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

Contents

CHAPTER 1

Introduction

1.1 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.* (2015a), semantic segmentation Chen *et al.* (2015); Visin *et al.* (2015b), 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.* (2015a) 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.*, 2015b) 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

2.1 Introduction

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 we know 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. For a detailed overview of Machine Learning, we suggest the interested reader to refer to Bishop (2006); Bengio and Courville (2016)

2.2 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.

2.2.1 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).

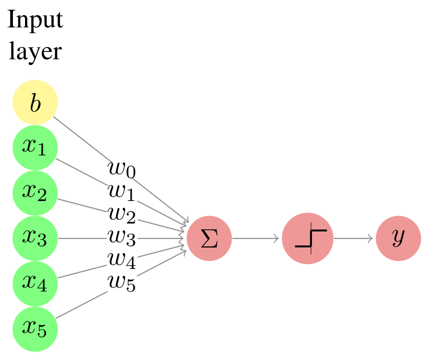


Figure 2.1: *A representation of the Perceptron.*

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.

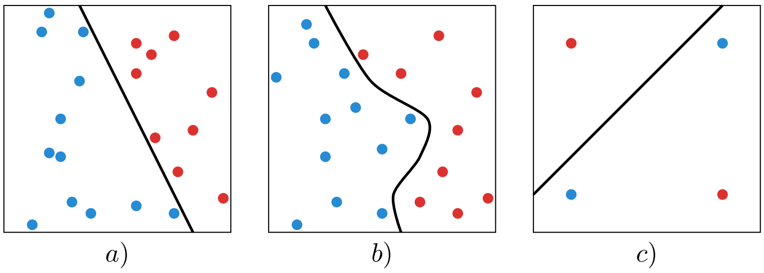


Figure 2.2: *a) A linearly separable problem. b) A nonlinearly separable problem. c) The XOR problem with a tentative solution that fails at separating the points of the space.*

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.* (1986) rediscovered the backpropagation algorithm (Linnainmaa, 1970; Werbos, 1974) that allowed to train ANNs composed by multiple layers by performing gradient descent (see Section 2.2.4). Since then ANNs have been constantly focus of study and innovation and established the state of the art in several domains.

2.2.2 MultiLayer Perceptron

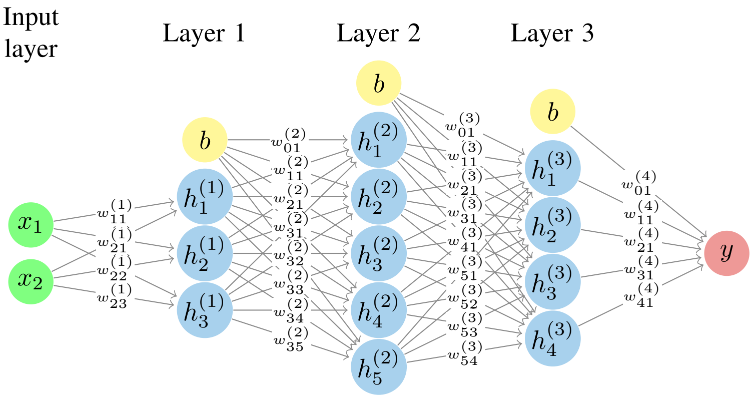


Figure 2.3: *A MultiLayer Perceptron. The sum and the nonlinearity nodes have been omitted for the sake of clarity.*

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

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

Although very common and widely used, the MLP is not the only architecture for ANNs. In section 2.3 and section 2.4 two of the most used alternatives will be described. Generally, each layer of an ANN computes some activation based on its input and a nonlinear 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.

2.2.3 Activation functions

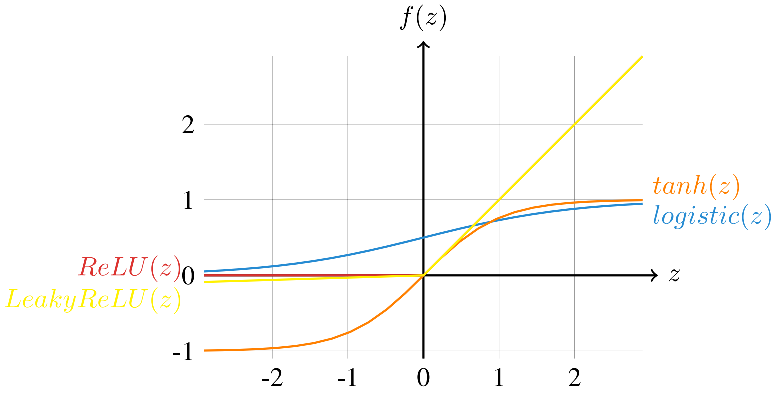


Figure 2.4: *Some of the most common activation functions: sigmoid, tanh, ReLU and Leaky ReLU*

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:

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

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

- *Saturation kills the gradient:* backpropagation Section 2.2.4 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.

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

2.2.4 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.

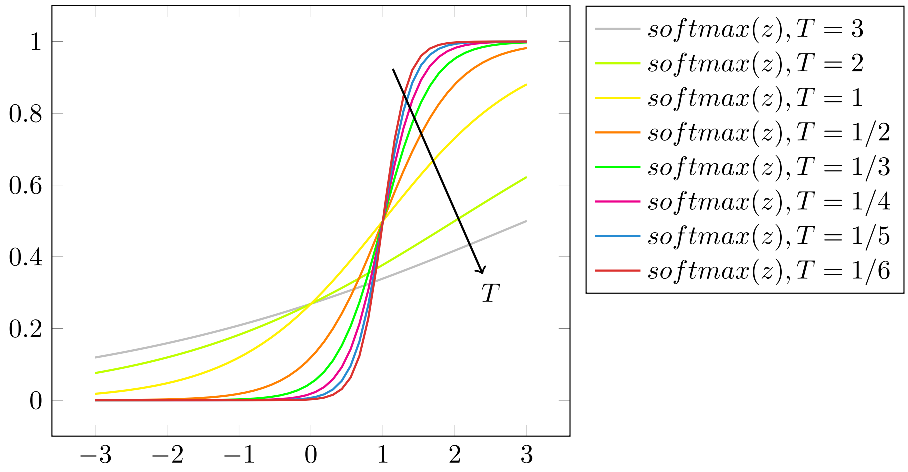


Figure 2.5: *The behaviour of softmax as temperature T grows. The plots have been obtained considering a bidimensional input setting where the preactivation associated to the second class z_1 is always* 1. *As T increases, the function becomes steeper.*

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 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

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

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)

2.3 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).

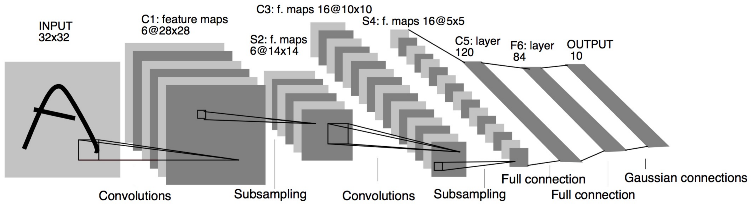


Figure 2.6: *The LeNet architecture for handwritten characters recognition.*

Section 2.2.2 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 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.*, 2011; Zeiler and Fergus, 2014; Long *et al.*, 2015; Radford *et al.*, 2015; Visin *et al.*, 2015b; 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 2.3.1 followed by a description of pooling in Section 2.3.2. The rest of this section will focus on the arithmetic to compute the output shape of a convolution given its parameters. In particular Section 2.3.7 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).

2.3.1 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.

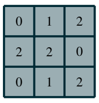
Images, sound clips and many other similar kinds of data have an intrinsic structure. More formally, they share these important properties:

- 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



0	1	2
2	2	0
0	1	2

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

[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.

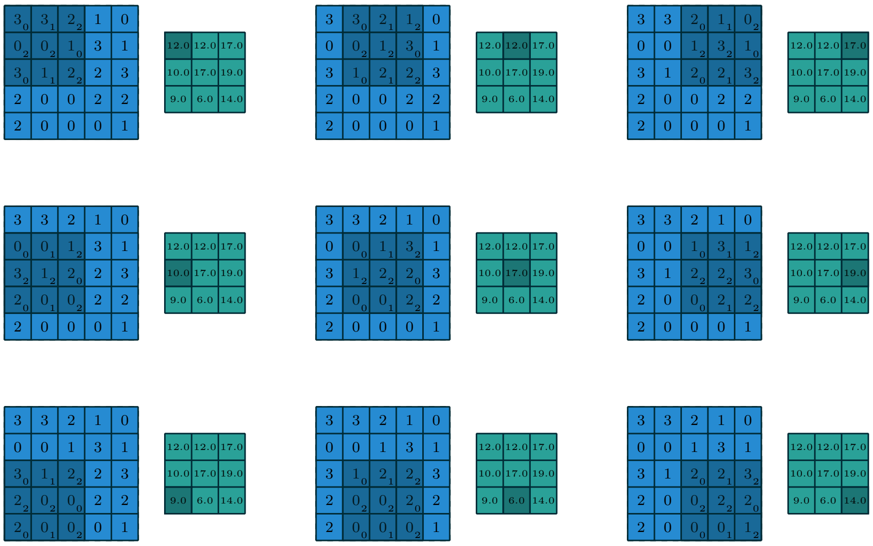


Figure 2.7: *Computing the output values of a discrete convolution.*

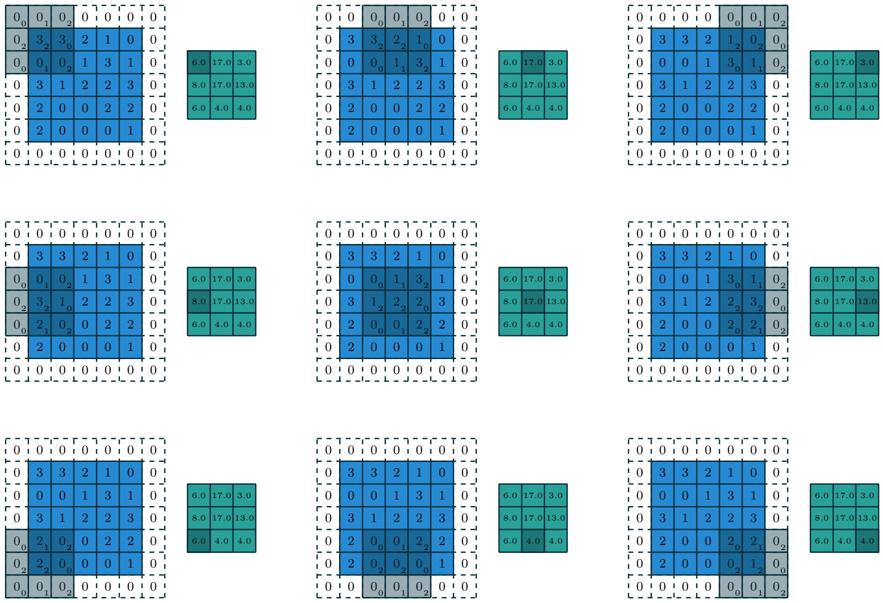


Figure 2.8: *Computing the output values of a discrete convolution for* $N = 2$, $i_1 = i_2 = 5$, $k_1 = k_2 = 3$, $s_1 = s_2 = 2$, *and* $p_1 = p_2 = 1$.

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).

2.3.2 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.

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.

2.3.3 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.

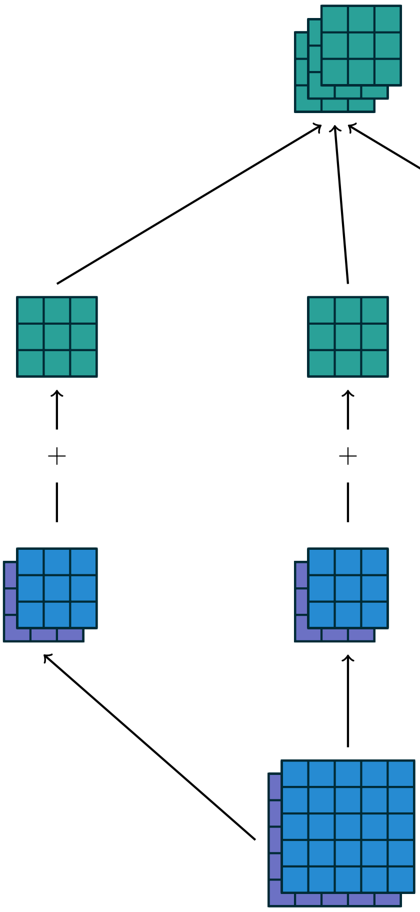


Figure 2.9: *A convolution mapping from two input feature maps to three output feature maps using a* $3 \times 2 \times 3 \times 3$ *collection of kernels* $\mathbf{w}$*. In the left pathway, input feature map 1 is convolved with kernel* $\mathbf{w}_{1,1}$ *and input feature map 2 is convolved with kernel* $\mathbf{w}_{1,2}$*, 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.*

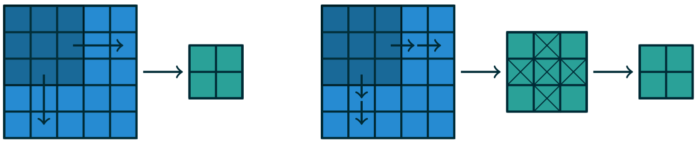


Figure 2.10: *An alternative way of viewing strides. Instead of translating the* 3×3 *kernel by increments of* $s = 2$ *(left), the kernel is translated by increments of* 1 *and only one in* $s = 2$ *output elements is retained (right).*

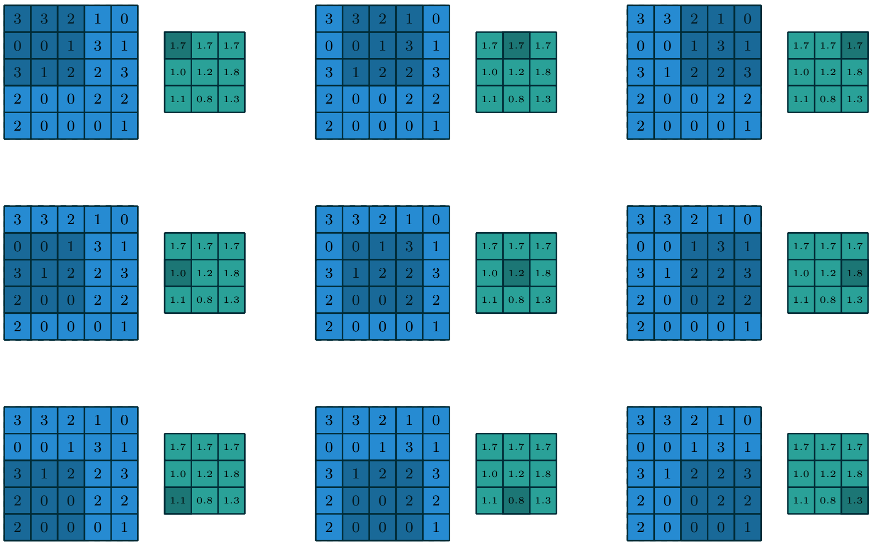


Figure 2.11: *Computing the output values of a* 3×3 *average pooling operation on a* 5×5 *input using* 1×1 *strides.*

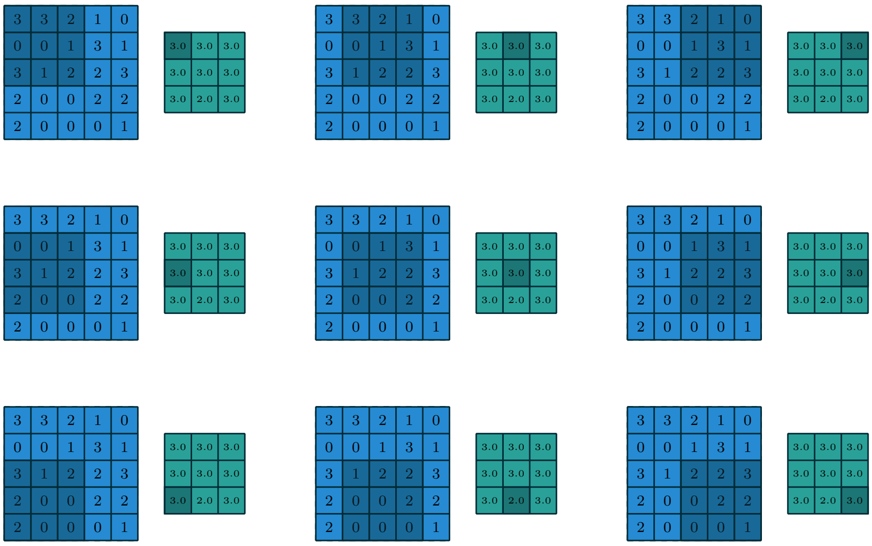


Figure 2.12: *Computing the output values of a* 3×3 *max pooling operation on a* 5×5 *input using* 1×1 *strides.*

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 $p = 0$,*

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

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) $p = 1$.

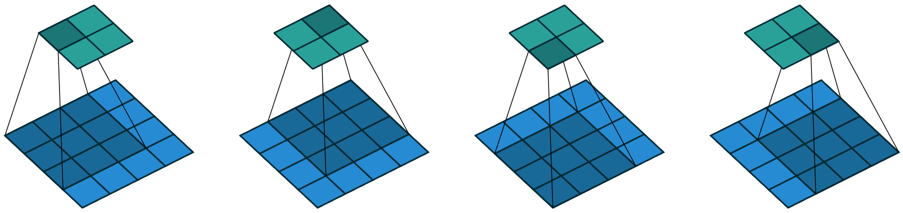


Figure 2.13: *(No padding, unit strides) Convolving a* 3×3 *kernel over a* 4×4 *input using unit strides (i.e.,* $i = 4$, $k = 3$, $s = 1$ *and* $p = 0$*).*

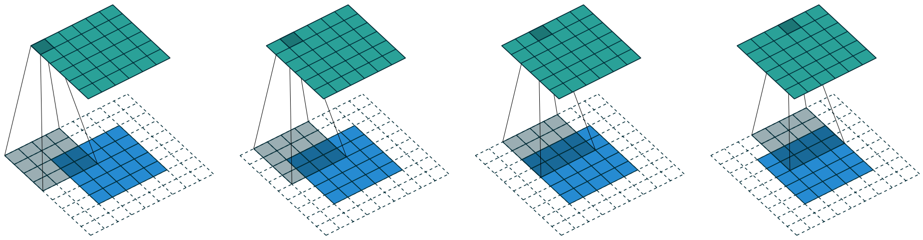


Figure 2.14: *(Arbitrary padding, unit strides) Convolving a* 4×4 *kernel over a* 5×5 *input padded with a* 2×2 *border of zeros using unit strides (i.e.,* $i = 5$, $k = 4$, $s = 1$ *and* $p = 2$*).*

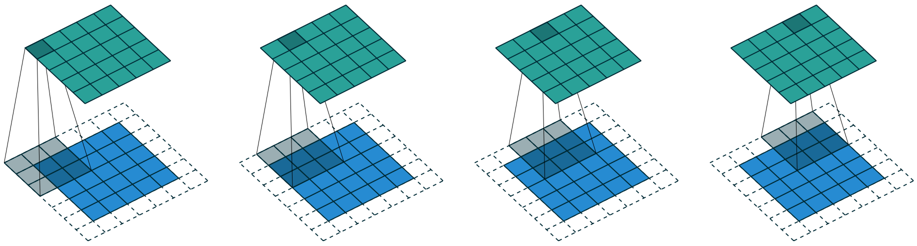


Figure 2.15: *(Half padding, unit strides) Convolving a* 3×3 *kernel over a* 5×5 *input using half padding and unit strides (i.e.,* $i = 5$, $k = 3$, $s = 1$ *and* $p = 1$*).*

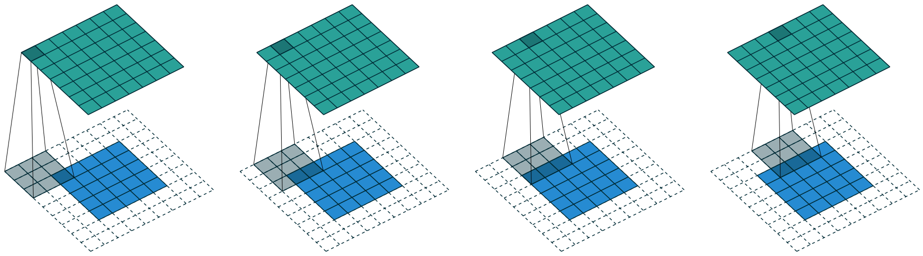


Figure 2.16: *(Full padding, unit strides) Convolving a* 3×3 *kernel over a* 5×5 *input using full padding and unit strides (i.e.,* $i = 5$, $k = 3$, $s = 1$ *and* $p = 2$*).*

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) $p = 2$.

2.3.4 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).

2.3.5 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 $j = i + a, \quad a \in \{0, \dots, s - 1\}$ 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*.

2.3.6 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).

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.

2.3.7 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.

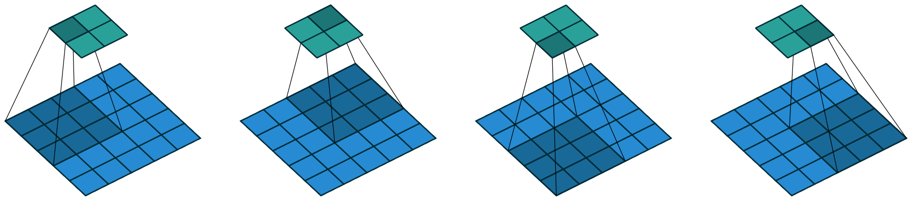


Figure 2.17: *(No zero padding, arbitrary strides) Convolving a* 3×3 *kernel over a* 5×5 *input using* 2×2 *strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 2$ *and* $p = 0$*).*

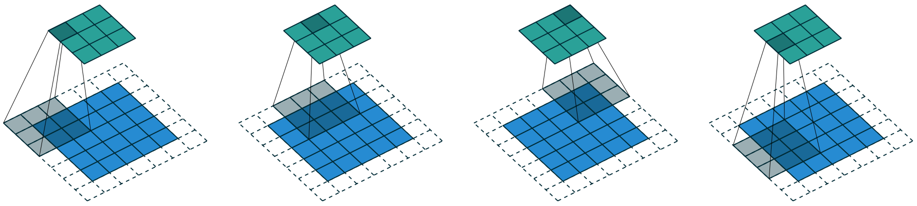


Figure 2.18: *(Arbitrary padding and strides) Convolving a* 3×3 *kernel over a* 5×5 *input padded with a* 1×1 *border of zeros using* 2×2 *strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 2$ *and* $p = 1$*).*

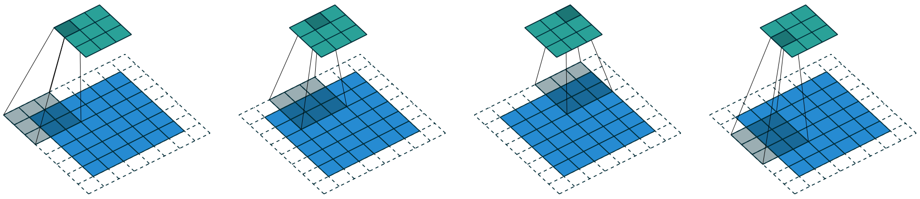
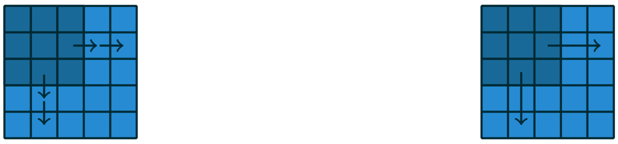


Figure 2.19: *(Arbitrary padding and strides) Convolving a* 3×3 *kernel over a* 6×6 *input padded with a* 1×1 *border of zeros using* 2×2 *strides (i.e.,* $i = 6$*,* $k = 3$*,* $s = 2$ *and* $p = 1$*). In this case, the bottom row and right column of the zero padded input are not covered by the kernel.*



(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.*

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.

2.3.8 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.

2.3.9 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]

[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.

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.

2.3.10 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 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.

[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.

2.3.11 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*
>
> $$o' = i' + (k - 1) - 2p.$$

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) $p = 1$.

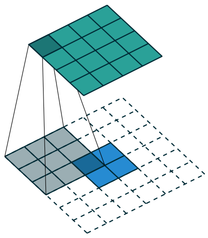
Full padding, transposed Knowing that the equivalent convolution of the transpose of a non-padded convolution involves full padding, it is unsurprising that the equivalent of the transpose of a fully padded convolution is a non-padded convolution:

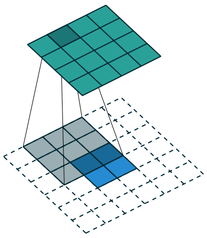
> **Relationship 11.** *A convolution described by $s = 1$, k and $p = k - 1$ has an associated transposed convolution described by $k' = k$, $s' = s$ and $p' = 0$ and its output size is*
>
> $$\begin{aligned} o' &= i' + (k - 1) - 2p \\ &= i' - (k - 1) \end{aligned}$$

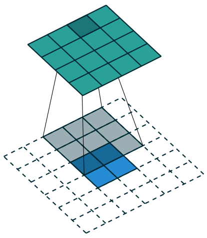
Figure 2.24 provides an example for $i = 5$, $k = 3$ and (therefore) $p = 2$.

2.3.12 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







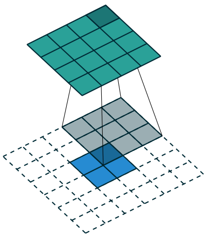
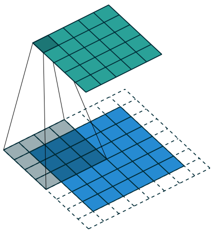
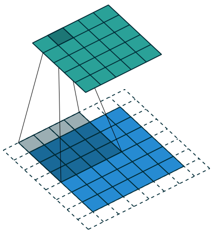
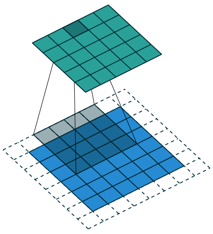


Figure 2.21: *The transpose of convolving a* 3×3 *kernel over a* 4×4 *input using unit strides (i.e.,* $i = 4$, $k = 3$, $s = 1$ *and* $p = 0$*). It is equivalent to convolving a* 3×3 *kernel over a* 2×2 *input padded with a* 2×2 *border of zeros using unit strides (i.e.,* $i' = 2$, $k' = k$, $s' = 1$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







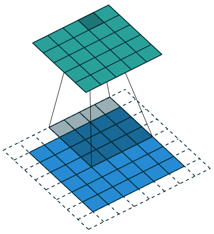
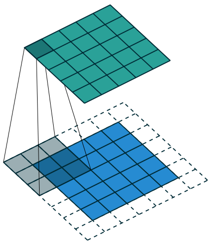
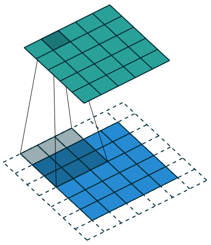
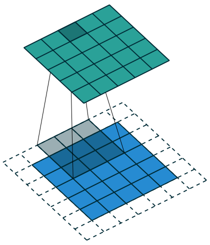


Figure 2.22: *The transpose of convolving a* 4×4 *kernel over a* 5×5 *input padded with a* 2×2 *border of zeros using unit strides (i.e.,* $i = 5$, $k = 4$, $s = 1$ *and* $p = 2$*). It is equivalent to convolving a* 4×4 *kernel over a* 6×6 *input padded with a* 1×1 *border of zeros using unit strides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







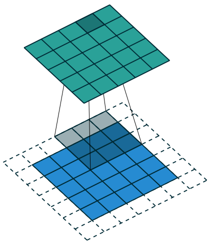


Figure 2.23: *The transpose of convolving a* 3×3 *kernel over a* 5×5 *input using half padding and unit strides (i.e.,* $i = 5$, $k = 3$, $s = 1$ *and* $p = 1$*). It is equivalent to convolving a* 3×3 *kernel over a* 5×5 *input using half padding and unit strides (i.e.,* $i' = 5$, $k' = k$, $s' = 1$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

$s < 1$. 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:

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

2.3.13 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 $a \in \{0, \ldots, s - 1\}$ 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 $p = 1$.

[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.

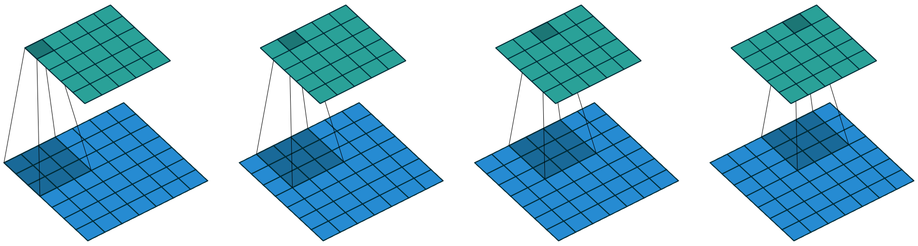


Figure 2.24: *The transpose of convolving a* 3×3 *kernel over a* 5×5 *input using full padding and unit strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 1$ *and* $p = 2$*). It is equivalent to convolving a* 3×3 *kernel over a* 7×7 *input using unit strides (i.e.,* $i' = 7$*,* $k' = k$*,* $s' = 1$ *and* $p'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

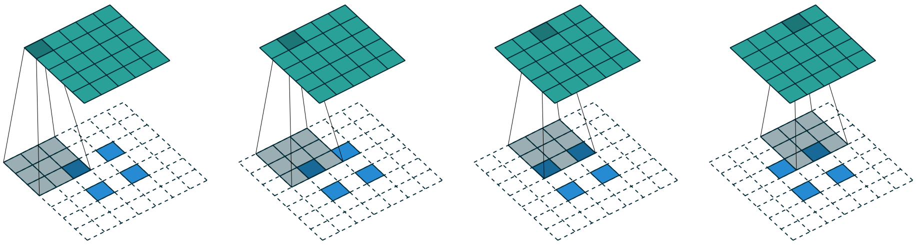


Figure 2.25: *The transpose of convolving a* 3×3 *kernel over a* 5×5 *input using* 2×2 *strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 2$ *and* $p = 0$*). It is equivalent to convolving a* 3×3 *kernel over a* 2×2 *input (with* 1 *zero inserted between inputs) padded with a* 2×2 *border of zeros using unit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

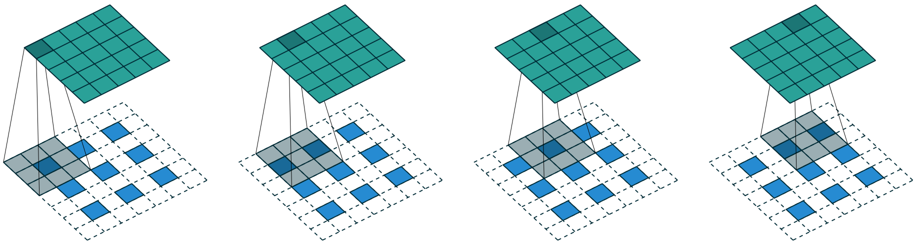


Figure 2.26: *The transpose of convolving a* 3×3 *kernel over a* 5×5 *input padded with a* 1×1 *border of zeros using* 2×2 *strides (i.e.,* $i = 5$*,* $k = 3$*,* $s = 2$ *and* $p = 1$*). It is equivalent to convolving a* 3×3 *kernel over a* 2×2 *input (with* 1 *zero inserted between inputs) padded with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

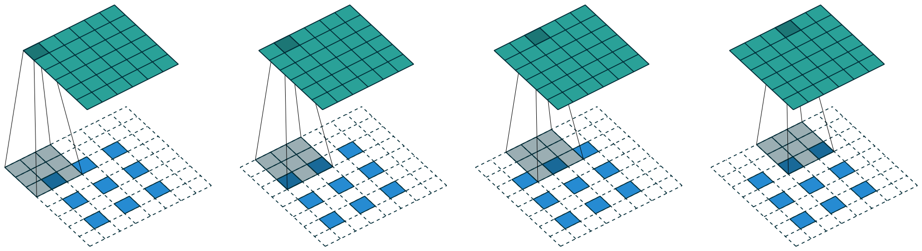


Figure 2.27: *The transpose of convolving a 3×3 kernel over a 6×6 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 6$, $k = 3$, $s = 2$ and $p = 1$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 1×1 border of zeros (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 $p' = 1$).*

2.4 Recurrent networks

It is well known that the brain is organized in functional areas, each of them usually structured in *layers* that process the incoming signal in an incremental fashion. These areas are usually 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). Similarly, ANNs usually employ either feedforward or feedback connections.

Write something about RNNs, LSTMs, GRUs

CHAPTER 3

ReNet

3.1 Introduction

Write something about ReNet

CHAPTER 4

ReSeg

4.1 Introduction

Write something about ReSeg

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

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*.

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

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

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

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.

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*.

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

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

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning.

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

Bibliography

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

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*.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012a). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'2012)*.

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.

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

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.

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

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

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*.

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

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

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

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.

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

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

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*.

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

Visin, F., Kastner, K., Courville, A. C., Bengio, Y., Matteucci, M., and Cho, K. (2015b). Reseg: A recurrent neural network for object segmentation.

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

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.

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

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