,*

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1.$$

As before, the floor function means that in some cases a convolution will produce the same output size for multiple input sizes. More specifically, if $i + 2p - k$ is a multiple of s, then any input size $j = i + a, \quad a \in \{0, \ldots, s - 1\}$ will produce the same output size. Note that this ambiguity applies only for $s > 1$.

Figure 2.18 shows an example with $i = 5$, $k = 3$, $s = 2$ and $p = 1$, while Figure 2.19 provides an example for $i = 6$, $k = 3$, $s = 2$ and $p = 1$. Interestingly, despite having different input sizes these convolutions share the same output size. While this doesn't affect the analysis for *convolutions*, this will complicate the analysis in the case of *transposed convolutions*.

2.2.6 Pooling arithmetic

In a neural network, pooling layers provide invariance to small translations of the input. The most common kind of pooling is *max pooling*, which consists in splitting the input in (usually non-overlapping) patches and outputting the maximum value of each patch. Other kinds of pooling exist, e.g., mean or average pooling, which all share the same idea of aggregating the input locally by applying a nonlinearity to the content of some patches (Boureau *et al.*, 2010a,b, 2011; Saxe *et al.*, 2011).

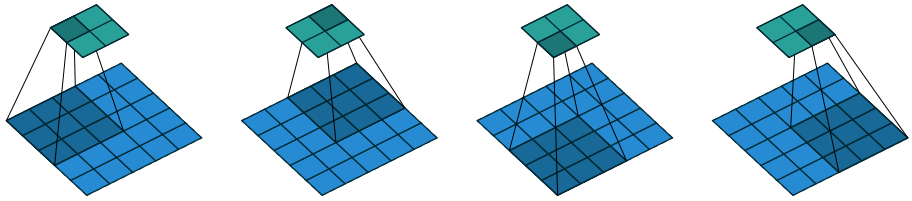


Figure 2.17: *(No zero padding, arbitrary strides) Convolving a* 3×3 *kernel over a* 5×5 *input using* 2×2 *strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 2$ *and* $p = 0$*).*

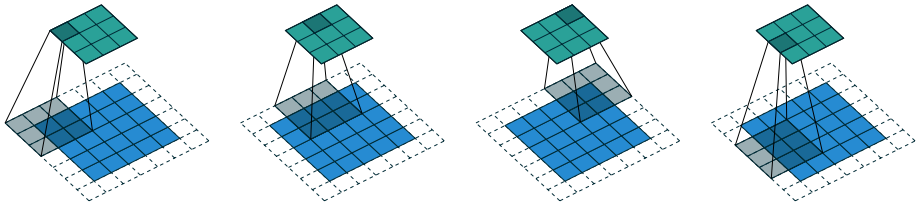


Figure 2.18: *(Arbitrary padding and strides) Convolving a* 3×3 *kernel over a* 5×5 *input padded with a* 1×1 *border of zeros using* 2×2 *strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 2$ *and* $p = 1$*).*

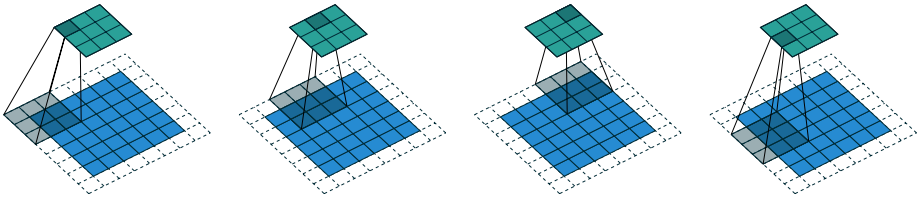
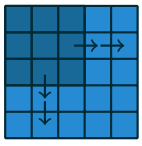
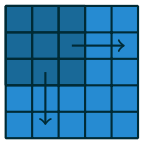


Figure 2.19: *(Arbitrary padding and strides) Convolving a* 3×3 *kernel over a* 6×6 *input padded with a* 1×1 *border of zeros using* 2×2 *strides (i.e.,* $i = 6$*,* $k = 3$*,* $s = 2$ *and* $p = 1$*). In this case, the bottom row and right column of the zero padded input are not covered by the kernel.*



(a) *The kernel has to slide two steps to the right to touch the right side of the input (and equivalently downwards). Adding one to account for the initial kernel position, the output size is* 3×3*.*



(b) *The kernel has to slide one step of size two to the right to touch the right side of the input (and equivalently downwards). Adding one to account for the initial kernel position, the output size is* 2×2*.*

Figure 2.20: *Counting kernel positions.*

Some readers may have noticed that the treatment of convolution arithmetic only relies on the assumption that some function is repeatedly applied onto subsets of the input. This means that the relationships derived in the previous chapter can be reused in the case of pooling arithmetic. Since pooling does not involve zero padding, the relationship describing the general case is as follows:

Relationship 7. *For any i, k and s,*

$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1.$$

This relationship holds for any type of pooling.

2.2.7 Transposed convolution arithmetic

The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution. For instance, one might use such a transformation as the decoding layer of a convolutional autoencoder or to project feature maps to a higher-dimensional space.

Once again, the convolutional case is considerably more complex than the fully-connected case, which only requires to use a weight matrix whose shape has been transposed. However, since every convolution boils down to an efficient implementation of a matrix operation, the insights gained from the fully-connected case are useful in solving the convolutional case.

Like for convolution arithmetic, the dissertation about transposed convolution arithmetic is simplified by the fact that transposed convolution properties don't interact across axes.

The chapter will focus on the following setting:

- 2-D transposed convolutions ($N = 2$),
- square inputs ($i_1 = i_2 = i$),
- square kernel size ($k_1 = k_2 = k$),
- same strides along both axes ($s_1 = s_2 = s$),
- same zero padding along both axes ($p_1 = p_2 = p$).

Once again, the results outlined generalize to the N-D and non-square cases.

2.2.8 Convolution as a matrix operation

Take for example the convolution represented in Figure 2.13. If the input and output were to be unrolled into vectors from left to right, top to bottom, the convolution could be

represented as a sparse matrix $\mathbf{C}$ where the non-zero elements are the elements $w_{i,j}$ of the kernel (with i and j being the row and column of the kernel respectively):

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

This linear operation takes the input matrix flattened as a 16-dimensional vector and produces a 4-dimensional vector that is later reshaped as the 2×2 output matrix.

Using this representation, the backward pass is easily obtained by transposing $\mathbf{C}$; in other words, the error is backpropagated by multiplying the loss with $\mathbf{C}^T$. This operation takes a 4-dimensional vector as input and produces a 16-dimensional vector as output, and its connectivity pattern is compatible with $\mathbf{C}$ by construction.

Notably, the kernel $\mathbf{w}$ defines both the matrices $\mathbf{C}$ and $\mathbf{C}^T$ used for the forward and backward passes.

2.2.9 Transposed convolution

Let's now consider what would be required to go the other way around, i.e., map from a 4-dimensional space to a 16-dimensional space, while keeping the connectivity pattern of the convolution depicted in Figure 2.13. This operation is known as a *transposed convolution*.

Transposed convolutions – also called *fractionally strided convolutions* – work by swapping the forward and backward passes of a convolution. One way to put it is to note that the kernel defines a convolution, but whether it's a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.

For instance, although the kernel $\mathbf{w}$ defines a convolution whose forward and backward passes are computed by multiplying with $\mathbf{C}$ and $\mathbf{C}^T$ respectively, it *also* defines a transposed convolution whose forward and backward passes are computed by multiplying with $\mathbf{C}^T$ and $(\mathbf{C}^T)^T = \mathbf{C}$ respectively.[3]

Finally note that it is always possible to emulate a transposed convolution with a direct convolution. The disadvantage is that it usually involves adding many columns and rows of zeros to the input, resulting in a much less efficient implementation.

Building on what has been introduced so far, this chapter will proceed somewhat backwards with respect to the convolution arithmetic chapter, deriving the properties of each transposed convolution by referring to the direct convolution with which it shares the kernel, and defining the equivalent direct convolution.

2.2.10 No zero padding, unit strides, transposed

The simplest way to think about a transposed convolution is by computing the output shape of the direct convolution for a given input shape first, and then inverting the input and output shapes for the transposed convolution.

Let's consider the convolution of a 3×3 kernel on a 4×4 input with unitary stride and no padding (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$). As depicted in Figure 2.13, this produces

[3] The transposed convolution operation can be thought of as the gradient of *some* convolution with respect to its input, which is usually how transposed convolutions are implemented in practice.

a 2×2 output. The transpose of this convolution will then have an output of shape 4×4 when applied on a 2×2 input.

Another way to obtain the result of a transposed convolution is to apply an equivalent – but much less efficient – direct convolution. The example described so far could be tackled by convolving a 3×3 kernel over a 2×2 input padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $k' = k$, $s' = 1$ and $p' = 2$), as shown in Figure 2.21. Notably, the kernel's and stride's sizes remain the same, but the input of the transposed convolution is now zero padded.[4]

One way to understand the logic behind zero padding is to consider the connectivity pattern of the transposed convolution and use it to guide the design of the equivalent convolution. For example, the top left pixel of the input of the direct convolution only contribute to the top left pixel of the output, the top right pixel is only connected to the top right output pixel, and so on.

To maintain the same connectivity pattern in the equivalent convolution it is necessary to zero pad the input in such a way that the first (top-left) application of the kernel only touches the top-left pixel, i.e., the padding has to be equal to the size of the kernel minus one.

Proceeding in the same fashion it is possible to determine similar observations for the other elements of the image, giving rise to the following relationship:

Relationship 8. *A convolution described by $s = 1$, $p = 0$ and k has an associated transposed convolution described by $k' = k$, $s' = s$ and $p' = k - 1$ and its output size is*

$$o' = i' + (k - 1).$$

Interestingly, this corresponds to a fully padded convolution with unit strides.

2.2.11 Zero padding, unit strides, transposed

Knowing that the transpose of a non-padded convolution is equivalent to convolving a zero padded input, it would be reasonable to suppose that the transpose of a zero padded convolution is equivalent to convolving an input padded with *less* zeros.

It is indeed the case, as shown in Figure 2.22 for $i = 5$, $k = 4$ and $p = 2$.

Formally, the following relationship applies for zero padded convolutions:

Relationship 9. *A convolution described by $s = 1$, k and p has an associated transposed convolution described by $k' = k$, $s' = s$ and $p' = k - p - 1$ and its output size is*

$$o' = i' + (k - 1) - 2p.$$

[4] Note that although equivalent to applying the transposed matrix, this visualization adds a lot of zero multiplications in the form of zero padding. This is done here for illustration purposes, but it is inefficient, and software implementations will normally not perform the useless zero multiplications.

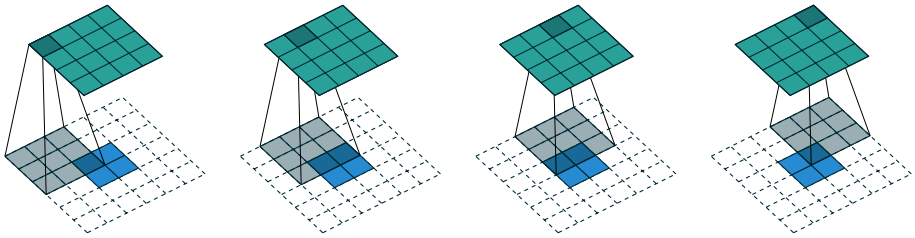


Figure 2.21: *The transpose of convolving a* 3×3 *kernel over a* 4×4 *input using unit strides (i.e.,* $i = 4$*,* $k = 3$*,* $s = 1$ *and* $p = 0$*). It is equivalent to convolving a* 3×3 *kernel over a* 2×2 *input padded with a* 2×2 *border of zeros using unit strides (i.e.,* $i' = 2$*,* $k' = k$*,* $s' = 1$ *and* $p' = 2$*).*

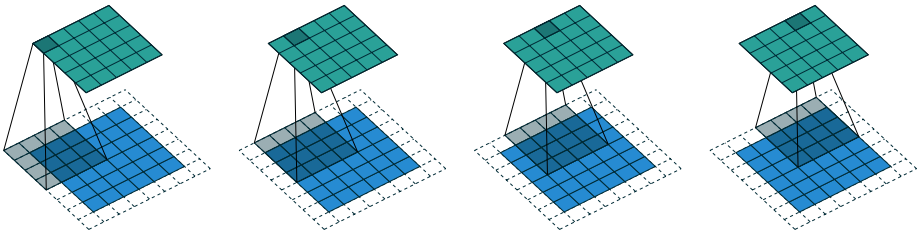


Figure 2.22: *The transpose of convolving a* 4×4 *kernel over a* 5×5 *input padded with a* 2×2 *border of zeros using unit strides (i.e.,* $i = 5$*,* $k = 4$*,* $s = 1$ *and* $p = 2$*). It is equivalent to convolving a* 4×4 *kernel over a* 6×6 *input padded with a* 1×1 *border of zeros using unit strides (i.e.,* $i' = 6$*,* $k' = k$*,* $s' = 1$ *and* $p' = 1$*).*

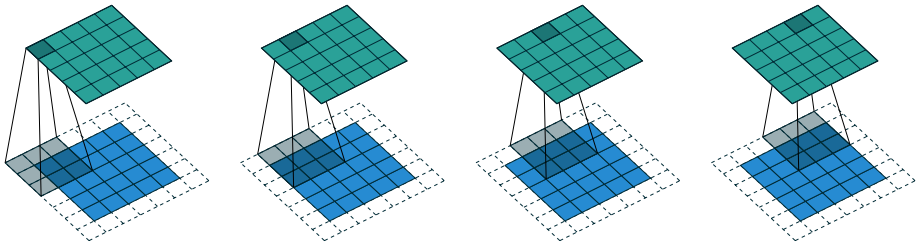


Figure 2.23: *The transpose of convolving a* 3×3 *kernel over a* 5×5 *input using half padding and unit strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 1$ *and* $p = 1$*). It is equivalent to convolving a* 3×3 *kernel over a* 5×5 *input using half padding and unit strides (i.e.,* $i' = 5$*,* $k' = k$*,* $s' = 1$ *and* $p' = 1$*).*

Half (same) padding, transposed By applying the same inductive reasoning as before, it is reasonable to expect that the equivalent convolution of the transpose of a half padded convolution is itself a half padded convolution, given that the output size of a half padded convolution is the same as its input size. Thus the following relation applies:

Relationship 10. *A convolution described by* $k = 2n + 1, \quad n \in \mathbb{N}$, $s = 1$ *and* $p = \lfloor k/2 \rfloor = n$ *has an associated transposed convolution described by* $k' = k$, $s' = s$ *and* $p' = p$ *and its output size is*

$$\begin{aligned} o' &= i' + (k - 1) - 2p \\ &= i' + 2n - 2n \\ &= i'. \end{aligned}$$

Figure 2.23 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 1$.

Full padding, transposed Knowing that the equivalent convolution of the transpose of a non-padded convolution involves full padding, it is unsurprising that the equivalent of the transpose of a fully padded convolution is a non-padded convolution:

Relationship 11. *A convolution described by* $s = 1$, k *and* $p = k - 1$ *has an associated transposed convolution described by* $k' = k$, $s' = s$ *and* $p' = 0$ *and its output size is*

$$\begin{aligned} o' &= i' + (k - 1) - 2p \\ &= i' - (k - 1) \end{aligned}$$

Figure 2.24 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 2$.

2.2.12 No zero padding, non-unit strides, transposed

Using the same kind of inductive logic as for zero padded convolutions, one might expect that the transpose of a convolution with $s > 1$ involves an equivalent convolution with $s < 1$. As will be explained, this is a valid intuition, which is why transposed convolutions are sometimes called *fractionally strided convolutions*.

Figure 2.25 provides an example for $i = 5$, $k = 3$ and $s = 2$ which helps understand what fractional strides involve: zeros are inserted *between* input units, which makes the kernel move around at a slower pace than with unit strides.[5]

For the moment, it will be assumed that the convolution is non-padded ($p = 0$) and that its input size i is such that $i - k$ is a multiple of s. In that case, the following relationship holds:

[5] Doing so is inefficient and real-world implementations avoid useless multiplications by zero, but conceptually it is how the transpose of a strided convolution can be thought of.

Relationship 12. *A convolution described by $p = 0$, k and s and whose input size is such that $i - k$ is a multiple of s, has an associated transposed convolution described by $\tilde{i}'$, $k' = k$, $s' = 1$ and $p' = k - 1$, where $\tilde{i}'$ is the size of the stretched input obtained by adding $s - 1$ zeros between each input unit, and its output size is*

$$o' = s(i' - 1) + k.$$

2.2.13 Zero padding, non-unit strides, transposed

When the convolution's input size i is such that $i + 2p - k$ is a multiple of s, the analysis can extended to the zero padded case by combining Relationship 9 and Relationship 12:

Relationship 13. *A convolution described by k, s and p and whose input size i is such that $i + 2p - k$ is a multiple of s has an associated transposed convolution described by $\tilde{i}'$, $k' = k$, $s' = 1$ and $p' = k - p - 1$, where $\tilde{i}'$ is the size of the stretched input obtained by adding $s - 1$ zeros between each input unit, and its output size is*

$$o' = s(i' - 1) + k - 2p.$$

Figure 2.26 provides an example for $i = 5$, $k = 3$, $s = 2$ and $p = 1$.

The constraint on the size of the input i can be relaxed by introducing another parameter $a \in \{0, \dots, s - 1\}$ that allows to distinguish between the s different cases that all lead to the same i':

Relationship 14. *A convolution described by k, s and p has an associated transposed convolution described by a, $\tilde{i}'$, $k' = k$, $s' = 1$ and $p' = k - p - 1$, where $\tilde{i}'$ is the size of the stretched input obtained by adding $s - 1$ zeros between each input unit, and $a = (i + 2p - k) \mod s$ represents the number of zeros added to the top and right edges of the input, and its output size is*

$$o' = s(i' - 1) + a + k - 2p.$$

Figure 2.27 provides an example for $i = 6$, $k = 3$, $s = 2$ and $p = 1$.

2.3 Recurrent networks

It is well known that the brain is organized in functional areas and sub-areas that process the incoming signal in an incremental fashion. These areas of the brain are typically connected both in a *feedforward*, i.e. to neurons belonging to deeper (i.e., further away from the sensory input) layers, and in a *feedback* fashion (i.e. to previous layers in the hierarchy). This is believed to help processing temporal data and allow for an iterative refinement of the computation.

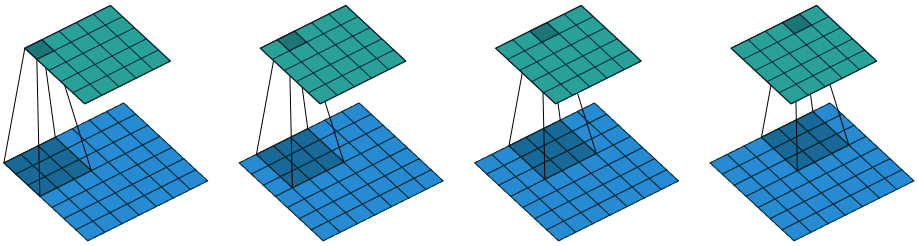


Figure 2.24: *The transpose of convolving a 3×3 kernel over a 5×5 input using full padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 2$). It is equivalent to convolving a 3×3 kernel over a 7×7 input using unit strides (i.e., $i' = 7$, $k' = k$, $s' = 1$ and $p' = 0$).*

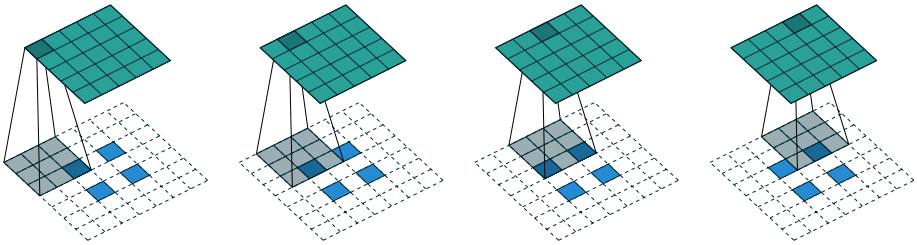


Figure 2.25: *The transpose of convolving a 3×3 kernel over a 5×5 input using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $\tilde{i}' = 3$, $k' = k$, $s' = 1$ and $p' = 2$).*

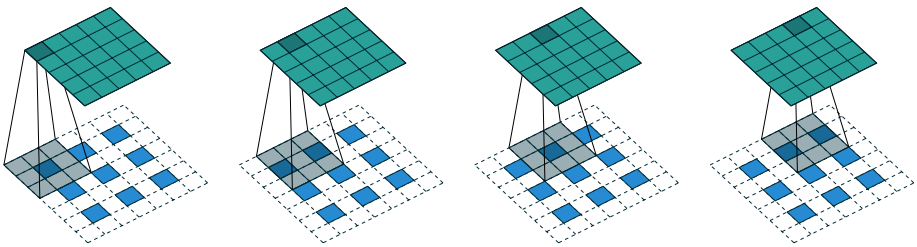


Figure 2.26: *The transpose of convolving a 3×3 kernel over a 5×5 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 1$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 1×1 border of zeros using unit strides (i.e., $i' = 3$, $\tilde{i}' = 5$, $k' = k$, $s' = 1$ and $p' = 1$).*

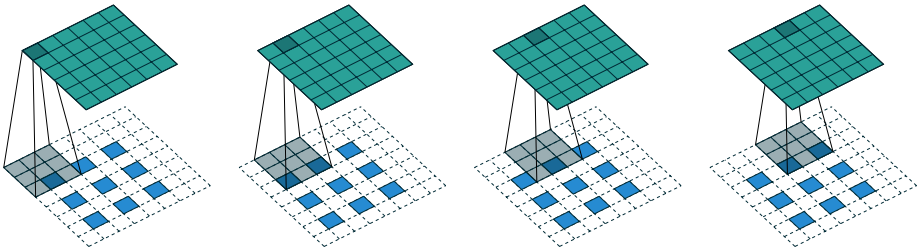


Figure 2.27: *The transpose of convolving a* 3×3 *kernel over a* 6×6 *input padded with a* 1×1 *border of zeros using* 2×2 *strides (i.e.,* $i = 6$*,* $k = 3$*,* $s = 2$ *and* $p = 1$*). It is equivalent to convolving a* 3×3 *kernel over a* 2×2 *input (with* 1 *zero inserted between inputs) padded with a* 1×1 *border of zeros (with an additional border of size* 1 *added to the top and right edges) using unit strides (i.e.,* $i' = 3$*,* $\tilde{i}' = 5$*,* $a = 1$*,* $k' = k$*,* $s' = 1$ *and* $p' = 1$*).*

Similarly, ANNs are not constrained to process the input data in a feedforward way. Recurrent Neural Networks (RNNs) implement feedback loops (see Figure 2.28 [6]) that propagate some information from one step to the next. It is customary to refer to these steps as time-steps, as RNNs are often considered in the context of a discretized time evolving domain, but nothing prevents from using RNNs with any kind of sequential data.

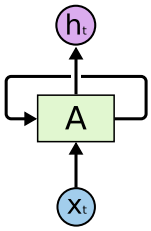


Figure 2.28: *A Recurrent Neural Network (RNN)*

Suppose to be modeling some sequential data $x_0, x_1, \ldots, x_t, x_{t+1}, \ldots$ such as e.g., an English sentence. At time t it is possible to model the probability distribution over a dictionary of English words, conditioned on the words seen up that moment – namely $x_0, x_1, \ldots, x_{t-1}$. This can be used, e.g., in a smart keyboard application to suggest the next most probable words to the user, or in an automatic help desk to generate a meaningful answer to a question.

It might not be immediately obvious what it means in practice to put a loop in an ANN and how to backpropagate through it. To better comprehend how RNNs work it is useful to consider its behavior explicitly by *unrolling* the RNN, as shown in Figure 2.29
An RNN applies the same model to each time step of the sequence or, equivalently, applies different models at each time step, which share their weights. This is similar to what CNNs do over space with convolutions, but is rather done over time with feedback connections.

[6] This and the following RNN figures, are taken or modified from the awesome introduction to RNNs and LSTMs by Chris Olah (Olah, 2015)

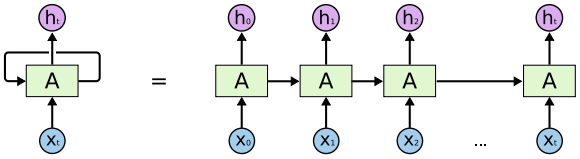


Figure 2.29: *A Recurrent Neural Network unrolled for tsteps*

The activation of an RNN at time t depends on the input at time t as well as on the information coming from the previous step $t-1$. RNNs have a very simple internal structure, that usually amounts to applying some affine transformation to the input and to the previous output, and computing some non-linearity (typically a tanh) of their sum.

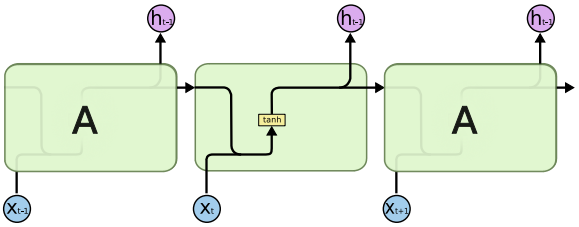


Figure 2.30: *The internal structure of an RNN*

To train it it suffices to unroll the computation graph and use the backpropagation algorithm (see Section 2.1.4) to proceed from the most recent time step, backward in time. This algorithm is usually referred to as *Backpropagation through time (BPTT)*.

The problem of BPTT is that it requires to apply the chain rule all the way from the current time step to $t = 0$ to propagate the gradients through time. This results in a long chain of products that can easily go to infinity or become zero if the elements of the multiplication are greater or smaller than 1 respectively. These two issues are known in the literature as *exploding gradient* and *vanishing gradient* and have been studied extensively in the past (see e.g., Hochreiter, 1991; Bengio *et al.*, 1994). The first one can be partially addressed by *clipping the gradient* when it becomes too large, but the second is not easy to overcome and can make training these kind of models very hard if not impossible.

2.3.1 Long Short Term Memory

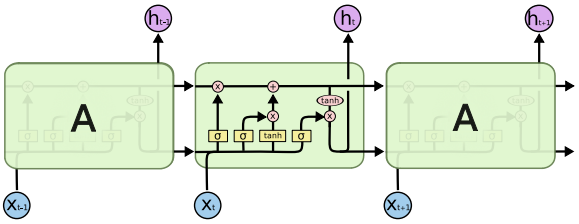
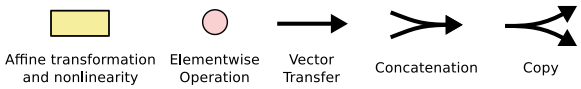


Figure 2.31: *A Long Short Term Memory (LSTM)*



Long Short Term Memory (LSTM) networks (see Figure 2.31) have been proposed to solve (or at least alleviate) the problems of RNNs with modeling long term dependencies. LSTMs have been designed to have an internal memory, or *state*, that can be updated and consulted at each time step. As opposed to vanilla RNNs, this allows LSTMs to separate their output from the information they want to carry over to future steps.

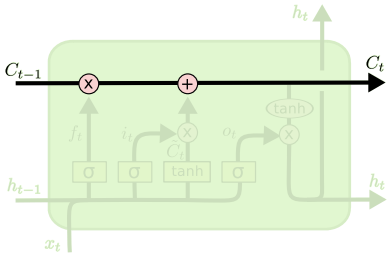
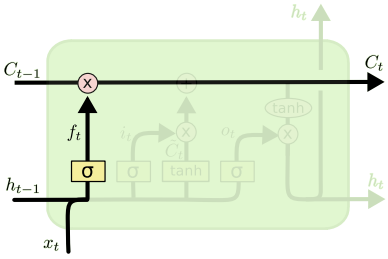


Figure 2.32: *The internal state of LSTMs*

Figure 2.32 highlights the internal memory path. It can be seen how the internal memory of the previous time step C_{t-1} is carried over to the current time step, where it is updated through a multiplicative and an additive interaction and concurs to determine the current state of the memory C_t. This is then, once again, propagated to the next time step.

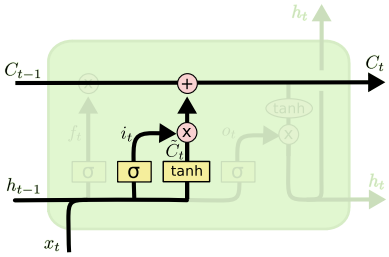


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \qquad (2.18)$$

Figure 2.33: *The LSTM forget gate*

LSTMs interact with memory through *gates*, computational nodes that determine the behavior of the model. The *forget gate* determines how much of the previous step's memory to forget or, equivalently, how much of the previous state to retain (Figure 2.33). This is modeled through a sigmoid layer (depicted as σ) that takes the current input x_t and the output of the previous step h_{t-1} and produces an activation vector between 0 and 1. This activation is multiplied by the previous state C_{t-1} and results in an intermediate memory state where some of the activations can be weaker than C_{t-1} and some others are potentially zeroed out (Equation 2.18).

The forget gate (Figure 2.34) allows the LSTM to discard information that is not relevant anymore. Symmetrically, LSTMs have a mechanism to add new information to the memory. This behavior is controlled by an *input gate* that modulates the amount of the



$$\begin{aligned} i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\ \tilde{C}_t &= tanh\left(W_C \cdot [h_{t-1}, x_t] + b_f\right) \end{aligned} \tag{2.19}$$

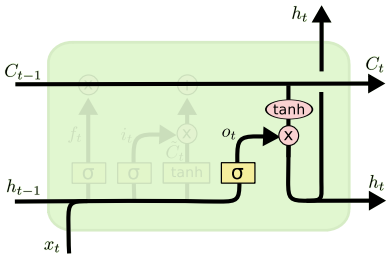
Figure 2.34: *The LSTM input gate*

current input that is going to be stored in the memory. This operation is split over two computation paths: similarly to the forget gate, the input gate takes the current input x_t and the output of the previous step h_{t-1} and exploits a sigmoid layer to produce an activation vector between 0 and 1. Simultaneously, a tanh layer generates a state update $\tilde{C}_t$ between -1 and 1. This is governed by the equations reported in Equation 2.19.

The input gate modulates how much of this state update will be applied to the old state to generate the current state. The forget gate f_t and the input gate i_t, together with the previous state C_{t-1} fully determine the state at time t through Equation 2.20.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{2.20}$$

The last gate of LSTMs is the *output gate* o_t that, as the name reveals, manipulates the output of the LSTM at time t (Figure 2.35). The usual sigmoid layer determines the state of the output gate. The memory resulting from the transformations due to the forget and input gates goes through a tanh nonlinearity and is multiplied by the output gate to finally produce the output (Equation 2.21).

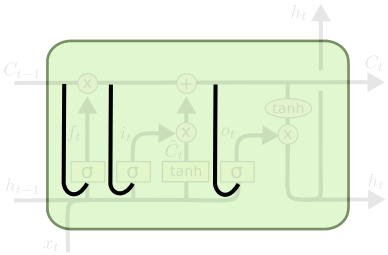


$$\begin{aligned} o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \\ h_t &= o_t \odot tanh\left(C_t\right) \end{aligned} \tag{2.21}$$

Figure 2.35: *The LSTM output gate*

LSTMs with peepholes

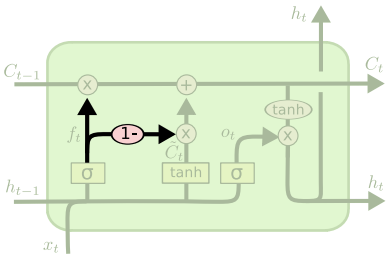
Gers and Schmidhuber (2000) suggests one variant of LSTMs where the gates also have information about the state of the LSTM. Note that, as illustrated in Figure 2.36, the output gate peeps into C_t, the state after the input and forget gate updates (see the output gate equation in Equation 2.22).



$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t, C_{t-1}] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t, C_{t-1}] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t, C_t] + b_o\right)
\end{aligned} \tag{2.22}
$$

Figure 2.36: *LSTM with peepholes*

LSTMs with coupled forget and input gates

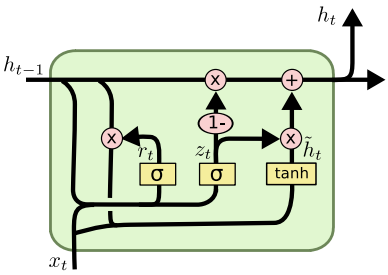


$$C_t = f_t \odot C_{t-1} + (1 - f_t) \odot \tilde{C}_t \tag{2.23}$$

Figure 2.37: *LSTM with coupled forget and input gate*

Another variant of LSTMs replaces the input gate with $1 - f_t$ (see Figure 2.37), so that the forget gate governs both behaviors. This boils down to forgetting only when a new input is going to be written. The modified state update equation is Equation 2.23.

2.3.2 Gated Recurrent Unit (GRU)



$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \\
\tilde{h}_t &= tanh\left(W_o \cdot [r_t \odot h_{t-1}, x_t]\right) \\
h_t &= (1 - z_t) \odot h_{t-1} + (z_t) \odot \tilde{h}_t
\end{aligned} \tag{2.24}
$$

Figure 2.38: *Gated Recurrent Units (GRUs)*

In 2014 Cho *et al.* (2014b) proposed a new kind of recurrent network called Gated Recurrent Unit (GRU) with less gates than LSTMs and a different internal structure. Along the lines of Section 2.3.1, in GRUs the forget and input gates are coupled into an *up-*

date gate z_t. The memory and output are also merged into a single state and the internal structure is modified to cope with these changes (see Figure 2.38

The advantage of GRUs w.r.t. LSTMs is that having less gates they are less memory and computationally intense, which is often a critical aspect for ANNs. GRUs have been shown to perform as well as LSTMs in some settings (see Chung *et al.* (2014)).

CHAPTER 3

Object classification with Recurrent Neural Networks

Classification is a broad task that consists in predicting to which category an input belongs to, out of some k given categories. The system is usually required to compute a function $f : \mathbb{R} \rightarrow 0, \ldots, k-1$ that assigns each input to a class. This can be thought as a discrimination task, where the algorithm is expected to learn similarities and differences between categories in order to characterize each class.

Some examples of classification are spam detection (classify a message as being spam or not), credit card fraud detection (identify if a transaction is legit or not), optical character recognition (OCR) (convert hand-written text to an electronic document, by classifying each character), speech understanding (given an utterance from a user, detect the sentence that was pronounced), medical diagnosis (given the symptoms predict the illness and suggest a cure), stock trading (determine if a stock should be bought, help or sold, from current and historical data), shape detection (classify a hand-drawn drawing from, e.g., a touch screen, as a specific shape) and emotion recognition (given a text, classify it as being positive, neutral or negative)

Object recognition is an instance of this task that takes an image as an input and is expected to return the id of the main class represented in the image. This is usually achieved by computing the probability distribution over the classes – i.e. the confidence of the algorithm for each class to be the main class in the scene – and then picking the one with highest score.

One downside of this method is that small and big mistakes are penalized the same,

i.e., the prediction is considered wrong whether the right class is the second most probable class in the distribution or is considered extremely unlikely by the algorithm. For this reason, some competitions such as e.g., ImageNet (Deng *et al.*, 2009; Russakovsky *et al.*, 2014), propose multiple challenges where the top-3 or top-5 guesses are considered, i.e., where an image is considered correctly predicted if the correct class is in the top 3 or 5 guesses of the network.

Traditionally the computer vision community used to address this task with heavily hand-engineered systems that typically resorted to finding easily detectable elements in the image – such as e.g. edges or corners – and computing a descriptor of the surrounding patch. Many detectors Dufournaud *et al.* (2000); Harris and Stephens (1988); Mikolajczyk and Schmid (2001); Lowe (2004); Mikolajczyk and Schmid (2005) and descriptors Lowe (1999); Mikolajczyk and Schmid (2005); Belongie *et al.* (2002) have been proposed to this end, until in 2012 a deep convolutional neural network shifted the balance toward learned ANNs indefinitely (Krizhevsky *et al.*, 2012a). Since then CNN-based models dominated the object recognition scene.

This chapter presents ReNet, one of the main contributions of this thesis. The ReNet model is a deep neural network architecture for object recognition, based on recurrent neural networks. The main idea behind this project is to propose an alternative to the typical CNN-based approach to object classification problems. The ReNet model replaces in fact the ubiquitous convolution+pooling layers of deep CNNs with four recurrent neural networks that sweep horizontally and vertically in both directions across the image.

The following sections motivate the model in the context of the state of the art at the time it was conceived, describe the ReNet model in detail and present the results of its evaluation on three widely-used object recognition benchmarks, namely MNIST (LeCun *et al.*, 1999), CIFAR-10 (Krizhevsky and Hinton, 2009) and SVHN (Netzer *et al.*, 2011). The experiments reveal that the ReNet model performs comparably to convolutional neural networks on all these datasets, suggesting the potential of RNNs as a competitive alternative to the conventional deep convolutional neural networks for image related tasks.

3.1 Motivation

Convolutional neural networks (CNN, Fukushima, 1980; LeCun *et al.*, 1989) have become the method of choice for object recognition after the impressive improvement on the state of the art of Krizhevsky *et al.* (2012a). CNNs have proved to be successful at a variety of benchmark problems including, but not limited to, handwritten digit recognition (see, e.g., Ciresan *et al.*, 2012b), natural image classification (see, e.g., Lin *et al.*, 2014; Simonyan and Zisserman, 2015; Szegedy *et al.*, 2014), house number recognition (see, e.g., Goodfellow *et al.*, 2014), traffic sign recognition (see, e.g., Ciresan *et al.*, 2012a), as well as for speech recognition (see, e.g., Abdel-Hamid *et al.*, 2012; Sainath *et al.*, 2013; Tóth, 2014). Furthermore, image representations from CNNs trained to recognize objects on a large set of more than one million images (Simonyan and Zisserman, 2015; Szegedy *et al.*, 2014) have been found to be extremely helpful in performing other computer vision tasks such as image caption generation (see, e.g., Vinyals *et al.*, 2014; Xu *et al.*, 2015), video description generation (see, e.g., Yao *et al.*, 2015) and object localization/detection (see, e.g., Sermanet *et al.*, 2014).

While the CNN has been especially successful in computer vision, recurrent neural

networks (RNN) have become the method of choice for modeling sequential data, such as text and sound. Natural language processing (NLP) applications include language modeling (see, e.g., Mikolov, 2012), and machine translation (Sutskever *et al.*, 2014; Cho *et al.*, 2014a; Bahdanau *et al.*, 2015). Other popular areas of application include offline handwriting recognition/generation (Graves and Schmidhuber, 2009; Graves *et al.*, 2008; Graves, 2013) and speech recognition (Chorowski *et al.*, 2014; Graves and Jaitly, 2014). RNNs have also been used together with CNNs in speech recognition (Sainath *et al.*, 2015). The recent revival of RNNs has largely been due to advances in learning algorithms (Pascanu *et al.*, 2013; Martens and Sutskever, 2011) and model architectures (Pascanu *et al.*, 2014; Hochreiter and Schmidhuber, 1997; Cho *et al.*, 2014a).

The architecture of ReNet is related and inspired by this earlier work, but relies on purely uni-dimensional RNNs coupled in a novel way, rather than on a multi-dimensional RNN. The basic idea behind the ReNet model is to replace each convolutional layer (with convolution+pooling making up a layer) in the CNN with four RNNs that sweep over lower-layer features in different directions: (1) bottom to top, (2) top to bottom, (3) left to right and (4) right to left. The recurrent layer ensures that each feature activation in its output is an activation at the specific location *with respect to the whole image*, in contrast to the usual convolution+pooling layer which only has a local context window. The lowest layer of the model sweeps over the input image, with subsequent layers operating on extracted representations from the layer below, forming a hierarchical representation of the input.

Graves and Schmidhuber (2009) have demonstrated an RNN-based object recognition system for offline Arabic handwriting recognition. The main difference between ReNet and the model of Graves and Schmidhuber (2009) is that it uses the usual sequence RNN, instead of the multidimensional RNN. The way ReNet has been conceived allows in fact to capture the context of the image without being forced to resort to multidimensional RNNs: the latter two RNNs (or, equivalently, the last bidirectional RNN), work on the hidden states computed by the first two RNNs (or the first bidirectional RNN). This allows to use plain RNNs instead of the more complex multidimensional ones, while making each output activation of the layer be computed with respect to the whole input image.

One important consequence of the proposed approach compared to the multidimensional RNN is that the number of RNNs at each layer scales linearly with respect to the number of dimensions d of the input image ($2d$). A multidimensional RNN, on the other hand, requires an exponential number of RNNs at each layer (2^d). Furthermore, the proposed variant is more easily parallelizable, as each RNN is dependent only along a horizontal or vertical sequence of patches. This architectural distinction results in the ReNet model being much more amenable to distributed computing than that of Graves and Schmidhuber (2009).

3.2 Model Description

Let us denote by $X = \{x_{i,j}\}$ the input image or the feature map from the layer below, where $X \in \mathbb{R}^{w \times h \times c}$ with w, h and c the width, height and number of channels, or the feature dimensionality, respectively. Given a receptive field (or patch) size of $w_p \times h_p$, we split the input image X into a set of $I \times J$ (non-overlapping) patches $P = \{p_{i,j}\}$, where $I = \frac{w}{w_p}$, $J = \frac{h}{h_p}$ and $p_{i,j} \in \mathbb{R}^{w_p \times h_p \times c}$ is the (i, j)-th patch of the input image. The first

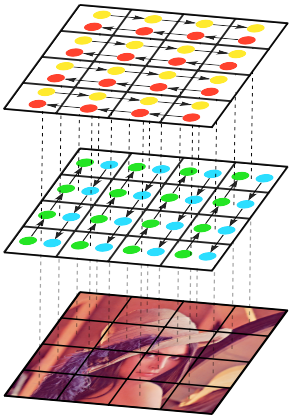


Figure 3.1: *A ReNet layer is composed by two sublayers, a vertical one and an horizontal one*

index i is the horizontal index and the other index j is the vertical index.

First, the image is swept vertically with two RNNs, with one RNN working in a bottom-up direction and the other working in a top-down direction. Each RNN takes as an input one (flattened) patch at a time and updates its hidden state, working *along each column* j of the split input image X.

$$v^F_{i,j} = f_{\text{VFWD}}(z^F_{i,j-1}, p_{i,j}), \text{ for } j = 1, \cdots, J \tag{3.1}$$

$$v^R_{i,j} = f_{\text{VREV}}(z^R_{i,j+1}, p_{i,j}), \text{ for } j = J, \cdots, 1 \tag{3.2}$$

Note that f_{VFWD} and f_{VREV} return the activation of the recurrent hidden state, and may be implemented either as a simple $\tanh$ layer, as a gated recurrent layer (Cho *et al.*, 2014a) or as a long short-term memory layer (Hochreiter and Schmidhuber, 1997).

After this vertical, bidirectional sweep, the intermediate hidden states $v^F_{i,j}$ and $v^R_{i,j}$ are concatenated at each location (i, j) to get a composite feature map $V = \{v_{i,j}\}^{j=1,\ldots,J}_{i=1,\ldots,I}$, where $v_{i,j} \in \mathbb{R}^{2d}$ and d is the number of recurrent units. Each $v_{i,j}$ is now the activation of a feature detector at the location (i, j) with respect to all the patches in the j-th column of the original input ($p_{i,j}$ for all i).

Next, the obtained composite feature map V is swept horizontally with two different RNNs (f_{HFWD} and f_{HREV}). In a similar manner as for the vertical sweep, these RNNs work along each row of V and produce an output feature map $H = \{h_{i,j}\}$, where $h_{i,j} \in \mathbb{R}^{2d}$. In this representation, each vector $h_{i,j}$ represents the features of the original image patch $p_{i,j}$ *in the context of the whole image*.

This is a critical feature of this model, in fact just one layer (to be precise, two sublayers) allows to capture the full context of the image, irrespective of the image size. This is possible thanks to the (potential) ability of RNNs to store in their memory any information that is relevant to retain the context of the part of the image that has been processed. The first two RNNs capture the horizontal dependencies in both directions. By reading their composite feature map, the second pair of RNNs has access in each position to a "summary" of the corresponding row. This is processed vertically, to capture the missing dependencies between rows. This intra- and inter-row processing results in a final

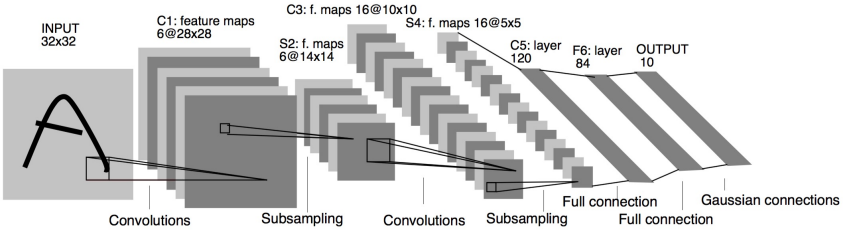


Figure 3.2: *The LeNet network*

composite feature map where each position is specific to a pixel (or patch) of the image but has information on the full image. Conversely, to span over the whole image with CNN-based architectures would require many more layers, whose number depend on the size of the input image.

Even if each ReNet layer captures the full input context, it is clearly still possible to stack multiple ReNet layers on top of each other into a deep network. Let us denote by ϕ the function from the input image (or feature map) X of one ReNet layer to the output feature map H (see Figure 3.1 for a graphical illustration). It is possible to compute a composition of functions $\Phi = \phi_1(\phi_2(\phi_3(\dots)))$ by stacking multiple ReNet layers, to capture increasingly complex features of the input image. After any number of recurrent layers are applied to an input image, the activation at the last recurrent layer may be flattened and fed into a differentiable classifier to solve an object recognition task. The experiments on this model, presented in Section 3.3, used several fully-connected layers followed by a softmax classifier, as shown in **??**.

The deep ReNet is a smooth, continuous function, and the parameters (those from the RNNs as well as from the fully-connected layers) can be estimated by the stochastic gradient descent algorithm with the gradient computed by backpropagation algorithm (see, e.g., Rumelhart *et al.*, 1986b) to maximize the log-likelihood.

3.2.1 Differences between LeNet and ReNet

This section will use LeNet (see Figure 3.2) to refer to the canonical convolutional neural network as shown by LeCun *et al.* (1989). There are many similarities and differences between the ReNet model and a convolutional neural network. The main key points of comparison will be highlighted in what follows.

At each layer, both networks apply the same set of filters to patches of the input image or of the feature map from the layer below. ReNet, however, propagates information through lateral connections that span across the whole image, while LeNet exploits local information only. The lateral connections should help extract a more compact feature representation of the input image at each layer, which can be accomplished by the lateral connections removing/resolving redundant features at different locations of the image. This should allow ReNet resolve small displacements of features across multiple consecutive patches. Also, the lack of this type of lateral connection in LeNet may lead to many more levels of convolution+pooling layers in order to detect redundant features from different parts of the image.

LeNet max-pools the activations of each filter over a small region to achieve local

translation invariance. In contrast, the proposed ReNet does not use any pooling due to the existence of learned lateral connections. The lateral connection in ReNet can emulate the local competition among features induced by the max-pooling in LeNet. This does not mean that it is not possible to use max-pooling in ReNet. The use of max-pooling in the ReNet could be helpful in reducing the dimensionality of the feature map, resulting in lower computational cost.

Max-pooling as used in LeNet may prove problematic when building a convolutional autoencoder whose decoder is an inverse[1] of LeNet, as the max operator is not invertible. The proposed ReNet is end-to-end smooth and differentiable, making it more suited to be used as a decoder in the autoencoder or any of its probabilistic variants (see e.g., Kingma and Welling, 2014).

In some sense, each layer of the ReNet can be considered as a variant of a usual convolution+pooling layer, where pooling is replaced with lateral connections, and convolution is done without any overlap. Similarly, Springenberg *et al.* (2014) recently proposed a variant of a usual LeNet which does not use any pooling. They used convolution with a larger stride to compensate for the lack of dimensionality reduction by pooling at each layer. However, this approach still differs from the proposed ReNet in the sense that each feature activation at a layer is only with respect to a subset of the input image rather than the whole input image.

The main disadvantage of ReNet is that it is not easily parallelizable, due to the sequential nature of the recurrent neural network (RNN). LeNet, on the other hand, is highly parallelizable due to the independence of computing activations at each layer. The introduction of sequential, lateral connections, however, may result in more efficient parametrization, requiring a smaller number of parameters with overall fewer computations, although this needs to be further explored. Note that this limitation on parallelization applies only to model parallelism, and any technique for data parallelism may be used for both the proposed ReNet and the LeNet.

3.3 Experiments

3.3.1 Datasets

The ReNet model has been evaluated on three widely-used benchmark datasets; MNIST, CIFAR-10 and the Street View Housing Numbers (SVHN). This section describes each dataset in detail.

MNIST

The MNIST dataset (LeCun *et al.*, 1999) consists of 70,000 handwritten digits from 0 to 9, centered on a 28×28 square canvas (see Figure 3.3 for some samples). Each pixel represents the grayscale in the range of $[0, 255]$.[2] It is customary to split this dataset into 50,000 training samples, 10,000 validation samples and 10,000 test samples. For a fair comparison, the results reported in Section 3.3.4 were obtained following the standard split.

[1] All the forward arrows from the input to the output in the original LeNet are reversed.

[2] Each pixel has been scaled to $[0, 1]$ by dividing it with 255.

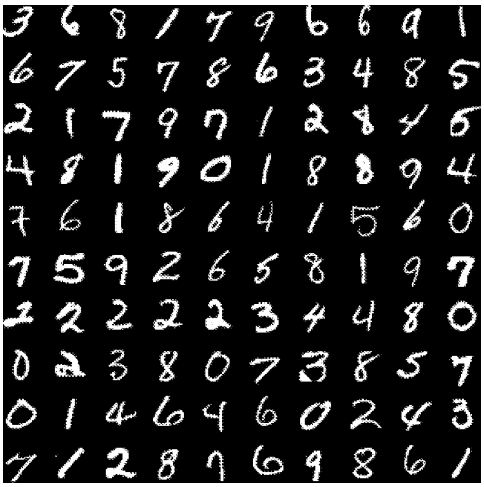


Figure 3.3: *Some digits from the MNIST dataset*

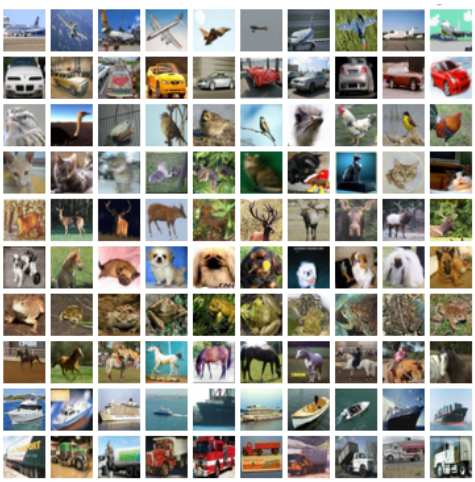


Figure 3.4: *Some samples of the 10 classes of the CIFAR-10 dataset*

CIFAR-10

The CIFAR-10 dataset (Krizhevsky and Hinton, 2009) is a curated subset of the 80 million tiny images dataset (see Figure 3.4 for some samples), originally released by Torralba *et al.* (2008). CIFAR-10 contains 60,000 images each of which belongs to one of ten categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Each image is 32 pixels wide and 32 pixels high with 3 color channels (red, green and blue.) Following the standard procedure, in the reported experiments the dataset was split into 40,000 training, 10,000 validation and 10,000 test samples. Furthermore, zero-phase component analysis (ZCA) was applied and the each pixel was normalized to have zero-mean and unit-variance across the training samples, as suggested by Krizhevsky and Hinton (2009).

Figure 3.5: *Some samples of housing numbers from the SVHN dataset*

Street View House Numbers

The Street View House Numbers (SVHN) dataset (Netzer *et al.*, 2011) consists of cropped images representing house numbers captured by Google StreetView vehicles as a part of the Google Maps mapping process. These images consist of digits 0 through 9 with values in the range of [0, 255] in each of 3 red-green-blue color channels (see Figure 3.5 for some samples). Each image is 32 pixels wide and 32 pixels high giving a sample dimensionality (32, 32, 3). The number of samples used for the training, validation, and test sets is 543,949, 60,439, and 26,032 respectively. Each pixel was normalized to have zero-mean and unit-variance across the training samples.

3.3.2 Data Augmentation

It has been long known that augmenting training data often leads to better generalization (see, e.g., Krizhevsky *et al.*, 2012a). The results reported in Section 3.3.4 were all obtained employing two primary data augmentations: *flipping* and *shifting*.

For the flipping data augmentation, the model was presented with samples that were either flipped horizontally with 25% chance, or vertically with 25% chance, or left unchanged. This technique allows the model to observe "mirror images" of the original images of the dataset during training, increasing the amount of training data. On SVHN and MNIST only horizontal flipping was used to prevent the case were an image labelled 6 is flipped in both direction, becoming a 9.

In the case of shifting two policies were adopted. The images were either shifted by 2 pixels to the left (25% chance), 2 pixels to the right (25% chance) or left as they were. After this first processing, the images were further shifted either by 2 pixels to the top (25% chance), 2 pixels to the bottom (25% chance) or left as they were. This two-step procedure makes the model more robust to slight shifting of an object in the image. The shifting was done without pre-padding the borders of the image, but rather preserving the

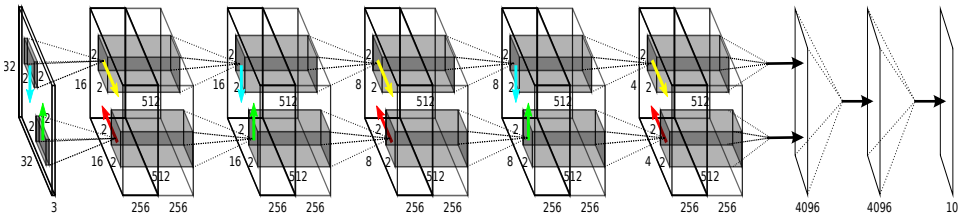


Figure 3.6: *The ReNet network used for SVHN classification*

	MNIST	CIFAR-10	SVHN
N_{RE}	2	3	3
$w_p \times h_p$	$[2 \times 2]$–$[2 \times 2]$	$[2 \times 2]$–$[2 \times 2]$–$[2 \times 2]$	$[2 \times 2]$–$[2 \times 2]$–$[2 \times 2]$
d_{RE}	256–256	320–320–320	256–256–256
N_{FC}	2	1	2
d_{FC}	4096–4096	4096	4096–4096
f_{FC}	$\max(0, x)$	$\max(0, x)$	$\max(0, x)$
Flipping	no	yes	no
Shifting	yes	yes	yes

Table 3.1: *Model architectures used in the experiments. Each row shows respectively the number of ReNet layers, the size of the patches, the number of neurons of each ReNet layer, the number of fully connected layers, the number of neurons of the fully connected layers, their activation function and the data augmentation procedure employed.*

original size by dropping the pixels which are shifted out of the input and shifting in zeros on the opposite side.

The choice of whether to apply these augmentation procedures on each dataset was chosen on a per-case basis in order to maximize validation performance.

3.3.3 Experimental settings

The principal parameters that define the architecture of the ReNet model are the number of ReNet layers (N_{RE}), their corresponding receptive field sizes ($w_p \times h_p$) and feature dimensionality (d_{RE}), the number of fully-connected layers (N_{FC}) and their corresponding numbers (d_{FC}) and types (f_{FC}) of hidden units.

Table 3.1 summarizes the settings of these hyperparameters that performed best on the validation set of the studied datasets. Figure 3.6 is a graphical illustration of the model selected with this metric for the SVHN dataset.

Training

All the networks have been trained using a recently proposed adaptive learning rate algorithm, called Adam (Kingma and Ba, 2014). In order to reduce overfitting dropout (Srivastava *et al.*, 2014) was applied after each layer, including both the ReNet layers (after the horizontal and vertical sweeps) and the fully-connected layers. The input layer was

Test Error	Model
0.28%	(Wan *et al.*, 2013)⋆
0.31%	(Graham, 2014b)⋆
0.35%	(Ciresan *et al.*, 2010)
0.39%	(Mairal *et al.*, 2014)⋆
0.39%	(Lee *et al.*, 2014)⋆
0.4%	(Simard *et al.*, 2003)⋆
0.44%	(Graham, 2014a)⋆
0.45%	(Goodfellow *et al.*, 2013)⋆
0.45%	**ReNet**
0.47%	(Lin *et al.*, 2014)⋆
0.52%	(Azzopardi and Petkov, 2013)

(a) MNIST

Test Error	Model
4.5%	(Graham, 2014a)⋆
6.28%	(Graham, 2014b)⋆
8.8%	(Lin *et al.*, 2014)⋆
9.35%	(Goodfellow *et al.*, 2013)⋆
9.39%	(Springenberg and Riedmiller, 2013)⋆
9.5%	(Snoek *et al.*, 2012)⋆
11%	(Krizhevsky *et al.*, 2012a)⋆
11.10%	(Wan *et al.*, 2013)⋆
12.35%	**ReNet**
15.13%	(Zeiler and Fergus, 2013)⋆
15.6%	(Hinton *et al.*, 2012)⋆

(b) CIFAR-10

Test Error	Model
1.92%	(Lee *et al.*, 2014)⋆
2.23%	(Wan *et al.*, 2013)⋆
2.35%	(Lin *et al.*, 2014)⋆
2.38%	**ReNet**
2.47%	(Goodfellow *et al.*, 2013)⋆
2.8%	(Zeiler and Fergus, 2013)⋆

(c) SVHN

Table 3.2: *Generalization errors obtained by the proposed ReNet along with those reported by previous works on each of the three datasets. For a fair comparison, only results obtained by a single model are listed, i.e., no ensembling of multiple models. In the case of SVHN, only models trained on the Format 2 (cropped digits) dataset are reported. ⋆ denotes a convolutional neural network.*

also corrupted by masking out each pixel with probability 0.2. Finally, each optimization run was early stopped based on validation error.

Note that the results reported in Section 3.3.4 were obtained without retraining on the joint training and validation sets. This is a common technique exploited by many works in the literature to boost the performance of the best model, selected after a full training on the training set early stopping on the validation performance. There is no reason not to think that this would further improve the reported results, but this work was mainly conceived as a proof of concept rather than to stress on the absolute performance. This, as well as e.g., ensembling multiple models, can be seen as one of the many potential areas of exploration for further work in case maximum performance is sought.

3.3.4 Results

Table 3.2, presents the results of ReNet on three datasets, along with previously reported results.

It is clear that ReNet performs comparably to deep convolutional neural networks which are the *de facto* standard for object recognition. This suggests that ReNet is a viable alternative to convolutional neural networks (CNN), even on tasks where CNNs have historically dominated. However, it is important to notice that ReNet does not outperform state-of-the-art convolutional neural networks on any of the three benchmark datasets,

which calls for more research in the future.

3.4 Discussion

The ReNet model successfully proved that RNN-based model can and should be explored also in contexts where traditional CNN-based model have established the state of the art for a long time, such as object recognition. Many aspects of RNN-based models applied to this context remain to be studied in more depth, some of which will be outlined in the rest of this section.

Choice of Recurrent Units

Note that the proposed architecture is independent of the chosen recurrent units. The preliminary experiments on this model showed that gated recurrent units, either the GRU or the LSTM, outperform a usual sigmoidal unit (affine transformation followed by an element-wise sigmoid function.) This indirectly confirms that the model utilizes long-term dependencies across an input image, and the gated recurrent units help capture these dependencies.

Analysis of the Trained ReNet

The ReNet model has been only evaluated in a quantitative fashion. However, the accuracies on the test sets do not reveal what kind of image structures the ReNet has captured in order to perform object recognition. Due to the large differences between ReNet and LeNet discussed in Sec. 3.2.1, it can be expected that the internal behavior of ReNet will differ from that of LeNet significantly. Further investigation along the line of (Zeiler and Fergus, 2014b) is needed, as well as exploring ensembles that combine RNNs and CNNs for bagged prediction.

Computationally Efficient Implementation

As discussed in Sec. 3.2.1, the proposed ReNet is less parallelizable due to the sequential nature of the recurrent neural network (RNN). Although this sequential nature cannot be addressed directly, our construction of ReNet allows the forward and backward RNNs to be run independently from each other, which allows for parallel computation. Furthermore, it is possible to exploit the many parallelization tricks widely used for training convolutional neural networks such as parallelizing fully-connected layers (Krizhevsky, 2014), having separate sets of kernels/features in different processors (Krizhevsky *et al.*, 2012a) and exploiting data parallelism.

CHAPTER 4

ReSeg

Chapter 3 introduced the ReNet model, a Recurrent Neural Network based model for object recognition. The ReNet model scans the image horizontally and vertically with 4 RNNs at each layer and is able to capture the full context of the input with just one layer thanks to a sophisticated interaction of the inner RNNs. This allows activations to be local yet conditioned on global information, an ideal setting for semantic segmentation.

Semantic segmentation is the task of labeling each pixel of an image with the class it belongs to. In, e.g., an urban scene scenario, this would mean to label all the pixels of all the cars in the image as belonging to the "car class", every pixel of all the pedestrians in the image as belonging to the pedestrian class, and so on.

This is a very difficult task for many reasons: classifying pixels requires to acquire both a global understanding of the scene, as well as a very detailed and spatially precise characterization of each object; drawing segmentation masks is very time consuming, which makes difficult and expensive to collect big datasets; labels are often subject to personal judgement, in fact different people tend to have different degrees of accuracy on small details and contours and at the same time, it is hard to define a clear separation path between the object and the background sometimes (e.g., the leafs of a tree)

This chapter introduces the second main contribution of this work. The ReSeg model Visin *et al.* (2016) builds on ReNet (see Chapter 3) to tackle the task of semantic segmentation. The primary motivation behind ReSeg is to exploit the peculiar structure of ReNet in order to capture the underlying semantic of the input image and use it to drive the prediction in each location. Thanks to the strong lateral connections built in the model, ReSeg is able to capture the *global* scene depicted in the image and exploit it to perform fine, high

resolution, semantic segmentation focusing on *local* details.

This work extends the preliminary results of Visin *et al.* (2015) modifying and extending the ReNet model to the more ambitious task of object segmentation. The performance of the proposed model are tested on some of the historically most used datasets in this field, namely the Weizmann Horse dataset Borenstein (2004), the Oxford Flowers 17 dataset Nilsback and Zisserman (2006) and the more recent and challenging Camvid dataset (Brostow *et al.*, 2009, 2008). The first two are tackled in a foreground/background segmentation setting as a proof of concept for the proposed ReSeg architecture. The performance of the model is then tested on the full segmentation task on Camvid, a standard benchmark of urban scenes.

The experiments show that the proposed adaptation of the ReNet for pixel-level object segmentation performs successfully on the object segmentation task achieving state-of-the-art in all three datasets and may have further applications in other structured prediction problems. [1] Furthermore, the ReNet and ReSeg architectures could be easily merged into a joint network to perform both tasks at the same time, sharing most of the computation. This could be interesting in application domains where object classification and segmentation have to be performed simultaneously, such as, e.g., autonomous driving and object retrieval.

In the following of this chapter, Section 4.1 motivates the model in the context of the state of the art at the time it was conceived, Section 4.2 describe the ReSeg model in detail and Section 4.3 presents the results of the experiments.

4.1 Motivation

In recent years, Convolutional Neural Networks (CNN) have become the *de facto* standard in many computer vision tasks, such as image classification and object detection Krizhevsky *et al.* (2012a); Erhan *et al.* (2014). Top performing image classification architectures usually involve *very* deep CNN trained in a supervised fashion on a large datasets Lin *et al.* (2014); Simonyan and Zisserman (2015); Szegedy *et al.* (2014) and have been shown to produce generic hierarchical visual representations that perform well on a wide variety of vision tasks.

Similarly, in the semantic segmentation panorama there is a tendency to convert the standard deep CNN classifier into Fully Convolutional Networks (FCN) (see e.g., Long *et al.* (2015b); Noh *et al.* (2015); Badrinarayanan *et al.* (2015); Ronneberger *et al.* (2015)) to obtain coarse image representations, which are subsequently upsampled to recover the lost resolution. However, these deep CNNs heavily reduce the input resolution through successive applications of pooling or subsampling layers. While these layers seem to contribute significantly to the desirable invariance properties of deep CNNs, they also make it challenging to use these pre-trained CNNs for tasks such as semantic segmentation, where a per pixel prediction is required.

The information recovery problem has been tackled in a large variety of ways. For instance, Eigen et al. proposed in Eigen and Fergus (2014) a multi-scale architecture that extracts coarse predictions, which are then refined using finer scales. Farabet et al. introduced in Farabet *et al.* (2013) a multi-scale CNN architecture; Hariharan et al. Hariharan

[1] Subsequent but independent work Yan *et al.* (2016) further confirmed the effectiveness of ReSeg, combining a variation of it with CRFs and reporting state of the art results on Pascal VOC Everingham *et al.* (2015).

et al. (2015) combine the information distributed over all layers to make accurate predictions. Other methods such as Long *et al.* (2015b); Badrinarayanan *et al.* (2015) use simple bilinear interpolation to upsample the feature maps of increasingly abstract layers and Ronneberger *et al.* (2015) concatenate the feature maps of the downsampling layers with the feature maps of the upsampling layers to help recover finer information. Finally, more sophisticated upsampling methods, such as unpooling Noh *et al.* (2015); Badrinarayanan *et al.* (2015) or deconvolution Long *et al.* (2015b) are now well established in the literature.

One common issue of all these methods is that they are not specifically designed to take into account and preserve both *local* and *global* contextual dependencies, which have shown to be useful for semantic segmentation tasks Singh and Kosecka (2013); Gatta *et al.* (2014). Rather, these models often employ Conditional Random Fields (CRFs) as a post-processing step to locally smooth the model predictions, but how to tackle long-range contextual dependencies remains relatively unexplored.

Recurrent Neural Networks (RNN) have been used in a variety of tasks for years and have been particularly successful in natural language processing (see, e.g., Mikolov, 2012; Sutskever *et al.*, 2014; Cho *et al.*, 2014a), handwriting recognition and generation (Graves and Schmidhuber, 2009; Graves *et al.*, 2008; Graves, 2013) and speech recognition (Chorowski *et al.*, 2014; Graves and Jaitly, 2014). Only recently RNN and RNN-like models have become popular in the semantic segmentation literature to capture long distance pixel dependencies with the goal to improve semantic segmentation Pinheiro and Collobert (2014); Gatta *et al.* (2014); Chen *et al.* (2015); Byeon *et al.* (2015); Stollenga *et al.* (2015).

For instance, in Pinheiro and Collobert (2014); Gatta *et al.* (2014), CNN are unrolled through different time steps to include semantic feedback connections. In Byeon *et al.* (2015), 2-dimensional Long Short Term Memory (LSTM), which consist of 4 LSTM blocks scanning all directions of an image (left-bottom, left-top, right-top, right-bottom), are introduced to learn long range spatial dependencies. Following a similar direction, in Stollenga *et al.* (2015), multi-dimensional LSTM are swept along different image directions; however, in this case, computations are re-arranged in a pyramidal fashion for efficiency reasons. Finally, in Visin *et al.* (2015), ReNet is proposed to model pixel dependencies in the context of image classification. It is worth noting that an important consequence of the adoption of the ReNet spatial sequences is that they are even more easily parallelizable, as each RNN is dependent only along a horizontal or vertical sequence of pixels; i.e., all rows/columns of pixels can be processed at the same time.

The ReSeg model, that is the subject of this chapter, aims to an *efficient* application of Recurrent Neural Networks RNN to retrieve contextual information from images. The goal of this model is to extend the ReNet architecture Visin *et al.* (2015), originally designed for image classification, to deal with the more ambitious task of semantic segmentation.

As explained in Section 3.2 the ReNet layers can efficiently capture contextual dependencies from images by first sweeping the image horizontally, and then sweeping vertically the feature maps produced by the horizontal processing. The output of a ReNet layer is therefore implicitly encoding the local features at each pixel position with respect to the whole input image, providing a rich feature map of local features conditioned on global information. The intuition behind the proposed ReSeg model is that this can be exploited for more fine detailed tasks than the object recognition originally proposed in ReNet, such as to address the pixel-level task of semantic segmentation.

To decrease the training time and benefit from generic local features, the ReSeg model first preprocesses the input with a FCN, i.e. the intermediate convolutional output of VGG-16 Simonyan and Zisserman (2015). Multiple ReNet layers then work on this generic feature map to extract meaningful local and global pixel dependencies. The resulting structured prediction architecture exploits the local generic features extracted by the CNNs and the ability of RNNs to retrieve distant dependencies to produce a rich encoding of the input. Finally, one or more transposed convolutional layer are stacked on top of the ReNet layers to upsample the intermediate feature maps back to the image size, in order to allow predictions at the pixel level.

The ReSeg architecture is efficient, flexible and suitable for a variety of pixel-level, fine grained tasks, e.g., detecting road signs, cars, pedestrians, in autonomous navigation settings; detecting tumors in fMRI scans or surgical videos; guide autonomous surgical robots by detecting medical instruments in operations; detect faces and other parts of the human body for end users applications such as interactive games or camera autofocus. The source code and model hyperparameters are available on https://github.com/fvisin/reseg.

4.2 Model Description

The architecture of the ReSeg model, motivated in Section 4.1, will be described in detail in this section.

ReSeg builds on top of ReNet Visin *et al.* (2015) and extends it to address the task of semantic segmentation. The model pipeline involves multiple stages. First, the input image is processed with the VGG-16 Simonyan and Zisserman (2015) network, pre-trained on ImageNet Deng *et al.* (2009) and not further fine-tuned. By design, only the first layers of VGG-16 have been used to prevent the resolution of the intermediate feature maps to become too small. The result of this preprocessing is then fed into one or more *ReNet layers* that sweep over the image horizontally and vertically. Finally, one or more *upsampling layers* are employed to resize the last feature maps to the same resolution as the input and a softmax non-linearity is applied to predict the probability distribution over the classes for each pixel.

The following sections analyze in detail each component of the processing pipeline

4.2.1 ReNet layer

As depicted in Figure 4.1, each recurrent layer is composed by 4 RNNs coupled together in such a way to capture the local and global spatial structure of the input data.

Specifically, the model takes as an input an image (or the feature map of the previous layer) $\mathbf{X}$ of elements $x \in \mathbb{R}^{H \times W \times C}$, where H, W and C are respectively the height, width and number of channels (or features) and splits it into $I \times J$ patches $p_{i,j} \in \mathbb{R}^{H_p \times W_p \times C}$. It then sweeps along each of its columns $p_{i,\cdot}$ vertically a first time with two RNNs $f^{\downarrow}$ and $f^{\uparrow}$, with U recurrent units each, that move top-down and bottom-up respectively. Note that the processing of each column is independent and can be done in parallel. This is a very important performance difference of the ReSeg model w.r.t. to other architectures in the literature that enforce to handle the data in a more constrainedly sequential nature.

At every time step each RNN reads the next non-overlapping patch $p_{i,j}$ and, based on

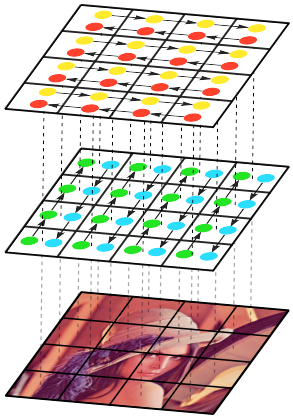


Figure 4.1: *A ReNet layer. The blue and green dots on the input image/feature map represent the steps of $f^{\downarrow}$ and $f^{\uparrow}$ respectively. On the concatenation of the resulting feature maps, $f^{\rightarrow}$ (yellow dots) and $f^{\leftarrow}$ (red dots) are subsequently swept. Their feature maps are finally concatenated to form the output of the ReNet layer, depicted as a blue heatmap in the figure.*

its previous state, emits a projection $o^{\star}_{i,j}$ and updates its state $z^{\star}_{i,j}$:

$$o^{\downarrow}_{i,j} = f^{\downarrow}(z^{\downarrow}_{i-1,j}, p_{i,j}), \text{ for } i = 1, \cdots, I \tag{4.1}$$

$$o^{\uparrow}_{i,j} = f^{\uparrow}(z^{\uparrow}_{i+1,j}, p_{i,j}), \text{ for } i = I, \cdots, 1 \tag{4.2}$$

It has to be stressed that the decision to read non-overlapping patches is a modeling choice to increase the image scan speed and lower the memory usage, but is not a limitation of the architecture.

Once the first two vertical RNNs have processed the whole input X, their projections $o^{\downarrow}_{i,j}$ and $o^{\uparrow}_{i,j}$ are concatenated to obtain a composite feature map $\mathbf{O}^{\updownarrow}$ whose elements $o^{\updownarrow}_{i,j} \in \mathbb{R}^{2U}$ can be seen as the activation of a feature detector at the location (i, j) with respect to all the patches in the j-th column of the input. For simplicity, in the rest of this manuscript this part of the model will be referred to as *vertical recurrent sublayer*.

After obtaining the concatenated feature map $\mathbf{O}^{\updownarrow}$, the model sweeps over each of its rows with a pair of new RNNs $f^{\rightarrow}$ and $f^{\leftarrow}$. Each element of $\mathbf{O}^{\updownarrow}$ is processed individually, rather than grouping them into patches as was done in the vertical recurrent sublayer. This was chosen so that the second recurrent sublayer has the same spatial granularity as the first one, but this is not a constraint of the model and different architectures can be explored.

With a similar but specular procedure as the one adopted for the first sublayer, the network reads one element of the intermediate feature map $o^{\updownarrow}_{i,j}$ at each step and emits two activations coming from two new RNNs that are concatenated into a unique feature map $\mathbf{O}^{\leftrightarrow} = \{h^{\leftrightarrow}_{i,j}\}^{j=1\ldots J}_{i=1\ldots I}$, with $o^{\leftrightarrow}_{i,j} \in \mathbb{R}^{2U}$. Each element $o^{\leftrightarrow}_{i,j}$ of this *horizontal recurrent sublayer* represents the features of one of the input image patches $p_{i,j}$ *with contextual information from the whole image*.

It is trivial to note that it is possible to concatenate many recurrent layers $\mathbf{O}^{(\mathbf{1}\cdots\mathbf{L})}$ one after the other and train them with any optimization algorithm that performs gradient

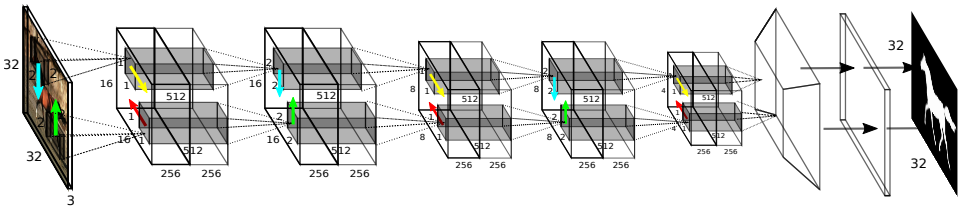


Figure 4.2: *The ReSeg network. For space reasons the pretrained VGG-16 convolutional layers used to preprocess the input to ReSeg are not represented. The first 2 RNNs (blue and green) are applied on 2x2x3 patches of the image, their 16x16x256 feature maps are concatenated and fed as input to the next two RNNs (red and yellow) which read 1x1x512 patches and emit the output of the first ReNet layer. Two similar ReNet layers are stacked, followed by an upsampling layer and a softmax nonlinearity.*

descent, as the composite model is a smooth, continuous function.

The recurrent layers that are the core of this architecture, can be either implemented as vanilla $\tanh$ RNN layers, Gated Recurrent Unit (GRU) layers Cho *et al.* (2014a) or LSTM layers Hochreiter and Schmidhuber (1997). Previous work has shown that the ReNet model can perform well with little concern for the specific recurrent unit used Visin *et al.* (2015). The ReSeg model was tested choosing GRU units over alternative implementations, as they strike a good balance between memory usage and computational power, but nothing prevents from using different kinds of RNN layers.

4.2.2 Upsampling layer

Since by design each recurrent layer processes non-overlapping patches, the size of the last composite feature map is smaller than the size of the initial input $\mathbf{X}$, whenever the patch size is greater than one. To be able to compute a segmentation mask at the same resolution as the ground truth, the prediction has to be expanded back before applying the softmax non-linearity.

Several different methods can be used to this end, e.g., fully connected layers, full convolutions and transposed convolutions. The first is not a good candidate in this domain as it does not take into account the topology of the input, which is essential for this task; the second is not optimal either, as it would require large kernels and stride sizes to upsample by the required factor. Transposed convolutions are both memory and computation efficient, and are the ideal method to tackle this problem.

Transposed convolutions – also known as *fractionally strided convolutions* – have been employed in many works in recent literature Zeiler *et al.* (2011a); Zeiler and Fergus (2014b); Long *et al.* (2015a); Radford *et al.* (2015); Im *et al.* (2016). This method is based on the observation that direct convolutions can be expressed as a dot product between the flattened input and a sparse matrix, whose non-zero elements are elements of the convolutional kernel. The equivalence with the convolution is granted by the connectivity pattern defined by the matrix.

Transposed convolutions apply the transpose of this transformation matrix to the input, resulting in an operation whose input and output shapes are inverted with respect to the original direct convolution. A very efficient implementation of this operation can be

obtained exploiting the gradient operation of the convolution – whose optimized implementation can be found in many of the most popular libraries for neural networks. For an in-depth and comprehensive analysis of each alternative, see Section 2.2.7.

4.3 Experiments

4.3.1 Datasets

The ReSeg architecture has been evaluated on several benchmark datasets. The model has been first tested on the easier case of background/foreground segmentation, where the algorithm is only requested to distinguish between the main subject in the scene and the rest of the image. These preliminary experiments were conducted on the Weizmann Horse and the Oxford Flowers datasets, to then focus on the full task of semantic segmentation on the more challenging Camvid dataset. The rest of this section proceeds as follows: each dataset is briefly introduced, the settings of each experiment are then outlined and finally the results are presented and discussed.

Weizmann Horse

The Weizmann Horse dataset, introduced in Borenstein (2004), is an image segmentation dataset consisting of 329 variable size images in both RGB and gray scale format, matched with an equal number of groundtruth segmentation images, of the same size as the corresponding image. The groundtruth segmentations contain a foreground/background mask of the focused horse, encoded as a real-value between 0 and 255. To convert this into a boolean mask, the data has been thresholded in the center of the range setting all smaller values to 0, and all greater values to 1.

Oxford Flowers 17

The Oxford Flowers 17 class dataset from Nilsback and Zisserman (2006) contains 1363 variable size RGB images, with 848 image segmentations maps associated with a subset of the RGB images. There are 8 unique segmentation classes defined over all maps, including flower, sky, and grass. To build a foreground/background mask, the original segmentation maps have been merged together setting any pixel not marked as class 38 (the flower class) to 0, and setting all the flower class pixels to 1. This binary segmentation task for Oxford Flowers 17 is described in detail in Wu and Kashino (2014).

CamVid Dataset

The Cambridge-driving Labeled Video Database (CamVid) Brostow *et al.* (2009) is a real-world dataset which consists of images recorded from a car with an internally mounted camera, capturing frames of 960×720 RGB pixels per frame, with a recording frame rate of 30 frames per second. A total of ten minutes of video was recorded, and approximately one frame per second has been manually annotated with per pixel class labels, from one of 32 possible classes. A small number of pixels were labelled as void in the original dataset. These do not belong to any of the 32 classes prescribed in the original data, and are ignored during evaluation. The CamVid dataset is split into 367 training, 101 validation and 233

test images for a total of 701 images with corresponding annotations. The results presented in this section used the same subset of 11 class categories as Badrinarayanan *et al.* (2015) and in order to make the experimental setup fully comparable to Badrinarayanan *et al.* (2015), were obtained downsampling all the images by a factor of 2 resulting in a final 480×360 resolution.

4.3.2 Data augmentation and preprocessing

The ReNet model adopted some data augmentation methods to enlarge the dataset by adding synthetic data randomly flipping and shifting the images. In ReSeg instead, this kinds of data augmentation were not employed.

The only data augmentation technique exploited was to randomly invert the colors by changing darker colors into lighter colors with probability 0.5, and vice-versa. This kind of data augmentation technique is only meaningful when working with gray-scale images and was employed for the Weizmann Horse dataset only to improve the segmentation performance on light coloured horses, that in that dataset are much less represented than darker ones. This allowed to significantly improve the performance on Weizmann Horse, probably due to the very limited number of images that caused the loss due to even a single image misclassification to be significant. The same technique would probably not be as effective on bigger and more balanced datasets.

The only other data manipulation technique adopted was to resize the *training* images to allow for training on batches of multiple images with the effect of speeding up the training. The performance loss due to the difference in training and validation/test size was negligible, probably thanks to the adoption of the pretrained VGG-16 layers that can extract meaningful features from the typical image sizes of the adopted datasets.

On the contrary, training benefitted a little from resizing the images on some of the datasets, probably because eliminating unnecessary and easily explained variance can help the model focus on harder characteristics, which generally leads to better performance on the task at hand.

A common choice for resizing is to resize every image to the mean width and height, calculated over the entire dataset of variable size images. This was the strategy adopted for the Weizmann Horse and Oxford Flowers dataset, while for the Camvid dataset the standard Badrinarayanan *et al.* (2015) downsampling factor of 2 was adopted, resulting in a final 480×360 resolution.

It should be noted that all transformations that involve changes in dimensionality or position must also be applied in the same form to the segmentation mask, and great care must be taken (especially during resizing/shifting) not to introduce unexpected errors. It is particularly important to resize the network prediction to the original size of the ground truth and not the opposite, not to misrepresent the segmentation accuracy. This is not a problem for validation and test, as the images are not resized in those cases.

4.3.3 Experimental settings

To gain confidence with the sensitivity of the model to the different hyperparameters, ReSeg was first evaluated on the Weissman Horse and Oxford Flowers datasets on a binary foreground/background segmentation task. Once a good initial setting of the hyperpa-

Method	Global	**Avg IoU**
All foreground baseline	25.4	79.9
All background baseline	74.7	0.0
Kernelized structural SVM Bertelli *et al.* (2011)	94.6	80.1
ReSeg (no VGG)	94.9	79.9
CRF learning Liu *et al.* (2015)	95.7	84.0
PatchCut Yang *et al.* (2015)	95.8	84.0
ReSeg	96.8	**91.6**

Table 4.1: *Weizmann Horses. Per pixel accuracy and average IoU are reported.*

Method	Global	**Avg IoU**
All background baseline	71.0	0.0
All foreground baseline	29.0	29.2
GrabCut Rother *et al.* (2004)	95.9	89.3
Tri-map Wu and Kashino (2014)	96.7	91.7
ReSeg	98	**93.7**

Table 4.2: *Oxford Flowers. Per pixel accuracy and average IoU are reported.*

rameters was found, most of the efforts were spent on the more challenging semantic segmentation task on the CamVid dataset.

The number of hyperparameters of this model is potentially very high, as for each ReNet layer different implementations are possible (namely vanilla RNN, GRU or LSTM), each one with its specific parameters. Furthermore, the number of features, the size of the patches and the initialization scheme have to be defined for each ReNet layer as well as for each transposed convolutional layer. To make it feasible to explore the hyperparameter space, some of the hyperparameters have been fixed by design and the remaining have been finetuned. In the rest of this section, the architectural choices for both sets of parameters will be detailed.

Initial experiments on the upsampling showed that transposed convolutional layers seem to provide a better performance w.r.t. less sophisticated (not learned) tiling strategies such as nearest neighbor tiling or bilinear interpolation when there is enough data to learn the transformation.

All the transposed convolution upsampling layers were followed by a ReLU Krizhevsky *et al.* (2012c) non-linearity and initialized with the fan-in plus fan-out initialization scheme described in Glorot and Bengio (2010). The recurrent weight matrices were instead initialized to be orthonormal, following the procedure defined in Saxe *et al.* (2014). Finally, the stride of the upsampling transposed convolutional layers was constrained to be tied to their filter size.

In the segmentation task, each training image carries classification information for all of its pixels. Differently from the image classification task, small batch sizes provide the model with a good amount of information with sufficient variance to learn and generalize well. We experimented with various batch sizes going as low as processing a single image at the time, obtaining comparable results in terms of performance. In our experiments we kept a fixed batch size of 5, as a compromise between train speed and memory usage. In all our experiments, we used L2 regularization Krogh and Hertz (1992), also known as

Method	Building	Tree	Sky	Car	Sign-Symbol	Road	Pedestrian	Fence	Column-Pole	Side-walk	Bicyclist	Avg class acc	Global acc	**Avg IoU**
Segmentation models														
Super Parsing Tighe and Lazebnik (2013)	**87.0**	67.1	**96.9**	62.7	30.1	95.9	14.7	17.9	1.7	70.0	19.4	51.2	83.3	n/a
Boosting+Higher order Sturgess *et al.* (2009)	84.5	72.6	**97.5**	72.7	34.1	95.3	34.2	45.7	8.1	77.6	28.5	59.2	83.8	n/a
Boosting+Detectors+CRF Ladický *et al.* (2010)	81.5	76.6	96.2	78.7	40.2	93.9	43.0	47.6	14.3	81.5	33.9	62.5	83.8	n/a
Neural Network based segmentation models														
SegNet-Basic (layer-wise training Badrinarayanan *et al.* (????))	75.0	84.6	91.2	82.7	36.9	93.3	55.0	37.5	44.8	74.1	16.0	62.9	84.3	n/a
SegNet-Basic Badrinarayanan *et al.* (2015)	80.6	72.0	93.0	78.5	21.0	94.0	62.5	31.4	36.6	74.0	42.5	62.3	82.8	46.3
SegNet Badrinarayanan *et al.* (2015)	**88.0**	**87.3**	92.3	80.0	29.5	**97.6**	57.2	**49.4**	27.8	84.8	30.7	65.9	88.6	50.2
ReSeg + Class Balance	70.6	84.6	89.6	81.1	**61.0**	95.1	**80.4**	35.6	**60.6**	**86.3**	**60.0**	73.2	83.5	53.7
ReSeg	86.8	84.7	93.0	**87.3**	48.6	**98.0**	63.3	20.9	35.6	**87.3**	43.5	68.1	88.7	**58.8**
Sub-model averaging														
Bayesian SegNet-Basic Kendall *et al.* (2015)	75.1	68.8	91.4	77.7	52.0	92.5	71.5	44.9	52.9	79.1	69.6	70.5	81.6	55.8
Bayesian SegNet Kendall *et al.* (2015)	80.4	85.5	90.1	86.4	67.9	93.8	73.8	64.5	50.8	91.7	54.6	76.3	86.9	63.1

Table 4.3: *CamVid. The table reports the per-class accuracy, the average per-class accuracy, the global accuracy and the average intersection over union. The best values and the values within* 1 *point from the best are highlighted in bold for each column. For completeness the Bayesian Segnet models are reported even if they are not directly comparable to the others as they perform a form of model averaging.*

Model	ps_{RE}	d_{RE}	fs_{UP}	d_{UP}	Building	Tree	Sky	Car	Sign-Symbol	Road	Pedestrian	Fence	Column-Pole	Side-walk	Bicyclist	Avg class acc	Global acc	**Avg IoU**
ReSeg + LCN	$(2 \times 2), (1 \times 1)$	(100, 100)	(2×2)	(50, 50)	81.5	80.3	**94.7**	78.1	42.8	**97.4**	53.5	34.3	36.8	68.9	47.9	65.1	84.8	52.6
ReSeg + Class Balance	$(2 \times 2), (1 \times 1)$	(100, 100)	(2×2)	(50, 50)	70.6	**84.6**	89.6	81.1	**61.0**	95.1	**80.4**	**35.6**	**60.6**	**86.3**	**60.0**	73.2	83.5	53.7
ReSeg	$(2 \times 2), (1 \times 1)$	(100, 100)	(2×2)	(50, 50)	**86.8**	**84.7**	93.0	**87.3**	48.6	**98.0**	63.3	20.9	35.6	**87.3**	43.5	68.1	88.7	**58.8**

Table 4.4: *Comparison of the performance of different hyperparameter on CamVid.*

weight decay, set to 0.001 to avoid instability at the end of training. We trained all our models with the Adadelta Zeiler (2012) optimization algorithm, for its desired property of not requiring a specific hyperparameter tuning. The effect of Batch Normalization in RNNs has been a focus of attention Laurent *et al.* (2015), but it does not seem to provide a reliable improvement in performance, so we decided not to adopt it.

In the experiments, we varied the number of ReNet layers and the number of upsampling transposed convolutional layers, each of them defined respectively by the number of features $d_{\text{RE}}(l)$ and $d_{\text{UP}}(l)$, the size of the input patches (or equivalently of the filters) $ps_{\text{RE}}(l)$ and $fs_{\text{UP}}(l)$.

4.3.4 Results

In Table 4.1, we report the results on the Weizmann Horse dataset. On this dataset, we verified the assumption that processing the input image with some pre-trained convolutional layers from VGG-16 could ease the learning. Specifically, we restricted ourselves to only using the first 7 convolutional layers from VGG, as we only intended to extract some low-level generic features and learn the task-specific high-level features with the ReNet layers. The results indeed show an increase in terms of average Intersection over Union (*IoU*) when these layers are being used, confirming our hypothesis.

Table 4.2 shows the results for Oxford Flowers dataset, when using the full ReSeg architecture (i.e., including VGG convolutional layers). As shown in the table, our method clearly outperforms the state-of-the-art both in terms of global accuracy and average IoU.

Table 4.3 presents the results on CamVid dataset using the full ReSeg architecture. Our model exhibits state-of-the-art performance in terms of IoU when compared to both standard segmentation methods and neural network based methods, showing an increase of 17% w.r.t. to the recent SegNet model. It is worth highlighting that incorporating sub-model averaging to SegNet model, as in Kendall *et al.* (2015), boosts the original model performance, as expected. Therefore, introducing sub-model averaging to ReSeg would also presumably result in significant performance increase. However, this remains to be tested.

4.4 Discussion

As reported in the previous section, our experiments on the Weizmann Horse dataset show that processing the input images with some layers of VGG-16 pre-trained network improves the results. In this setting, pre-processing the input with Local Contrast Normalization (LCN) does not seem to give any advantage (see Table 4.4). We did not use any other kind of pre-processing.

While on both the Weizmann Horse and the Oxford Flowers datasets we trained on a binary background/foreground segmentation task, on CamVid we addressed the full semantic segmentation task. In this setting, when the dataset is highly imbalanced, the segmentation performance of some classes can drop significantly as the network tries to maximize the score on the high-occurrence classes, *de facto* ignoring the low-occurrence ones. To overcome this behaviour, we added a term to the cross-entropy loss to bias the prediction towards the low-occurrence classes. We use *median frequency balancing* Eigen and Fergus (2014), which re-weights the class predictions by the ratio between the median of the frequencies of the classes (computed on the training set) and the frequency of each class. This increases the score of the low frequency classes (see Table 4.4) at the price of a more noisy segmentation mask, as the probability of the underrepresented classes is overestimated and can lead to an increase in misclassified pixels in the output segmentation mask, as shown in Figure 4.3.

On all datasets we report the per-pixel accuracy (*Global acc*), computed as the percentage of true positives w.r.t. the total number of pixels in the image, and the average per-class Intersection over Union (*Avg IoU*), computed on each class as true positive divided by the sum of true positives, false positives and false negatives and then averaged. In the full semantic segmentation setting we also report the per-class accuracy and the average per-class accuracy (*Avg class acc*).

4.5 Conclusion

We introduced the ReSeg model, an extension of the ReNet model for image semantic segmentation. The proposed architecture shows state-of-the-art performances on CamVid, a widely used dataset for urban scene semantic segmentation, as well as on the much smaller Oxford Flowers dataset. We also report state-of-the-art performances on the Weizmann Horses.

In our analysis, we discuss the effects of applying some layers of VGG-16 to process the input data, as well as those of introducing a class balancing term in the cross-entropy loss function to help the learning of under-represented classes. Notably, it is sufficient to

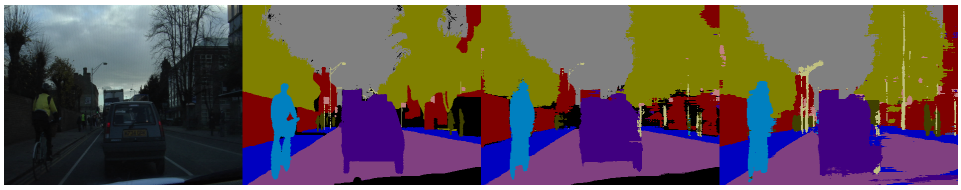


Figure 4.3: *Camvid segmentation example with and without class balancing. From the left: input image, ground truth segmentation, ReSeg segmentation, ReSeg segmentation with class balancing. Class balancing improves the low frequency classes as e.g., the street lights, at the price of a worse overall segmentation.*

process the input images with just a few layers of VGG-16 for the ReSeg model to gracefully handle the semantic segmentation task, confirming its ability to encode contextual information and long term dependencies.

CHAPTER 5

Convolutional RNNs for Video Semantic Segmentation

5.1 Introduction

Write something about video segmentation

CHAPTER 6

Conclusion

We made it. That gotta count for something, uh?

Bibliography

Abdel-Hamid, O., Mohamed, A., Jiang, H., and Penn, G. (2012). Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*.

Arpit, D., Zhou, Y., Kota, B. U., and Govindaraju, V. (2016). Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. *arXiv preprint arXiv:1603.01431*.

Azzopardi, G. and Petkov, N. (2013). Trainable COSFIRE filters for keypoint detection and pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, **35**(2), 490–503.

Badrinarayanan, V., Handa, A., and Cipolla, R. (????). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Robust Semantic Pixel-Wise Labelling.

Badrinarayanan, V., Handa, A., and Cipolla, R. (2015). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. page 5.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. Technical report, arXiv:1409.0473.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR 2015)*.

Belongie, S., Malik, J., and Puzicha, J. (2002). Shape matching and object recognition using shape contexts. *IEEE transactions on pattern analysis and machine intelligence*, **24**(4), 509–522.

Bengio, I. G. Y. and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, **5**(2), 157–166.

Bertelli, L., Yu, T., Vu, D., and Gokturk, B. (2011). Kernelized structural svm learning for supervised object segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 2153–2160. IEEE.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Borenstein, E. (2004). Combining top-down and bottom-up segmentation. In *In Proceedings IEEE workshop on Perceptual Organization in Computer Vision, CVPR*, page 46.

Boureau, Y., Bach, F., LeCun, Y., and Ponce, J. (2010a). Learning mid-level features for recognition. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'10)*. IEEE.

Boureau, Y., Ponce, J., and LeCun, Y. (2010b). A theoretical analysis of feature pooling in vision algorithms. In *Proc. International Conference on Machine learning (ICML'10)*.

Boureau, Y., Le Roux, N., Bach, F., Ponce, J., and LeCun, Y. (2011). Ask the locals: multi-way local pooling for image recognition. In *Proc. International Conference on Computer Vision (ICCV'11)*. IEEE.

Brostow, G. J., Shotton, J., Fauqueur, J., and Cipolla, R. (2008). Segmentation and recognition using structure from motion point clouds. In *ECCV (1)*, pages 44–57.

Brostow, G. J., Fauqueur, J., and Cipolla, R. (2009). Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, **30**(2), 88–97.

Byeon, W., Breuel, T. M., Raue, F., and Liwicki, M. (2015). Scene labeling with lstm recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3547–3555.

Chen, L.-C., Barron, J. T., Papandreou, G., Murphy, K., and Yuille, A. L. (2015). Semantic image segmentation with task-specific edge detection using cnns and a discriminatively trained domain transform. *arXiv preprint arXiv:1511.03328*.

Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014a). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*. to appear.

Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014b). On the properties of neural machine translation: Encoder–Decoder approaches. In *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*.

Chorowski, J., Bahdanau, D., Cho, K., and Bengio, Y. (2014). End-to-end continuous speech recognition using attention-based recurrent nn: First results. arXiv:1412.1602.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. NIPS'2014 Deep Learning workshop, arXiv 1412.3555.

Ciresan, D., Meier, U., Masci, J., and Schmidhuber, J. (2012a). Multi-column deep neural network for traffic sign classification. *Neural Networks*, **32**, 333–338.

Ciresan, D., Meier, U., and Schmidhuber, J. (2012b). Multi-column deep neural networks for image classification. Technical report, arXiv:1202.2745.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep big simple neural nets excel on handwritten digit recognition. *arXiv*, **abs/1003.0358**.

Cooijmans, T., Ballas, N., Laurent, C., and Courville, A. (2016). Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Drachman, D. A. (2005). Do we have brain to spare? *Neurology*, **64**(12), 2004–2005.

Dufournaud, Y., Schmid, C., and Horaud, R. (2000). Matching images with different resolutions. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 1, pages 612–618. IEEE.

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Eigen, D. and Fergus, R. (2014). Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. *CoRR*, **abs/1411.4734**.

Erhan, D., Szegedy, C., Toshev, A., and Anguelov, D. (2014). Scalable object detection using deep neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 2155–2162, Washington, DC, USA. IEEE Computer Society.

Everingham, M., Eslami, S. M. A., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. (2015). The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, **111**(1), 98–136.

Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2013). Learning hierarchical features for scene labeling. *IEEE TPAMI*, **35**(8), 1915–1929.

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

Gatta, C., Romero, A., and van de Weijer, J. (2014). Unrolling loopy top-down semantic feedback in convolutional deep networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2014, Columbus, OH, USA, June 23-28, 2014*, pages 504–511.

Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE.

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

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *AISTATS'2011*.

Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1319–1327.

Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. (2014). Multi-digit number recognition from Street View imagery using deep convolutional neural networks. In *International Conference on Learning Representations*.

Graham, B. (2014a). Fractional max-pooling. *arXiv*, **abs/1412.6071**.

Graham, B. (2014b). Spatially-sparse convolutional neural networks. *arXiv*, **abs/1409.6070**.

Graves, A. (2013). Generating sequences with recurrent neural networks. Technical report, arXiv:1308.0850.

Graves, A. and Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. In *ICML'2014*.

Graves, A. and Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *NIPS'2008*, pages 545–552.

Graves, A., Liwicki, M., Bunke, H., Schmidhuber, J., and Fernández, S. (2008). Unconstrained on-line handwriting recognition with recurrent neural networks. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *NIPS'2007*, pages 577–584.

Hariharan, B., Arbeláez, P., Girshick, R., and Malik, J. (2015). Hypercolumns for object segmentation and fine-grained localization. In *Computer Vision and Pattern Recognition (CVPR)*.

Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Citeseer.

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley, New York.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv*, **abs/1207.0580**.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, **9**(8), 1735–1780.

Im, D. J., Kim, C. D., Jiang, H., and Memisevic, R. (2016). Generating images with recurrent adversarial networks. *arXiv preprint arXiv:1602.05110*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *ICCV'09*.

Kendall, A., Badrinarayanan, V., and Cipolla, R. (2015). Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv:*`1412.6980 [cs.LG]`.

Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012a). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012b). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012c). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*, pages 950–957. Morgan Kaufmann.

Ladický, L., Sturgess, P., Alahari, K., Russell, C., and Torr, P. H. S. (2010). What, where and how many? Combining object detectors and CRFs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **6314 LNCS**(PART 4), 424–437.

Laurent, C., Pereyra, G., Brakel, P., Zhang, Y., and Bengio, Y. (2015). Batch normalized recurrent neural networks. *CoRR*, **abs/1510.01378**.

Le Cun, Y., Bottou, L., and Bengio, Y. (1997). Reading checks with multilayer graph transformer networks. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 1, pages 151–154. IEEE.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. **1**(4), 541–551.

LeCun, Y., Haffner, P., Bottou, L., and Bengio, Y. (1999). Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, pages 319–345. Springer.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, **521**, 436–444.

Lee, C., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. (2014). Deeply-supervised nets. *arXiv*, **abs/1409.5185**.

Lin, M., Chen, Q., and Yan, S. (2014). Network in network. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*.

Linnainmaa, S. (1970). *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors.* Master's thesis, Univ. Helsinki.

Liu, F., Lin, G., and Shen, C. (2015). Crf learning with cnn features for image segmentation. *Pattern Recognition*.

Long, J., Shelhamer, E., and Darrell, T. (2015a). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440.

Long, J., Shelhamer, E., and Darrell, T. (2015b). Fully convolutional networks for semantic segmentation. *CVPR (to appear)*.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, **60**(2), 91–110.

Mairal, J., Koniusz, P., Harchaoui, Z., and Schmid, C. (2014). Convolutional kernel networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2627–2635.

Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with Hessian-free optimization. In *Proc. ICML'2011*. ACM.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, **5**, 115–133.

Mikolajczyk, K. and Schmid, C. (2001). Indexing based on scale invariant interest points. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 1, pages 525–531. IEEE.

Mikolajczyk, K. and Schmid, C. (2005). A performance evaluation of local descriptors. *IEEE transactions on pattern analysis and machine intelligence*, **27**(10), 1615–1630.

Mikolov, T. (2012). *Statistical Language Models based on Neural Networks*. Ph.D. thesis, Brno University of Technology.

Minsky, M. L. and Papert, S. A. (1969). *Perceptrons*. MIT Press, Cambridge.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proc. 27th International Conference on Machine Learning*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. Deep Learning and Unsupervised Feature Learning Workshop, NIPS.

Nilsback, M.-E. and Zisserman, A. (2006). A visual vocabulary for flower classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 1447–1454.

Noh, H., Hong, S., and Han, B. (2015). Learning deconvolution network for semantic segmentation. *arXiv preprint arXiv:1505.04366*.

Olah, C. (2015). Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *ICML'2013*.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2014). How to construct deep recurrent neural networks. In *ICLR*.

Pinheiro, P. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. *JMLR*, **1**(32), 82–90.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Ronneberger, O., P.Fischer, and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer. (available on arXiv:1505.04597 [cs.CV]).

Rosenblatt, F. (1957). The perceptron — a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, N.Y.

Rother, C., Kolmogorov, V., and Blake, A. (2004). Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics (TOG)*, **23**(3), 309–314.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning representations by back-propagating errors. *Nature*, **323**, 533–536.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, **323**(6088), 533–536.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). ImageNet Large Scale Visual Recognition Challenge.

Sainath, T. N., Mohamed, A.-r., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for lvcsr. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8614–8618. IEEE.

Sainath, T. N., Vinyals, O., Senior, A., and Sak, H. (2015). Convolutional, long short-term memory, fully connected deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*.

Saxe, A., Koh, P. W., Chen, Z., Bhand, M., Suresh, B., and Ng, A. (2011). On random weights and unsupervised feature learning. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 1089–1096, New York, NY, USA. ACM.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*.

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. *International Conference on Learning Representations*.

Silver, D. and Hassabis, D. (2016). Alphago: Mastering the ancient game of go with machine learning. *Research Blog*.

Simard, P. Y., Steinkraus, D., and Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *7th International Conference on Document Analysis and Recognition (ICDAR 2003), 2-Volume Set, 3-6 August 2003, Edinburgh, Scotland, UK*, pages 958–962.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *ICLR*.

Singh, G. and Kosecka, J. (2013). Nonparametric scene parsing with adaptive feature relevance and semantic context. In *2013 IEEE Conference on Computer Vision and Pattern Recognition, Portland, OR, USA, June 23-28, 2013*, pages 3151–3157.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 2960–2968.

Springenberg, J. T. and Riedmiller, M. A. (2013). Improving deep neural networks with probabilistic maxout units. *arXiv*, **abs/1312.6116**.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15**, 1929–1958.

Srivastava, N., Mansimov, E., and Salakhutdinov, R. (2015). Unsupervised learning of video representations using lstms. *CoRR, abs/1502.04681*, **2**.

Stollenga, M. F., Byeon, W., Liwicki, M., and Schmidhuber, J. (2015). Parallel multi-dimensional lstm, with application to fast biomedical volumetric image segmentation. In *Advances in Neural Information Processing Systems*, pages 2980–2988.

Sturgess, P., Alahari, K., Ladicky, L., and Torr, P. H. S. (2009). Combining Appearance and Structure from Motion Features for Road Scene Understanding. *Procedings of the British Machine Vision Conference 2009*, pages 62.1–62.11.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *NIPS'2014*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*.

Szegedy, C., Ioffe, S., and Vanhoucke, V. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*.

Tighe, J. and Lazebnik, S. (2013). Superparsing: Scalable nonparametric image parsing with superpixels. *International Journal of Computer Vision*, **101**(2), 329–349.

Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: A large dataset for non-parametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **30**(11), 1958–1970.

Tóth, L. (2014). Combining time-and frequency-domain convolution in convolutional neural network-based phone recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 190–194. IEEE.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2014). Show and tell: a neural image caption generator. arXiv 1411.4555.

Visin, F., Kastner, K., Cho, K., Matteucci, M., Courville, A., and Bengio, Y. (2015). Renet: A recurrent neural network based alternative to convolutional networks. *arXiv preprint arXiv:1505.00393*.

Visin, F., Ciccone, M., Romero, A., Kastner, K., Cho, K., Bengio, Y., Matteucci, M., and Courville, A. (2016). Reseg: A recurrent neural network-based model for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

Wan, L., Zeiler, M. D., Zhang, S., LeCun, Y., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1058–1066.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Harvard University.

Wu, X. and Kashino, K. (2014). Tri-map self-validation based on least gibbs energy for foreground segmentation. In *Proceedings of the British Machine Vision Conference*. BMVA Press.

Xingjian, S., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-k., and Woo, W.-c. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in Neural Information Processing Systems*, pages 802–810.

Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. arXiv:1502.03044.

Yan, Z., Zhang, H., Jia, Y., Breuel, T., and Yu, Y. (2016). Combining the best of convolutional layers and recurrent layers: A hybrid network for semantic segmentation. *CoRR*, **abs/1603.04871**.

Yang, J., Price, B., Cohen, S., Lin, Z., and Yang, M.-H. (2015). Patchcut: Data-driven object segmentation via local shape transfer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1770–1778.

Yao, L., Torabi, A., Cho, K., Ballas, N., Pal, C., Larochelle, H., and Courville, A. (2015). Video description generation incorporating spatio-temporal features and a soft-attention mechanism. *arXiv:1502.08029*.

Zeiler, M., Taylor, G., and Fergus, R. (2011a). Adaptive deconvolutional networks for mid and high level feature learning. In *Proc. International Conference on Computer Vision (ICCV'11)*, pages 2146–2153. IEEE.

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. Technical report, arXiv 1212.5701.

Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. *arXiv*, **abs/1301.3557**.

Zeiler, M. D. and Fergus, R. (2014a). Visualizing and understanding convolutional networks. In *Computer vision–ECCV 2014*, pages 818–833. Springer.

Zeiler, M. D. and Fergus, R. (2014b). Visualizing and understanding convolutional networks. In *ECCV'14*.

Zeiler, M. D., Taylor, G. W., and Fergus, R. (2011b). Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2018–2025. IEEE.